软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(5):1464−1480 [doi: 10.13328/j.cnki.jos.005723]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 SOM 神经网络的二阶变异体约简方法*

宋 利, 刘 靖

(内蒙古大学 计算机学院,内蒙古 呼和浩特 010021)

通讯作者: 刘靖, E-mail: liujing@imu.edu.cn

摘 要: 二阶变异测试通过向源程序中人工注入两个缺陷来模拟程序实际的复杂缺陷,在软件测试中具有重要意义.但由一阶变异体组合形成二阶变异体后数量会急剧增长,极大地增加了程序的执行开销.为了减少二阶变异体数量,降低程序的执行开销,提出一种基于 SOM 神经网络的二阶变异体约简方法.该方法首先采用较为全面的二阶变异体错误组合策略,对一阶变异体组合形成二阶变异体;然后,根据二阶变异体执行过程中的中间值相似性,进行基于 SOM 神经网络的变异体聚类.使用经典的基准程序和开源程序进行了方法的验证,实验结果表明,一方面,使用错误覆盖更为全面的组合策略能够充分模拟程序的复杂缺陷,聚类约简后,二阶变异体的个数在极大减少的同时,二阶变异充分度和一阶变异充分度更加接近,但是因为执行的二阶变异体数目明显降低,从而使得运行聚类后的二阶变异体时间开销明显比执行全部二阶变异体降低;另一方面,实验过程发现了有利于增加测试组件的隐藏二阶变异体.

关键词: 变异测试;二阶变异体;SOM 神经网络;变异体聚类;变异体约简

中图法分类号: TP311

中文引用格式: 宋利,刘靖.基于 SOM 神经网络的二阶变异体约简方法.软件学报,2019,30(5):1464−1480. http://www.jos.org.cn/1000-9825/5723.htm

英文引用格式: Song L, Liu J. Research on second-order mutant reduction based on SOM neural network Ruan Jian Xue Bao/ Journal of Software, 2019,30(5):1464−1480 (in Chinese). http://www.jos.org.cn/1000-9825/5723.htm

Second-order Mutant Reduction Based on SOM Neural Network

SONG Li, LIU Jing

(College of Computer Science, Inner Mongolia University, Hohhot 010021, China)

Abstract: Second-order mutation testing simulates the actual complex defects in the original program by manually injecting two defects into the original program, which is of great significance in the mutation testing. However, the number of second-order mutants formed by the combination of first-order mutants will greatly increase, which will bring large execution costs. In order to reduce the number of second-order mutants and reduce the time consumption in the running procedure, this study proposes a method of second-order mutant reduction based on SOM neural network. The proposed method firstly utilizes a morecomprehensive combination strategy to generate feasible second order mutants based on traditional first-order mutant generation, and then construct accurate SOM neural network according to the similarity of intermediate values in the execution of second-order mutants, and at last mutants are clustered based on such model to achieve second-order mutant reduction and subtle mutant detection. This study uses the benchmark and open source projects to verify the method. Experimental results show that on the one hand, although the number of mutants is very large, it has decreased significantly through the SOM neural network, while the second-order mutation score level is the same as the pre-unclustered mutation score. However, because the number of second-order mutants performed is significantly reduced, the time cost of mutation testing was greatly lower than the execution of all mutants. On the other hand, subtle second-order mutants that facilitate the addition of test components are found.

* 基金项目: 国家自然科学基金(61662051, 61262017)
Foundation item: National Natural Science Foundation of China (61662051, 61262017)
本文由智能化软件新技术专刊特约编辑申富饶教授和李戈副教授推荐.
收稿时间: 2018-08-31; 修改时间: 2018-10-31; 采用时间: 2018-12-13

Key words: mutation testing; second-order mutant; SOM neutral network; mutant clustering; mutant reduction

变异测试[1]是一种通过向源程序中人工注入缺陷来模拟软件中实际缺陷的技术,被注入缺陷的程序称为变异体.传统的变异测试是指一阶变异测试,即通过向源程序中人工注入一个缺陷来模拟软件中的一个缺陷[2].但程序中的实际缺陷往往是由两个或者更多个缺陷一起造成的[3,4],所以高阶变异测试被提出来用以解决程序中复杂缺陷的问题,在模拟程序的复杂缺陷方面有着重要意义[5].

由一阶变异体组合形成的高阶变异体数量会大量增加,大大增加了执行开销[6,7].因此,减少高阶变异体数量和发现隐藏高阶变异体(subtle higher-order mutant)已经成为当前高阶变异测试的研究热点[8].

启发式算法被应用于高阶变异测试中解决高阶变异体数量[9−11]的问题,有效减少了高阶变异体的数量,降低了程序的执行开销,但却会存在未被发现错误组合的缺点,这将降低测试组件质量.启发式算法也被应用于解决发现隐藏高阶变异体的问题[12,13],虽克服了未被发现错误组合的缺点,但却增加了时间开销.与此同时,聚类法被证明在变异体约简上有很好的效果[14,15].Ma 等人[14]使用聚类法在不影响变异充分度的前提下减少了变异体数量,降低了程序的执行开销,但在运用到高阶变异测试时却因缺乏传值而受限.针对该问题,本文着重研究二阶变异测试.本文分析了影响二阶变异体执行过程中中间值相似性的因素,并将该中间值相似性描述为条件下二阶等价变异体,使用基于 SOM 神经网络模型[16,17]的方法进行变异体聚类以达到充分考虑二阶变异体组合时减少二阶变异体执行开销和发现隐藏二阶变异体(subtle second-order mutant,本文定义为二阶变异体中单独的一阶变异体均被测试例杀死,但两者结合后的二阶变异体不能被现有测试例杀死的二阶变异体)的效果.

本文第 1 节介绍相关工作.第 2 节是本文方法中用到的基础概念.第 3 节对所提方法进行具体的介绍.第 4 节对所提方法进行理论分析.第 5 节给出实验结果及分析.第 6 节是本文的结论.

1 相关工作

在变异测试中,如果在同一测试例下,变异体的结果和原始程序的结果不同,则称该变异体被杀死,否则它是存活的.被杀死的变异体个数与所有的非等价变异体个数的比值称为变异充分度[2].传统的变异体是指通过在原始程序中注入一个缺陷而产生的一阶变异体.实际上,Purushothaman 和 Perry 已经证明,程序中存在的许多缺陷都是复杂的,这类缺陷无法使用一阶变异体进行模拟.为了解决该问题,高阶变异体则通过在原始程序中注入两个或更多个故障来模拟程序中的复杂缺陷.然而高阶变异测试的成本却很高,因此,降低高阶变异测试的成本是非常重要的.

关于降低高阶变异测试成本方法的研究已经有很多学者在探索,综述文献[8]对相关研究进行了很好的总结分析.其中,减少高阶变异体数量的方法主要有 3 种不同类型:第 1 种是选择有价值的高阶变异体,如基于搜索的软件工程的遗传算法[5];第 2 种是减少变异算子的数量,例如 last-to first 的变异算子选择算法和互不同的变异算子选择算法[9,10];第 3 种是减少变异位置的个数,如数据流技术[11,18].关于发现隐藏高阶变异体的方法[8]则更多的关注于使用基于搜索的软件工程中的贪婪和遗传算法的方法来解决[6,12,13].

与此同时,聚类法已经在变异测试中得到了很好的应用[14].该方法综合考虑了动态变异体和变异体筛选的优点.研究表明,该方法在减少变异体数量方面具有很好的效果.虽然聚类法已经在变异测试领域取得了很大进展[14,15,19,20],但是该方法更注重减少一阶变异体的数量并且缺乏中间结果的保存.因此,我们在二阶变异测试中尝试采用一种新的方法,即应用 SOM(self-organizing map)神经网络方法进行聚类.

SOM 神经网络[16]是通过模拟大脑神经系统自组织特征映射功能进行的一种无监督竞争式学习前馈网络,经过神经网络学习提取数据中的重要特征或某种内在规律,对数据进行聚类.而 SOM 神经网络的学习算法则是通过模拟生物神经元之间的兴奋、协调与抑制、竞争作用的信息处理的动力学原理来指导网络的学习与工作.所以该网络具有很强的聚类能力,并且在各领域得到了广泛的应用[17].

