

Identifying Patch Correctness in Test-Based Program Repair

Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, Gang Huang

摘要

近年来,基于测试的自动程序修复已经吸引了很多人的注意。然而实践中的测试套件往往弱,无法保证正确性,现有的方法往往会产生大量不正确的补丁。为了减少产生的错误补丁的数量,我们提出了一种新的方法,启发式地确定产生的补丁的正确性。其核心思想是利用测试案例执行的行为相似性。原始程序和补丁程序上通过的测试可能表现得相似,而原始程序和补丁程序上失败的测试可能表现得不同另外,如果两个测试表现出类似的运行时行为,这两个测试很可能有相同的测试结果。基于这些观察,我们生成新的测试输入以增强测试套件,并使用其行为相似性来确定补丁的正确性我们的方法在一人由 139 个补丁组成的数据集上进行了评估,这些补丁由现有的程序修复系统产生,包括 jGen- Prog、Nopol、jKaliACS 和 HDRepair。我们的方法成功地阻止了 56.3% 的不正确补丁的生成,而没有阻止任何正确的补丁。

关键词: 基于测试的程序修复; 补丁; 正确性

1 引言

任何软件的开发都不可避免的会产生程序 bug, 其产生原因可以追溯到软件开发的各个阶段, 历史数据表明, 超过 45% 的现代软件开发成本消耗与定位和修复 bug 过程中。因此无论是在工业生产还是学术领域, 定位和修复 bug 都是软件工程的核心问题。

如果利用人力来进行 bug 修复, 无疑是费时费力的, 而为了降低修复过程中的时间和人力成本, 自动程序修复方法应运而生。该方法可以根据给定的程序问题, 自动生成程序补丁, 进而修复程序中的错误。在过去几十年里, 人们提出了大量基于测试的程序修复方法, 但很多修复系统并不能很好的保证生成补丁的正确性, 极大的影响了这些系统的可用性。

本次课程的论文复现工作拟通过对以前的程序修复系统产生的补丁的数据集上来评估 PATCH-SIM 方法和 TEST-SIM 方法, 与传统的方法, 如: Anti-patterns, Opad 和 Syntactic and Semantic Distance 相比, 这两个方法提高了自动程序修复方法的精度, 可以为解决 Oracle 问题提供了一个有潜力的方向。

2 相关工作

2.1 基于测试的程序修复

基于测试的程序修复通常被视为一个搜索问题, 通过定义一个补丁的搜索空间, 通常通过一组预定义的修复模板, 目标是在搜索空间中找到正确的补丁。定位补丁的典型方法包括以下几种。

- 搜索算法。一些方法使用元启发式或随机搜索来定位补丁。

- 统计学。一些方法建立了一个统计模型，根据各种信息源如现有的补丁和现有的源代码，选择可能修复缺陷的补丁。
- 约束求解。一些方法将搜索问题转换成可满足性或优化问题，并使用约束解算器来定位补丁。

虽然生成补丁的具体方法不同，但测试套件不全面的问题仍然是对基于测试的程序修复的一个挑战，并可能导致产生不正确的补丁。正如我们的评估所显示的，我们的方法可以有效地增强这些现有的方法，提高它们的精确度。

2.2 补丁分类

面对弱测试套件的挑战，一些研究人员也提出了确定补丁正确性的方法。一些研究人员寻求确定性的方法。Xin 和 Reiss^[1]假设存在一个完美的预测来对测试结果进行分类，并生成新的测试输入以识别违反预测的行为。Yang^[2]等人生成测试输入并监测违法固有预测的情况，包括崩溃和内存安全问题。与他们相比，我们的方法不需要完美的预测，并且有可能识别出不违反固有预测的错误补丁，但有将正确补丁错误分类的风险。

其他方法也使用启发式手段对补丁进行分类。Tan 等人^[3]提出了 Anti-patterns 来捕捉属于特定静态结构的典型的不正确补丁。我们的方法主要依赖于动态信息，这两种方法有可能被结合起来。Yu 等人^[4]研究了通过最小化对生成的测试的行为影响来过滤补丁的方法，并发现这种方法不能提高现有程序修复方法的精度。与他们的方法相比，我们的方法对生成的测试进行分类，并对不同的类别提出不同的行为要求。最后，Weimer^[5]等人强调了识别补丁的正确性的可能方向。

3 本文方法

3.1 本文基础理论

这篇论文最重要的贡献就是提出了两个创新性的观点。第一个是 PATCH-SIM，即补丁的相似性，在应用了正确的补丁之后，通过的测试通常表现的与之前相似，而失败的测试通常表现的不同。因为本来就能通过的测试，打补丁后肯定还是能通过测试；而本来失败的测试，在打了补丁后就应该能通过了。第二个是 TEST-SIM，即测试的相似性，当两个测试有类似的执行情况时，它们很可能有相同的测试结果，比如一个新的测试输入的执行与一个失败测试的执行相似时，那新测试就会有不正确的输出；同理，新测试的执行与一个成功的测试执行相似时，那新测试就会有正确的输出。