本文分析了影响二阶变异体执行中间值相似性的因素.基于因素分析,提出了一种使用 SOM 神经网络聚类模型进行变异体约简及发现隐藏二阶变异体的方法.具体而言,首先对不同变异点处的变异体组合成二阶变异体,在此基础上进行基于 SOM 神经网络聚类模型的变异体约简,根据不同变异点组合中间结果是否一样为一

类,然后从每一簇中选择一组变异体进行继续执行,达到对二阶变异体进行聚类和发现隐藏二阶变异体的效果.

2 条件下二阶等价变异体

二阶变异体的约简原理,主要是根据二阶变异体执行过程中的中间值相似性,即条件下二阶等价变异体进行的,所以本节就条件下二阶等价变异体的概念给出相关定义和示例.

2.1 概念定义

a) 等价变异体:指对于任何测试数据,执行源程序和变异体后,输出结果相同的变异体[20];

b) 变异体被杀死:指对于相同的测试数据,分行执行变异体和源程序,若变异体和源程序的执行结果不同,则称该变异体被杀死[20];

c) 条件下等价变异体(conditionally overlapped mutants):指对于相同的测试用例,一组一阶变异体的中间结果相同,但是有可能会被该测试用例杀死的一组变异体[14];

d) 二阶变异体的中间结果:本文定义为仅运行至二阶变异体中两个变异点后得到的结果,而非运行完整个程序后得到的结果,记为 intermediate results of second-order mutants,简单记为 *Irs*;

e) 条件下二阶等价变异体:本文定义为对于相同的测试用例,一组二阶变异体的中间结果相同,但可能会被该测试用例杀死的一组二阶变异体,记为 second-order conditionally overlapped mutants,简单记为 *Second_CoM*.

2.2 条件下二阶等价变异体示例

为易于理解本文所提出的"条件下二阶等价变异体"概念,以图 1(a)所示 Java 片段为例进行说明.先使用关系运算符 ROR 和算术运算符 AOR 分别作用于程序中的第 3 行、第 7 行,生成各自的一个一阶变异体如图 1(b)、图 1(c)所示;再同时作用于第 3 行和第 7 行生成一个二阶变异体如图 1(d)所示.

```
1 int my(int A,int B){
2 int C;
3 if (A>B)
4 C=C+A;
5 else
6 C=C−A;
7 C=C+B;
8 return C;
9 }
```
(a)

```
1 int my(int A,int B){
2 int C;
3 if (A<B)
4 C=C+A;
5 else
6 C=C−A;
7 C=C+B;
8 return C;
9 }
```
(b)

```
1 int my(int A,int B){
2 int C;
3 if (A>B)
4 C=C+A;
5 else
6 C=C−A;
7 C=C*B;
8 return C;
9 }
```
(c)

```
1 int my(int A,int B){
2 int C;
3 if (A<B)
4 C=C+A;
5 else
6 C=C−A;
7 C=C*B;
8 return C;
9 }
```
(d)

Fig.1 A Java example for related concepts

图 1 相关概念的 Java 示例

以(*A*=4,*B*=2)作为测试数据分别执行源程序和二阶变异体,得到的测试数据信息见表 1 所列.

由表 1 可知,一共生成了 28 个二阶变异体,其中最后一列分别为 28 个二阶变异体的中间结果.根据条件下二阶等价变异体 *Second_CoM* 的概念可知,运行全部程序只需 6 个二阶变异体即可.理由如下:由变异算子组合 ID 为 1,5,17,21 的二阶变异体的中间结果为–8;变异算子组合 ID 为 2,6,18,22 的二阶变异体的中间结果为–6;变异算子组合 ID 为 3,7,19,23 的二阶变异体的中间结果为–2;变异算子组合 ID 为 4,8,12,16,20,24,28 的二阶变异体的中间结果为 0;变异算子组合 ID 为 9,13,25 的二阶变异体的中间结果为 8;变异算子组合 ID 为 10,11,14,15,26,27 的二阶变异体的中间结果为 2.即该测试例下,共有 6 组条件下二阶等价变异体,每组条件下二阶等价变异体具有相同的被杀死可能性.所以只需从每一组中选择一个二阶变异体继续运行即可,因此,运行整个程序只需 6 个二阶变异体.

从示例可以看出,对于条件下二阶等价变异体,因为条件下二阶等价变异体中的二阶变异体中间结果的相似性,所以继续执行程序至最终结果时也一样.因此只需从每一簇中选择一个二阶变异体继续运行即可,这与执行全部的二阶变异体相比会节省很多时间.所以本文提出使用 SOM 神经网络对条件下二阶等价变异体进行聚

类,以减少执行二阶变异体的时间.

Table 1 Intermediate results of second-order mutants for the example

表 1 示例中二阶变异体的中间结果

ID	变异点所在行	变异算子组合	代码变化	二阶变异体的中间结果
1	3~7	ROR_1-AOR_1	*A*<*B*, *C*=*C***B*	–8
2	3~7	ROR_1-AOR_2	*A*<*B*, *C*=*C*–*B*	–6
3	3`7	ROR_1-AOR_3	*A*<*B*, *C*=*C*/*B*	–2
4	3~7	ROR_1-AOR_4	*A*<*B*, *C*=*C*%*B*	0
5	3~7	ROR_2-AOR_1	*A*≤*B*, *C*=*C***B*	–8
6	3~7	ROR_2-AOR_2	*A*≤*B*, *C*=*C*-*B*	–6
7	3~7	ROR_2-AOR_3	*A*≤*B*, *C*=*C*/*B*	–2
8	3~7	ROR_2-AOR_4	*A*≤*B*, *C*=*C*%*B*	0
9	3~7	ROR_3-AOR_1	*A*≥*B*, *C*=*C***B*	8
10	3~7	ROR_3-AOR_2	*A*≥*B*, *C*=*C*–*B*	2
11	3~7	ROR_3-AOR_3	*A*≥*B*, *C*=*C*/*B*	2
12	3~7	ROR_3-AOR_4	*A*≥*B*, *C*=*C*%*B*	0
13	3~7	ROR_4-AOR_1	*A*≠*B*, *C*=*C***B*	8
14	3~7	ROR_4-AOR_2	*A*≠*B*, *C*=*C*–*B*	2
15	3~7	ROR_4-AOR_3	*A*≠*B*, *C*=*C*/*B*	2
16	3~7	ROR_4-AOR_4	*A*≠*B*, *C*=*C*%*B*	0
17	3~7	ROR_5-AOR_1	*A*==*B*, *C*=*C***B*	–8
18	3~7	ROR_5-AOR_2	*A*==*B*, *C*=*C*–*B*	–6
19	3~7	ROR_5-AOR_3	*A*==*B*, *C*=*C*/*B*	–2
20	3~7	ROR_5-AOR_4	*A*==*B*, *C*=*C*%*B*	0
21	3~7	ROR_6-AOR_1	False, *C*=*C***B*	–8
22	3~7	ROR_6-AOR_2	False, *C*=*C*–*B*	–6
23	3~7	ROR_6-AOR_3	False, *C*=*C*/*B*	–2
24	3`7	ROR_6-AOR_4	False, *C*=*C*%*B*	0
25	3~7	ROR_7-AOR_1	True, *C*=*C***B*	8
26	3~7	ROR_7-AOR_2	True, *C*=*C*–*B*	2
27	3~7	ROR_7-AOR_3	True, *C*=*C*/*B*	2
28	3~7	ROR_7-AOR_4	True, *C*=*C*%*B*	0

3 基于 SOM 神经网络的二阶变异体约简方法

基于 SOM 神经网络的二阶变异体约简方法的总体框架如图 2 所示.

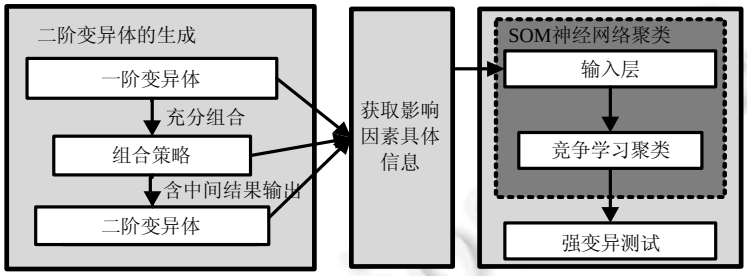

Fig.2 Framework of second-order mutant reduction based on SOM neural network

图 2 基于 SOM 神经网络的二阶变异体约简的框架

该方法主要包括 3 个步骤,本节接下来将详细阐述这 3 个步骤.

3.1 二阶变异体的产生

如图 2 所示,本文中二阶变异体的生成部分主要由以下 3 步组成.

(1) 一阶变异体生成:使用变异测试工具 muJava 进行一阶变异体的产生.

(2) 组合策略:提出使用更为全面的缺陷组合策略对一阶变异体组合形成二阶变异体.即对任意两个不同位置处的任意两个一阶变异体进行组合,以充分模拟程序中的复杂缺陷.以第 2.2 节例子进行说明,第

3 行的 7 个变异体与第 7 行的 4 个变异体进行充分组合,产生 28 个二阶变异体,能够将这两个变异点处的全部错误组合表示出来.

(3) 二阶变异体生成:首先对两个一阶变异体逐行读取,然后根据一阶变异体中变异点行数和变异算子组合等信息替换掉对应的语句形成二阶变异体.为了二阶变异体在执行测试示例时能够及时获取二阶变异体中间结果,本文对 muJava 工具的源码进行了修改,算法 1 描述了以变异算子 AOR 为例的修改.

算法 1. 含中间结果输出的变异体生成.

输入:源程序语句,原始表达式,AOR 算子作用于变异点以后的表达式.

输出:含中间结果显示的变异体的.java 文件.

```
BEGIN
/*获取变异点变异后表达式*/
String mutant_Immediatelyresult=mutant.getLeft()+mutant.operatorString()+mutant.getRight();
/*将源程序语句读取到新生成的.java 文件中*/
out.print("statement");
/*使变异点变化后的结果在变异体中有输出*/
out.print("System.out.println("+mutant_Immediatelyresult+")");
/*将变异点如位置和变异算子等信息写入变异体日志 log 中*/
CodeChangeLog.writeLog(class_name+MutationSystem.LOG_IDENTIFIER+
mutated_line+method_signature+log);
END
```

以上即为二阶变异体的生成过程,接下来将分析影响条件下二阶等价变异体的因素,从而确定 SOM 神经网络中的特征属性.

3.2 特征属性的分析和确定

本文在运用 SOM 神经网络对二阶变异体进行聚类时,首先分析了影响条件下二阶等价变异体的可能性因素,并将其作为输入层特征属性.主要从以下 3 个方面进行分析.

(1) 影响二阶变异体被杀死的原因

当二阶变异体的中间结果与源程序不同时,二阶变异体也会被杀死,所以本文根据二阶变异体的中间结果进行分析,以确定二阶变异体被杀死的原因.二阶变异体中按照两个变异点的先后顺序,分别记为第 1 变异点和第 2 变异点.图 3 所示为根据两个变异点组合的变异体的执行结果进行的分类分析.

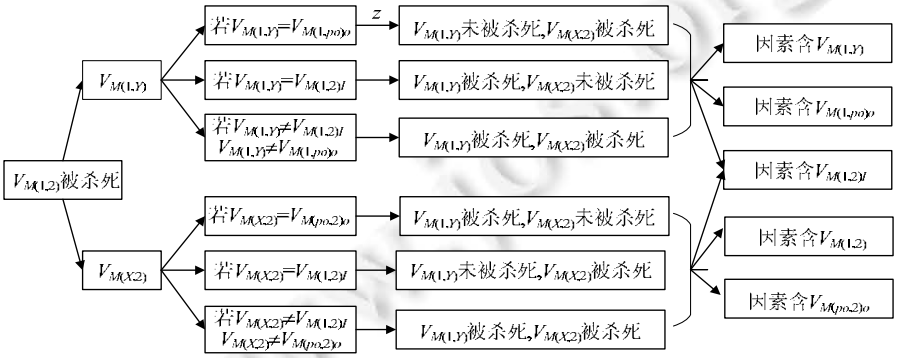

Fig.3 Analysis of the causes of second-order mutants being killed

图 3 二阶变异体被杀死的原因分析

具体分析如下.

当某二阶变异体被杀死时,可能的原因是二阶组合中的两个一阶单独或者共同造成二阶变异体被杀死的,

现分为两种情况进行分析讨论.

a. 假设第 1 变异点变异后的结果(记 $V_{M(1,Y)}$为)造成二阶变异体被杀死,则又可分如下 3 种情况分析.

- 若 $V_{M(1,Y)}$与第 1 变异点的预期结果(两者不发生变异时,由第 1 个变异点运行至第 2 变异点时的结果,记为 $V_{M(1,po)o}$)相同,则该假设"$V_{M(1,Y)}$造成二阶变异体被杀死"不成立.又因二阶变异体被杀死,所以在 $V_{M(1,Y)}=V_{M(1,po)o}$ 条件下,造成二阶变异体被杀死的原因是第 2 变异点变异后的结果(记为 $V_{M(X,2)}$)造成的.即 $V_{M(1,Y)}$未被杀死,$V_{M(X,2)}$被杀死时导致了二阶变异体被杀死.
- 若 $V_{M(1,Y)}$与二阶变异体的中间值(两个变异点同时变异时,运行至第 2 变异点时的结果,记为 $V_{M(1,2)I}$)相同,所以 $V_{M(1,Y)}$与源程序结果不同,导致 $V_{M(1,Y)}=V_{M(1,2)I}$.又因 $V_{M(1,2)I}$ 的结果与源程序不同,所以 $V_{M(1,Y)}$造成了二阶变异体被杀死.即 $V_{M(1,Y)}$被杀死,$V_{M(X,2)}$未被杀死时导致了二阶变异体被杀死.
- 若 $V_{M(1,Y)}\neq V_{M(1,po)o}$,$V_{M(1,Y)}\neq V_{M(1,2)I}$ 说明 $V_{M(1,Y)}$与源程序的第 1 变异点的预期结果不同,所以导致 $V_{M(1,Y)}\neq V_{M(1,po)o}$;但是 $V_{M(1,Y)}\neq V_{M(1,2)I}$,说明 $V_{M(X,2)}$在 $V_{M(1,Y)}$上再次改变了二阶变异体的中间值,导致 $V_{M(1,Y)}\neq V_{M(1,2)I}$.即 $V_{M(1,Y)}$被杀死,$V_{M(X,2)}$被杀死时导致了二阶变异体被杀死.即 $V_{M(1,Y)}$被杀死,$V_{M(X,2)}$被杀死时导致了二阶变异体被杀死.

b. 假设第 2 变异点变异后的结果($V_{M(X,2)}$)时造成的,则又可分如下 3 种情况分析.

- 若 $V_{M(X,2)}$与第 2 变异点的预期结果(两者不发生变异时,运行至第 2 变异点时的结果,记为 $V_{M(po,2)o}$)相同,则该假设"$V_{M(X,2)}$造成二阶变异体被杀死"不成立.又因为二阶变异体被杀死,所以在 $V_{M(X,2)}=V_{M(po,2)o}$ 条件下,造成二阶变异体被杀死的原因是 $V_{M(1,Y)}$造成的.即 $V_{M(1,Y)}$被杀死,$V_{M(X,2)}$未被杀死时导致了二阶变异体被杀死.
- 若 $V_{M(X,2)}=V_{M(1,2)I}$,所以 $V_{M(X,2)}$与源程序结果不同,导致 $V_{M(X,2)}=V_{M(1,2)I}$.又因 $V_{M(1,2)I}$ 的结果与源程序不同,所以 $V_{M(X,2)}$造成二阶变异体被杀死.即 $V_{M(1,Y)}$未被杀死,$V_{M(X,2)}$被杀死时导致了二阶变异体被杀死.
- 若 $V_{M(X,2)}\neq V_{M(po,2)o}$,$V_{M(X,2)}\neq V_{M(1,2)I}$,说明 $V_{M(X,2)}$与源程序的第 2 变异点的预期结果不同,故导致 $V_{M(X,2)}\neq V_{M(po,2)o}$;但 $V_{M(X,2)}\neq V_{M(1,2)I}$,说明 $V_{M(1,Y)}$在 $V_{M(X,2)}$上再次改变了二阶变异体的中间值,导致 $V_{M(X,2)}\neq V_{M(1,2)I}$.即 $V_{M(1,Y)}$被杀死,$V_{M(X,2)}$被杀死时导致了二阶变异体被杀死.

分析了二阶变异体的中间结果被杀死的原因后,可知影响二阶变异体的中间结果相似性,即影响条件下二阶等价变异体的因素含 $V_{M(1,Y)}$,$V_{M(1,po)o}$,$V_{M(X,2)}$,$V_{M(po,2)o}$,$V_{M(1,2)I}$.

(2) 分析直观的影响因素

众所周知:当二阶变异体被杀死时,与两个变异体在程序中的位置以及变异算子等有关.现在以经典的三角形程序中的部分代码为例进行说明,如图 4 所示.

```
...
1  If (a≥0||b≤0||c≤0)                          变异后为真时
2          return INVALID;
3  tri=0;                                  第二变异体变异结果对
4  If (a==b)                               二阶变异体最终结果无影响
5          tri=tri−1;
6  If (a==c)
7          tri=tri+2;
8  If (b==c)
9          tri=tri+3;
10 If (tri==0)
11         If (a+b<c||a+c<b||b+c<a)
12                 return INVALID;
13         else
14                 return SCALENE;
15 If (tri>3)
16         return EQUILATERAL;
...
```

Fig.4 Analysis of intuitive factors

图 4 直观影响因素的分析

对程序的第 1 行、第 5 行语句分别施加 ROR 变异算子和 AOR 变异算子,可以得到:如果第 1 变异体为真时,则第 2 变异体的结果对程序的最终结果无影响;当第 1 行语句为真时,则第 3 行~第 16 行任意变异点的结果均对最终结果无影响.所以,影响二阶变异体中间结果的因素含测试用例、变异点位置、变异算子、两个变异体的距离.

(3) 影响隐藏二阶变异体的因素

根据隐藏二阶变异体的定义,即二阶变异体中单独的第 1 变异体和第 2 变异体均被杀死,但两者组合后的二阶变异体没有被杀死的二阶变异体.因此,因素中必含两个变异点分别变异前运行至对应变异点时的预期结果(记为 $V_{P(1,po)o}$ 和 $V_{P(po,2)o}$)、两个变异点分别变化后运行至对应变异点时的结果(记为 $V_{M(1,po)I}$ 和 $V_{M(po,2)I}$)、两个变异点同时发生变异后的结果(即 $V_{M(1,2)I}$).注意,$V_{M(1,po)o}$ 和 $V_{P(1,po)o}$ 的区别是:$V_{M(1,po)o}$ 表示只有第 1 个变异点发生变异、第 2 个变异点未发生变异时的预期结果;$V_{P(1,po)o}$ 表示源程序中任意两个变异点均未发生变异时,第 1 个变异点的预期结果.同理,$V_{M(po,2)o}$ 和 $V_{P(po,2)o}$ 的区别也是如此.

综上所述,本文使用 SOM 神经网络聚类时的特征属性为:测试用例、变异点位置、变异算子、两个变异体的距离、$V_{M(1,Y)}$、$V_{M(1,po)o}$、$V_{M(X,2)}$、$V_{M(po,2)o}$、$V_{M(1,2)I}$、$V_{P(1,po)o}$、$V_{P(po,2)o}$、$V_{M(1,po)}$、$V_{M(po,2)o}$.

3.3 SOM神经网络的设计

通过第 2.2 节的分析,确定了影响二阶变异体中间结果的因素.接下来建立适用于二阶变异体的 SOM 神经网络.

根据第 2.1 节中的方法获取到一阶变异体和二阶变异体的中间结果的信息,将这些信息作为输入层初始数据.对其中的非数值信息,进行 ASCII 编码转换.本文中的 SOM 神经网络是在 PyCharm 中实现,算法的主要步骤和实现如图 5 所示.

```
#1 、 对数据集和权值矩阵进行归一化处理
dataSet, old_dataSet = normalize(dataSet)
com_weight = normalize_weight(com_weight)
# 遍历归一化后的数据
for data in dataSet:
#2、 得到获胜神经元
    n,m= getWinner(data , com_weight )
# 得到神经元的 N 邻域
  neibor = getNeibor(n,m,N_neibor, com_weight)
  for x in neibor :
    j_n=x[0];j_m=x[1];N=x[2]
     #3 、 权值调整
    com_weight[j_n][j_m]=com_weight[j_n][j_m]+eta(t,N)*
                          (data−com_weight[j_n][j_m])
```

Fig.5 Implementation of SOM neural network algorithm

图 5 SOM 神经网络算法的实现

其核心学习算法每一步的具体实现如下.

(1) 初始化

令获取到的信息为输入数据集 *dataSet*,从训练集中随机取一输入模式并进行归一化处理;竞争层中,各神经元对应的内星权向量为 $com\_weight_j(j=1,2,...,m)$,$m$ 为输出层神经元数目,对其进行归一化处理.归一化的方式如公式(1)和公式(2),建立初始优胜邻域 $N_j^*(0)$ 和学习率初值设为 0.9.

$$\widehat{dataSet}=\frac{dataSet}{\|dataSet\|} \tag{1}$$

$$\widehat{com\_weight}_j=\frac{com\_weight_j}{\|com\_weight_j\|} \tag{2}$$

(2) 寻找获胜神经元

根据影响二阶变异体的中间结果的因素,本文中,获胜神经元的计算方式为计算输入样本与权值向量的欧

几里得距离(记为 d_j).即计算输入数据集 *dataSet* 归一化后的数据与竞争层中所有神经元对应的内星权向量 $com_weight_j(j=1,2,...,m)$的欧几里得距离,如公式(3):

$$d_j = \| \widehat{dataSet} - \widehat{com\_weight}_j \| = \sqrt{\sum_{j=1}^{m} [\widehat{dataSet} - \widehat{com\_weight}_j]^2} \tag{3}$$

距离最小的神经元赢得竞争,即距离越小的神经元相似度越大,更容易聚为一类.

(3) 定义优胜邻域 $N_j^*(t)$

以获胜神经元为中心设定一个邻域半径,该半径确定的范围称为优胜邻域.设以 j^*为中心的 t 时刻的优胜邻域为 $N_j^*(t)$,一般初始邻域 $N_j^*(0)$ 较大,训练过程中 $N_j^*(t)$ 随着训练时间的增加而收缩.

(4) 调整权值

对优胜邻域 $N_j^*(t)$ 内的所有节点进行权值调整,调整方法如公式(4)所示.

$$com\_weight_{ij}(t+1) = \widehat{com\_weight}_{ij}(t) + eta(t,N)\left(\widehat{dataSet} - \widehat{com\_weight}_{ij}(t)\right), i = 1,...,n,\ j \in N_j^*(t) \tag{4}$$

其中, $\widehat{com\_weight}_{ij}(t)$ 表示神经元 i 在 j 时刻的权值;$0<eta(t,N)<1$ 为学习率,是一个邻域内第 i 个神经元与获胜神经元 j^*之间的拓扑距离 N 的函数,随着学习进展而减少,即调整的程度越来越小,趋于聚类中心.

(5) 重新归一化处理

权向量经过调整后,得到的新向量不再是单位向量,因此要对学习调整后的向量进行重新归一化,循环运算,当 $eta(t,N) \leqslant eta_{\min}$ 时结束训练.不满足时重新进行步骤(1),直到学习率为 0.

最后,使用聚类后的约简过的二阶变异体进行程序的完整执行,即强变异测试.

以上 3 小节即为本文所提方法的 3 个步骤,能充分模拟程序中的复杂缺陷,并且减少程序执行开销的预期效果.接下来,从理论上进行分析和证明所提方法的正确性和有效性.

4 理论分析

为了更加清晰和唯一确定地理解所提出的方法,该节采用 Shin 等人[21]提出的变异测试的数学理论框架,对本文所提方法的核心理论进行分析和证明.该框架是以程序与程序的差异性为基础展开的,而非程序的正确性.即该框架具有能够说明两个变异体是否能够产生相同结果的功能,而非仅限于说明某个(些)变异体的结果与源程序之间是否相同.同时,所提方法也是基于二阶变异体的中间结果的比较进行的,所以可以使用该框架进行理论分析和证明.本节将从两方面进行数学分析和证明,即变异体约简和发现隐藏二阶变异体.

4.1 变异体约简理论分析

为方便分析和证明,从基于 SOM 神经网络的聚类结果中的每簇中任选一个二阶变异体用以构成最小二阶变异体集时,以选取每簇中的第 1 个二阶变异体为例进行分析(实际上,从每簇二阶变异体中选取的需要完全执行的二阶变异体是随机的).给定全部的二阶变异体和测试用例集.假设产生的全部二阶变异体一共有 M_{all} 个,聚类后有 N 个簇,分别编号为 $M1,M2,...,MN$,也就是 $S\_Coms=\{M1,M2,...,MN\}$;假设簇 $M1$ 中现有变异体 m 个,分别简单编号为 $M1_{x1},M1_{x2},...,M1_{xm}$;假设 N 个簇中的任意簇中的二阶变异体的个数为 y,例如选取 $M1$ 中的二阶变异体的个数为 y.现选取簇 $M1$ 中的任意一个变异体作为原点,例如以 $M1_{x1}$ 作为原点,则簇 $M1$ 中的所有变异体的中间结果之间相似性可以用数学表示为

$$\forall M1_{x1} \in M1, \exists t \in TS, \neg d(t, M1_{x1}, M1_{xi}) = 1 (i=2,3,\ldots,m) \tag{5}$$

其中,d 表示测试差分(简称差分).测试差分表示了程序之间的差异性[21],可用公式(6)[21]表示.

$$d(t, px, py) = \begin{cases} 1, & px\text{和}py\text{运行结果不同} \\ 0, & px\text{和}py\text{运行结果相同} \end{cases} \tag{6}$$

在理论框架中,原点对于公式所表示的内容具有重要意义,而公式(5)中选取 $M1_{x1}$ 作为原点,则表示 $M1_{xi}$ 与 $M1_{x1}$ 的差异性.

公式(6)显示了在相同测试用例下,如果两个程序的运行结果不同,差分 d 记为 1;否则记为 0.类似的,采用该

框架表示本文中变异体之间的差异性时,可用如下公式表示.

$$\forall M1_{xi}\in M_{all},\forall Mj_{xk}\in(M_{all}-M1),\exists t\in TS,\text{使得}:\begin{cases}d(t,M1_{x1},M1_{xi})=0,\ i=2,3,...,m\\ d(t,M1_{x1},Mj_{xk})=1,\ j=2,3,...,N;k=1,2,...,y\end{cases} \tag{7}$$

其中,Mj_{xk} 表示某一簇中的任意一个二阶变异体.该公式显示了同一簇中的二阶变异体具有相同的二阶变异体的中间结果(即 *Irs*).由于同一簇中的二阶变异体具有相同的 *Irs*,因此测试差分的值为 0.不同簇中的二阶变异体具有不同的 *Irs*,所以测试差分 *d* 的值为 1.

结合公式(5)和公式(7),本文中的二阶变异体的约简原理可作如下推导.

$$\forall mi\in M_{all},\exists t\in TS,d(t,m1,mi)=1 \tag{8}$$

$$\Rightarrow\forall mi\in\sum_{i=1}^{n}Mi,\exists t\in TS,d(t,m1,mi)=1 \tag{9}$$

$$\approx\forall mi\in\sum_{i=1}^{n}Mi_{x1},\exists t\in TS,d(t,M1_{x1},mi)=1 \tag{10}$$

$$\Rightarrow\text{执行全部二阶变异体与约简后二阶变异体比较,结果相同} \tag{11}$$

其中,公式(8)中,M_{all} 为全部二阶变异体;公式(9)中,*Mi* 是第 *i* 簇二阶变异体;公式(10)中,Mi_{x1} 表示第 *i* 簇中的一个二阶变异体.结合本文选取和确定的特征属性决定了每簇中的二阶变异体具有相似的功能,若该集合具有某种功能,则该集合中的任意一个二阶变异体均具有相似的功能.每簇中选取的二阶变异体构成的新的集合具有全部二阶变异体的功能.

综上所述,理论上,采用本文方法聚类后的二阶变异体个数会减少很多,且不影响原来二阶变异体的测试总结果(例如错误组合的表示情况).

4.2 隐藏二阶变异体理论分析

所提方法发现隐藏二阶变异体的方法和执行步骤为:使用 SOM 神经网络聚类后,找到某一测试例下的隐藏二阶变异体;找出所有测试例下的隐藏二阶变异体,提取出不同测试例下相同的隐藏二阶变异体,即为找到的隐藏二阶变异体.这些隐藏二阶变异体不能被现有的测试用例杀死.其中,对于某一测试用例下隐藏二阶变异体的数学模型为

$$\exists mi\in M_{all},\forall t\in TS,\text{使得}:\begin{cases}d(t,po,mi(1,po))=1\\ d(t,po,mi(po,2))=1\\ d(t,po,mi(1,2))=0\end{cases} \tag{12}$$

其中,*po* 为源程序;*mi*(1,*po*)表示第 1 个变异点处变异体结果被杀死,第 2 个变异点未发生变异的变异体.由公式(6)可知,测试差分的值表示该簇中任意一个二阶变异体与测试用例的运行结果情况,所以公式(12)可用于表示某测试例下的隐藏二阶变异体.

但是,使用本文方法时需要找出所有测试例下的隐藏二阶变异体,即需找到满足公式(13)的二阶变异体.

$$\exists mi\in M_{all},\forall\vec{t}\in TS,\text{使得}:\begin{cases}\vec{d}(\vec{t},po,mi(1,po))=\langle1,0,...,1\rangle\\ \vec{d}(\vec{t},po,mi(po,2))=\langle1,0,...,1\rangle\\ \vec{d}(\vec{t},po,mi(1,2))=\langle0,0,...,0\rangle\end{cases} \tag{13}$$

其中,此处的测试用例和差分均使用了矢量,指多个测试用例下的隐藏二阶变异体[21].差分矢量结果中若有混合的 1 和 0,则它们的顺序是任意的(因为变异体可能是被任意测试用例杀死,而不局限于被第 1 个和最后一个测试用例杀死),此处均记为⟨1,0,...,1⟩;若只有 0,则没有测试用例可以杀死该变异体.针对多个测试用例下找到隐藏二阶变异体的过程可使用程序空间[21]进行分析,对程序空间改进后,发现隐藏二阶变异体的过程如图 6.其中,深黑色圆圈指变异体被测试用例杀死,浅灰色圆圈指未被杀死.图 6(1)~图 6(3)表示了某二阶变异体在只有一个变异点被杀死时,该二阶变异体能够被 3 个测试例 *t*1~*t*3 中任意一个测试例杀死;但是如果两个变异点均发生变异后,该二阶变异体反而不能被 3 个测试用例杀死,即图 6(4)所示.此时需添加新的测试用例来杀死该隐藏二阶变异体,即如图 6(5)所示;图 6(6)表示推广到二阶变异体中有多个隐藏二阶变异体,然后针对不同隐藏二阶变异体设计新的测试用例(如图 6(7)所示),使得该隐藏二阶变异体被杀死,用以提高测试组件的质量.

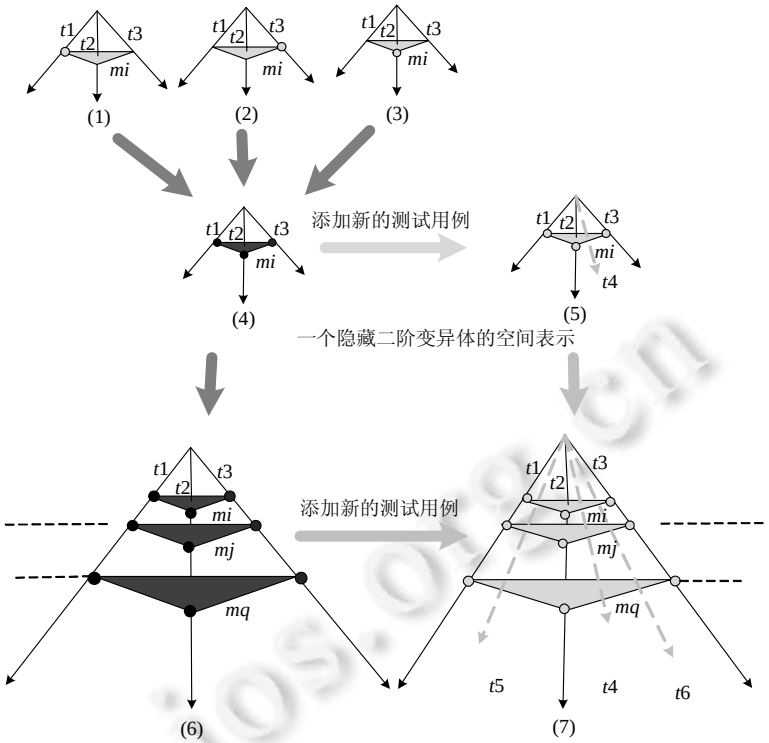

Fig.6 Mathematical model of finding subtle second-order mutants

图 6 发现隐藏二阶变异体的数学模型

针对多个测试用例下的多个隐藏二阶变异体添加新的测试例前后的变化,使用差分矢量可表示为

$$\begin{array}{ll}\exists mi\in M_{all},\forall \vec{t}\in TS,\text{使得:} & \exists mi\in M_{all},\forall \vec{t}'\in TS',\text{使得:}\\ \begin{cases}\vec{d}(\vec{t},po,mi(1,po))=\langle 1,0,...,1\rangle\\ \vec{d}(\vec{t},po,mi(po,2))=\langle 1,0,...,1\rangle\\ \vec{d}(\vec{t},po,mi(1,2))=\langle 0,0,...,0\rangle\end{cases}\xrightarrow{\vec{t}'=t\cup new(\vec{t})} & \begin{cases}\vec{d}(\vec{t}',po,mi(1,po))=\langle 1,0,...,1\rangle\\ \vec{d}(\vec{t}',po,mi(po,2))=\langle 1,0,...,1\rangle\\ \vec{d}(\vec{t}',po,mi(1,2))=\langle 1,0,...,0\rangle\end{cases}\end{array} \tag{14}$$

其中,$\vec{t}'$ 为添加新测试用例后的测试组件.前后差别最大的地方在于 $\vec{d}(\vec{t}',po,mi(1,2))=\langle 1,0,...,0\rangle$,即两个变异点均发生变异后的变异体也可以被新设计添加的测试用例杀死.

通过使用数学理论框架对本文所提方法进行的分析和理论证明,使得本文所提方法的核心思想更加地容易理解.更重要的是,使用框架中的元素对本文中的方法进行了推导证明,证明了方法的正确性和有效性.接下来,将从实验上验证所提方法的实际效果.

5 实验及结果分析

5.1 实验准备

(1) 程序和实验环境

本文首先选取两个经典的基准程序,即三角形(Triangle)[7,11,14]和找中间数(Mid)[7,11,14]来进行验证,然后使用规模较大的实用性开源项目 Snake 和小组团队开发程序 Shopping 来验证方法的可应用性和有效性.实验运行环境为 Eclipse 集成开发环境,实验中的一阶变异体由修改后的 muJava 工具产生,聚类后变异体由 SOM 神经网络聚类后产生,基准程序中的测试用例来源于文献[11],实用性项目因为现有文献中没有相同的项目资料,所以

使用自己团队编写的测试用例用来验证.因每次对同一数据聚类的结果有细微偏差,所以本文选择其平均值.

(2) 变异算子

本文选取研究中经常使用的变异算子[7,11,14]来进行实验.在符合优秀程序员准则后的程序中,算术运算符、关系运算符和条件运算符等出错的概率比较高,所以这 7 个变异算子覆盖了多数程序语言中缺陷常出现的情况,所以这些变异算子具有一定的代表性.其信息见表 2.

Table 2 List of mutation operators

表 2 变异算子列表

变异算子	全称
AOIS	Arithmetic operator insertion (short-cut)
AOIU	Arithmetic operator insertion (unary)
LOI	Logical operator insertion
ROR	Rational operator replacement
COI	Conditional operator insertion
COR	Conditional operator replacement
AOR	Arithmetic operator replacement

其中,

- AOIS 表示算术运算符插入,包含有两个操作符,分别为“++,––”.变异点的变异例如“a⇒a––”;
- AOIU 表示一元算术运算符插入,包含一个操作符,即“–”.变异点的变异例如“a⇒–a”;
- LOI 表示逻辑运算符插入,包含一个操作符,即“~”.变异点的变异如“if (b==a)⇒if (b==~a)”;
- ROR 表示关系运算符替换,包含 6 个操作符,分别为“<,≤,>,≥,==,≠”.变异点的变异例如“a>b⇒a<b”;
- COI 表示条件运算符插入,包含一个操作符,即“!”.变异点的变异例如“a≤0⇒!(a≤0)”;
- COR 表示条件运算符替换,包含两个操作符,分别为“&&,||”.变异点的变异例如“(a&&b)⇒(a||b)”;
- AOR 表示算术运算符替换,包含 5 个操作符,分别为“*,/,%,+,–”.变异点的变异例如“(a+b)⇒(a–b).

5.2 实验结果及分析

本文的实验将从以下几个方面开展:所提方法能否减少二阶变异体个数;所提方法对变异充分度是否影响;所提方法能否降低变异测试时间开销;所提方法能否发现隐藏二阶变异体.针对以上几个方面进行如下实验.

(1) 二阶变异体个数的实验

为了验证本文方法能够减少运行的二阶变异体个数.首先用本文方法生成一阶和二阶变异体,通过对一阶和二阶变异体个数的比较,以显示二阶变异体个数的庞大,并和文献[11]方法(记为已有方法)比较.选择该文献的原因如下.

- 首先,该文献中解决的问题和本文所要解决的问题是一致的,即考虑文章中出现的错误组合,然后对二阶变异体个数进行约简;而其他文献中关于高阶变异体的约简则是通过对一阶变异体折半组合以达到减少变异体数量的目的,与本文研究的问题有稍许偏差.
- 其次,该文献作者所在研究组一直跟踪研究高阶变异测试,且该篇文章是该研究组近期发表的一篇非综述类研究高阶变异体约简的文章,该文献的工作具有较好的代表性,所以选其作为对比对象.

实验对比的具体信息见表 3.

Table 3 Number of first-order and second-order mutants generated for benchmark programs

表 3 基准程序生成的一阶和二阶变异体个数

程序	方法	一阶变异体个数	二阶变异体聚类前个数
Triangle	已有方法	219	47 557
	本文方法	219	62 369
Mid	已有方法	119	13 935
	本文方法	119	24 756

其中,已有方法生成二阶变异体的方法使用的是 LastToFirst DifferentOperators 策略,该策略忽略了某一个

错误组合后再同其他错误组合的情况,造成程序中某些错误组合被遗漏;本文充分考虑程序中可能出现的错误组合,采用较为全面的组合策略对一阶变异体进行组合生成二阶变异体;接着进行聚类前后二阶变异体个数比较的实验,使用二阶变异体减少率的形式展示,具体见表 4 和表 5.

Table 4 Number of second-order mutants after filtering for Mid

表 4 Mid 筛选后的二阶变异体数量

方法	测试例	筛选后二阶变异体个数	筛选后减少率(%)
已有方法	*t*1,*t*2,*t*3	20	99.86
本文方法	*t*1	17	99.93
	*t*2	19	99.92
	*t*3	15	99.94

Table 5 Number of second-order mutants after filtering for Triangle

表 5 Triangle 筛选后的二阶变异体数量

方法	测试例	筛选后二阶变异体个数	筛选后减少率(%)
已有方法	*t*1,*t*2,*t*3	52	99.89
本文方法	*t*1	19	99.97
	*t*2	25	99.96
	*t*3	23	99.96

注:筛选后二阶变异体:本文方法指由聚类后从每簇中任选一个二阶变异体组成的二阶变异体

如表 3 所示,为两个程序在本文方法和文献[11]方法下生成的二阶变异体(未筛选)个数.由最后一列可知,与已有方法比,本文方法生成了更多的二阶变异体.如就 Triangle 程序而言,已有方法产生的总的二阶变异体的个数为 47 557,而本文方法则生成了 62 369 个二阶变异体,这将有利于充分表示程序中的复杂缺陷.由表 4 和表 5 可知,一方面,二阶变异体聚类后个数极少,与聚类前相比减少了很多;另一方面,与已有方法相比,本文方法执行了更少的二阶变异体,如以 Triangle 为例,已有方法减少了 99.86%,而本文方法的减少率至少为 99.92%.这表明,基于 SOM 神经网络的二阶变异体聚类能够有效减少二阶变异体个数.

为了验证结论是否有效,本文对表 4 和表 5 中本文方法和已有方法筛选后的减少率做了差异显著性分析(使用的是无重复双因素分析,其中 a=0.05),结果见表 6.

Table 6 Result of the significance difference analysis of reduction rate of second-order mutants for benchmark programs

表 6 基准程序中二阶变异体减少率的显著性差异分析的结果

差异源	*F*	*P*-value	*F* crit
Mid 筛选前后减少率组间	147	0.006 743	18.512 82
Triangle 筛选前后减少率组间	484	0.002 06	18.512 82

表 6 从统计学上的显著性差异进行分析,从表格中可以看出,基准程序 Mid 和 Triangle 的 *P*-value 均小于 0.01,说明两组数据具备显著性差异的可能性大于 99%,表明本文方法与已有方法的减少率差异极为显著,具有统计学意义上的差异显著性.说明基于 SOM 神经网络的二阶变异体聚类能够有效减少二阶变异体个数.

(2) 变异充分度的实验

为验证本文方法对变异充分度的影响,首先进行一阶和二阶变异充分度比较的实验,来反映聚类后二阶变异体对变异充分度的影响;然后和文献[11]方法进行变异充分度比较,来反映本文方法的优劣性.实验时使用的一阶变异体和二阶变异体个数的具体信息见表 7,变异测试中变异体被杀死的具体信息见表 8 和表 9,变异充分度的比较信息见表 10,其中,二阶变异测试充分度计算方法为:假设 100 个非等价二阶变异体,聚类后有 20 个簇.若 10 个簇的代表性二阶变异体被杀死,而 10 个簇中共含有 60 个二阶变异体,则变异充分度为 60%.

Table 7 Number of mutants under test cases for benchmark programs

表 7 基准程序的测试例执行的变异体个数

程序	方法	选取的一阶变异体个数	选取的二阶变异体个数
Triangle	已有方法	28	12
	本文方法	28	13
Mid	已有方法	36	10
	本文方法	36	13

Table 8 Information on the number of first-order mutants killed by executing test cases for benchmark programs

表 8 基准程序执行测试用例时被杀死的一阶变异体个数的信息

程序	方法	被杀死的一阶变异体	没有被杀死的一阶变异体	等价一阶变异体	总和
Triangle	已有方法	26	2	0	28
	本文方法	26	2	0	28
Mid	已有方法	32	0	4	36
	本文方法	32	0	4	36

Table 9 Information on the number of second-order mutants killed by executing test cases for benchmark programs

表 9 基准程序执行测试用例时被杀死的二阶变异体个数的信息

程序	方法	被杀死的二阶变异体	没有被杀死的二阶变异体	等价二阶变异体	总和
Triangle	已有方法	11	1	0	12
	本文方法	12	1	0	13
Mid	已有方法	9	0	1	10
	本文方法	12	0	1	13

Table 10 Mutation score of first-order and second-order mutation testing for benchmark programs

表 10 基准程序的一阶和二阶变异测试的变异充分度

程序	方法	一阶变异充分度(%)	二阶变异体变异充分度(%)
Triangle	已有方法	92.9	91.7
	本文方法	92.9	92.5
Mid	已有方法	100.0	100.0
	本文方法	100.0	100.0

表 8 和表 9 展示了变异测试中变异体被杀死的具体信息,根据这两个表可得到表 10 的变异充分度比较信息.其中,本文方法的某个二阶变异体被杀死指该二阶变异体所在簇中的二阶变异体被杀死.以表 9 中三角形基准程序为例,本文所提方法中被杀死的二阶变异体为 12 个,则指的是有 12 个簇的二阶变异体被杀死.

由表 10 可知,

- 一方面,本文所提方法的二阶变异充分度和一阶变异充分度很接近.以三角形为例,本文方法中二阶变异体的变异充分度为 92.5%,较已有文献中的 91.7%,更加的接近一阶变异充分度.根据文献[7,11]所述,当二阶变异充分度和一阶变异充分度越接近时,表明二阶变异体选取的越好,即本文选取聚类后的二阶变异体具有很好的效果.
- 另一方面,也表明了本文所提方法取得的二阶变异充分度较文献[11]更接近一阶变异充分度,表明本文所提方法与已有方法相比,能够运行更有效的少量二阶变异体,取得更好的二阶变异充分度.

为了验证该方法的可应用性和有效性,本文使用一个规模较大的开源项目和小组团队开发程序.程序的信息见表 11.

Table 11 Information list for practical projects

表 11 实用项目的信息列表

项目名	类的个数	代码行数
Snake(github.com/JiaxinTse/Snake)	13	2 160
Shopping(小组开发)	12	1 560

其中,Snake 是一个开源的贪吃蛇游戏,通过键盘上的方向键控制蛇前进的方向,游戏实时记录自己当前的长度及可以计时;Shopping 是一个自己小组使用 SSH 框架开发的电子商城,主要功能有订单管理、用户管理、商品管理.

使用本文提出的组合策略将 Snake 项目和 Shopping 项目的一阶变异体组合生成二阶变异体,其中的具体信息见表 12.

Table 12 Information of mutants generation for practical projects

表 12 实用项目的变异体生成的具体信息

项目名称	核心类	一阶变异体个数	二阶变异体个数
Snake	Obstacle,Foodset,CreatNode,SnakeNode,Move Windows	131 15	10 414
Shopping	ProductAction,UserAction,AdminAction ProductEntity,UserEntity,AdminEntity	91 84	12 126

由表 12 中可知,虽然项目代码行较多,但是由于其中多数为不易出错的代码,如 Snake 中 GUI 代码,该类代码相似的重复性操作较多(由熟练程序员假设可知,此部分属于不易出错范畴),含操作符代码不多,所以一阶变异体并不多,如 Snake 一阶变异体数为 146 个.

因为项目的主要功能分布在某些类中,我们重点对核心类进行分析,实验中,非核心类多数不产生变异体;为验证其可应用性,使用自己团队编写的测试用例进行实验,其中,测试用例并不能保证杀死全部的变异体,选择在全部测试用例下进行分析,而没有再分析单个测试例的变异测试充分度.将聚类前的一阶变异测试充分度和聚类后的二阶变异测试充分度做对比,用以验证本文方法的可应用性.具体信息见表 13.

Table 13 Mutation score for practical projects

表 13 实用项目的变异充分度

项目名	一阶变异体个数	筛选前二阶变异体个数	一阶变异测试充分度(%)	筛选后二阶变异测试充分度(%)
Snake	146	10 414	13.68	13.73
Shopping	175	12 126	14.76	14.69

由表 13 可知,两个项目在使用本文方法约简后,二阶变异体测试充分度较一阶变异测试充分度较为接近,如 Snake 的一阶变异测试充分度为 13.68%,其筛选后二阶变异体测试充分度为 13.73%.由文献[7,11]可知,当二阶变异充分度和一阶变异充分度接近时,表明二阶变异体约简的效果越好.为验证该结果是否具有统计学上的差异显著性,对实验中的一阶变异测试充分度和筛选后二阶变异测试充分度的数据做差异显著性分析,分析结果见表 14.

Table 14 Result of the significant difference analysis of mutation score before and after filtering for practical projects

表 14 筛选前后的实用项目的变异充分度的显著性差异分析的结果

差异源	*F*	*P*-value	*F* crit
组间	0.027 778	0.894 863	161.447 6

从表 14 中可以看出,显著性差异分析结果的 *P*-value 为 0.894 863,大于 0.05,说明两组数据具备显著性差异的可能性小于 95%,表明本文方法的筛选后二阶变异充分度较一阶变异测试充分度较为接近.

(3) 时间开销和发现隐藏二阶变异体的实验

因为发现隐藏二阶变异体的实验需知道全部二阶变异体执行过测试例的信息,所以可从聚类后根据符合该特征的簇中获取隐藏二阶变异体.基于该特征,可将发现隐藏二阶变异体和变异体约简的时间开销同时分析.

为了验证本文方法能够降低时间开销,同时便于发现隐藏二阶变异体实验的比较,所以实验中的程序使用基准程序 Coordinate[7],测试例为非充分测试例(即测试组件不能杀死全部二阶变异体),并将本文方法使用的时间和文献[7]中的时间进行比较.

a) 时间开销的实验

本文通过和文献[7]的方法(记为文献法)比较平均时间来反映时间消耗.实验需执行全部的测试组件,而不是执行单独的测试用例进行测试.Coordinate 程序的平均时间信息见表 15.

Table 15 Average time cost of executing mutants for programs

表 15 程序执行变异体的平均时间消耗

方法	时间消耗				
	二阶变异体生成时间(s)	聚类时间(s)	测试例运行二阶变异体时间(s)	发现隐藏二阶变异体时间(s)	发现隐藏二阶变异体总时间(s)
文献法	无	无	无	39 685.000	39 685.000
本文方法	756.826	38 010.099	0.067	80.900	38 847.892

由表 15 可知,一方面,虽然聚类花费时间较长,但是聚类后的二阶变异体运行测试例的时间却很短;另一方面,本文中所提方法在发现隐藏二阶变异体功能上,比该文献方法中的时间开销减少了一些,如文献法使用了 39 685.000s,而本文方法则使用了 38 847.892s.说明本文方法能够减少程序执行开销且用时较少.

b) 发现隐藏二阶变异体的实验

为了验证本文方法能够发现隐藏二阶变异体,实验中每次使用不同个数的算子运用到程序中,生成不同个数的二阶变异体.如图 7 所示,为该文献方法和本文方法获取到的隐藏二阶变异体的平均数.其中,本文隐藏二阶变异体的获取方法为:根据聚类后的二阶变异体,提取出第一变异体被杀死、第二变异体被杀死,但是二者共同作用时未被杀死的二阶变异体,进行比较,得到隐藏二阶变异体.

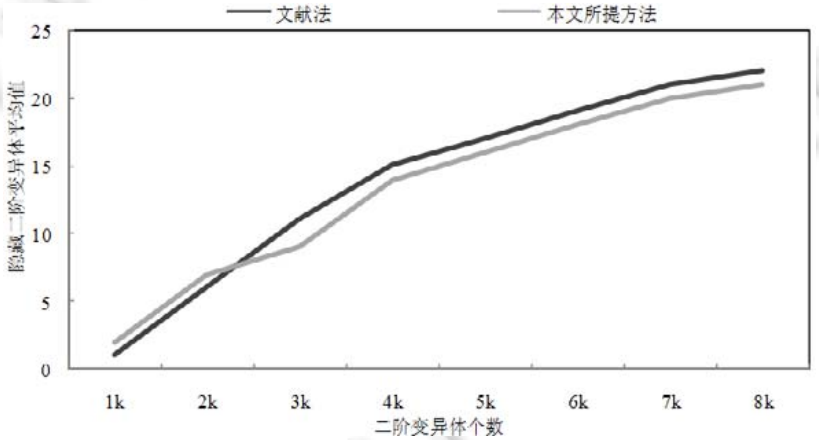

Fig.7 Average of finding subtle second-order mutants

图 7 发现隐藏二阶变异体平均值

由图 7 可知:

- 本文方法寻找的到隐藏二阶变异体个数的平均数,在总数不是很大时优于该文献.
- 当数据量增大时,与该文献方法几乎一样稍微偏低.原因为数量较大时,根据聚类后的二阶变异体,提取出第一变异体被杀死、第二变异体被杀死,但是二者共同作用时未被杀死的二阶变异体数有少许误差.

综合来说,表明本文方法能够发现隐藏二阶变异体,且数量相差不多.

6 结 论

二阶变异测试中的二阶变异体能够模拟程序中的复杂缺陷,具有重要意义.但由一阶变异体组合形成的二阶变异体数量庞大,因此减少执行的二阶变异体个数、降低程序的执行开销是至关重要的.本文提出了一种基于 SOM 神经网络的二阶变异体约简方法.

- 首先,采用较为全面的二阶变异体错误组合策略对一阶变异体组合形成二阶变异体;
- 然后,根据二阶变异体执行过程中的中间值相似性,进行基于 SOM 神经网络模型的变异体聚类;
- 最后,以基准程序和开源项目进行验证.结果表明,组合策略充分模拟了软件中的复杂错误,通过对二阶变异体执行过程中的中间值进行基于 SOM 神经网络的聚类,保证了在二阶变异测试充分度较为接近一阶变异充分度的情况下减少了二阶变异测试的执行开销;与此同时,在较短的时间里能够发现隐藏二阶变异体.

另外,二阶变异测试是高阶变异测试的基础,所以该方法可以推广到高阶变异测试中的其他阶的测试.当推广到不同的阶数时,需要另外增加特征属性以达到更好的聚类效果.如果不增加特征属性,也会有一定的聚类效果,但是效果不如现有的方法好,所以需要额外增加特征属性.

References:

[1] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978,11(4): 34−41. [doi: 10.1109/C-M.1978.218136]

[2] Shan JH, Gao YF, Liu MH, Liu JH, Zhang L, Sun JS. A new approach to automated test data generation in mutation testing. Chinese Journal of Computers, 2008,31(6):1025−1034 (in Chinese with English abstract).

[3] Purushothaman R, Perry DE. Toward understanding the rhetoric of small source code changes. IEEE Trans. on Software Engineering, 2005,31(6):511−526. [doi: 10.1109/TSE.2005.74]

[4] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009,51(10):1379−1393. [doi: 10.1016/j.infsof.2009.04.016]

[5] Langdon WB, Harman M, Jia Y. Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software, 2010,83(12):2416−2430. [doi: 10.1016/j.jss.2010.07.027]

[6] Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. In: Proc. of the 8th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. 2008. 249−258. [doi: 10.1109/SCAM.2008.36]

[7] Omar E, Ghosh S, Whitley D. Subtle higher order mutants. Information and Software Technology, 2017,81:3−18. [doi: 10.1016/j.infsof.2016.01.016]

[8] Ghiduk AS, Girgis MR, Shehata MH. Higher order mutation testing: A systematic literature review. Computer Science Review, 2017,25(6):29−48. [doi: 10.1016/j.cosrev.2017.06.001]

[9] Polo M, Piattini M, GarcaRodrguez I. Decreasing the cost of mutation testing with second order mutants. Software Testing, Verication and Reliability, 2008,19(2):111−131. [doi: 10.1002/stvr.392]

[10] Mateo PR, Usaola MP, Alemn JLF. Validating second-order mutation at system level. IEEE Trans. on Software Engineering, 2013, 39(4):570−587. [doi: 10.1109/TSE.2012.39]

[11] Ghiduk AS. Reducing the number of higher-order mutants with the aid of data flow. E-Informatica Software Engineering Journal, 2016,10(1):31−49. [doi: 10.5277/e-Inf160102]

[12] Nguyen QV, Madeyski L. Problems of mutation testing and higher order mutation testing. In: Proc. of the Advanced Computational Methods for Knowledge Engineering. 2014. 157−172. [doi: 10.1007/978-3-319-06569-4_12]

[13] Nguyen QV, Madeyski L. Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants. Cybernetics and Systems, 2016,47(1-2):48−68. [doi: 10.1080/01969722.2016.1128763]

[14] Ma YS, Kim SW. Mutation testing cost reduction by clustering overlapped mutants. Journal of Systems and Software, 2016,115: 18−30. [doi: 10.1016/j.jss.2016.01.007]

[15] Hussain S. Mutation clustering [MS. Thesis]. Kings College London, 2008.

[16] Kohonen T. The self-organizing map. In: Proc. of the IEEE'90. 1990. 1464−1480. [doi: 10.1016/S0925-2312(98)00030-7]

[17] Delgado S, Higuera C, Calle-Espinosa J, Morn F, Montero F. A som prototype-based cluster analysis methodology. Expert Systems with Applications, 2017,88:14−28. [doi: 10.1016/j.eswa.2017.06.022]

[18] Nguyen QV, Madeyski L. Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation. Journal of Intelligent & Fuzzy Systems, 2017,32(2):1173−1182. [doi: 10.3233/JIFS-169117]

[19] Kurtz B, Ammann P, Delamaro ME, Offutt J, Deng L. Mutant subsumption graphs. In: Proc. of the 2014 IEEE 7th Int'l Conf. on Software Testing, Verication and Validation Workshops. Piscataway: IEEE, 2014. 176−185. [doi: 10.1109/ICSTW.2014.20]

[20] Yao X, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE 2014). New York: ACM Press, 2014. 919−930. [doi: 10.1145/2568225.2568265]

[21] Shin D, Bae DH. A theoretical framework for understanding mutation-based testing methods. In: Proc. of the 2016 IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). Piscataway: IEEE, 2016. 299−308. [doi: 10.1109/ICST.2016.22]

附中文参考文献:

[2] 单锦辉,高友峰,刘明浩,刘江红,张路,孙家骕.一种新的变异测试数据自动生成方法.计算机学报,2008,31(6):1025−1034.

宋利(1993－),女,河南新乡人,硕士生,主要研究领域为软件测试方法与工具,云计算.

刘靖(1981－),男,博士,副教授,CCF 高级会员,主要研究领域为云计算,容错计算,软件测试.