3.2 本文方法概述

图 1 显示了我们方法的整体过程。我们将原始的错误程序、一组测试用例和一个补丁作为输入，并产生一个分类结果，告诉人们该补丁是否正确。

我们的方法由五个部分组成，分为三类：测试生成（包括测试输入生成器）、距离测量（包括测试距离测量器和补丁距离测量器）和结果分类（包括测试分类器和补丁分类器）。首先，测试生成器会生成一组测试输入。然后，我们在原始错误程序上运行生成的测试，我们动态地收集有关测试执行的运行时信息。基于运行时信息，测试距离测量器计算每个新生成的测试输入和每个原始测试输入之间的距离距离是一个实数，表示两个测试执行的不同程度。其结果是一个测试距离的向量。然后，这个向量被传递给测试分类器，该分类器根据 TEST-SIM，通过比较其与合格测试的距离和与不合格测试的距

离，将测试分类为合格或失败。

现在有一个增强的测试输入集，它被分为合格或不合格，我们可以用它们来确定补丁的正确性。给定一个补丁，补丁距离测量器在原始程序和补丁程序上运行每个测试，并测量这两个执行之间的距离。其结果是一个补丁距离的向量。最后，这个向量被传入补丁分类器，该分类器基于观察 PATCH-SIM，通过距离来确定补丁的正确性。

在本节的其余部分，我们将分别介绍这三个类别的组件。

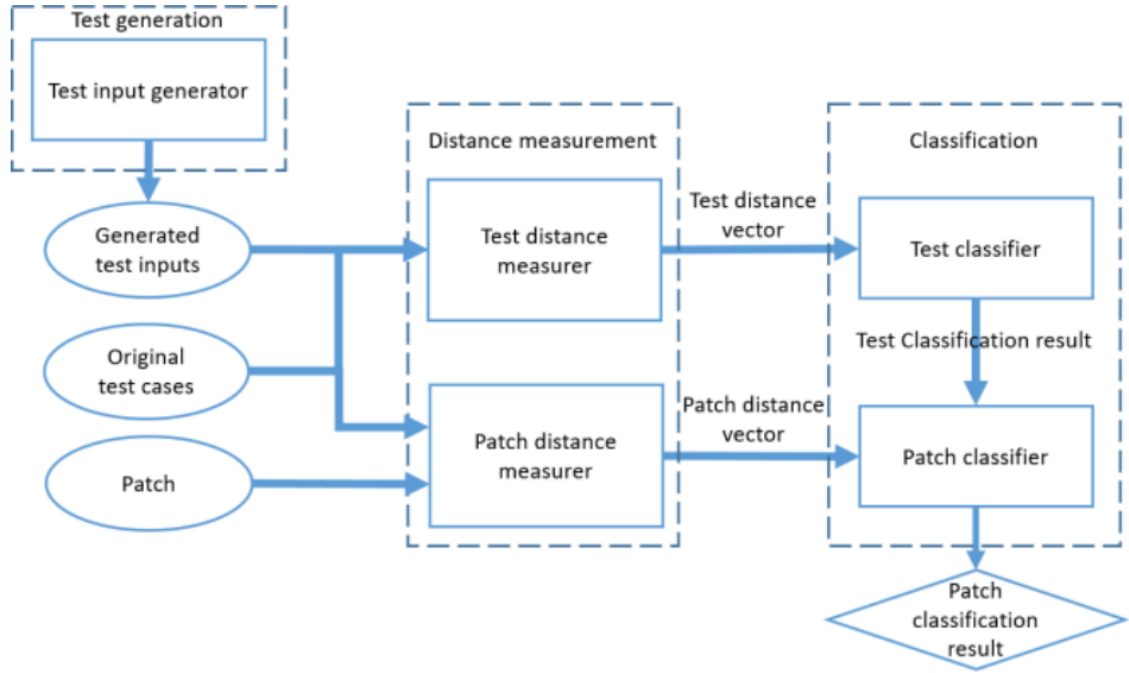


图 1: 方法示意图

3.3 测试生成

给定一个程序，测试生成器为这个程序生成测试输入。此外，由于我们的目标是确定补丁的正确性，我们要求生成的测试涵盖被补丁的方法。如果一个补丁修改了多个方法，生成的测试应该至少涵盖其中一个。

3.4 距离测量

3.4.1 测量距离

我们可以反过来利用 PATCH-SIM 和 TEST-SIM 这两个特性来判断补丁或测试是否正确，而关键在于如何衡量测试执行的相似性，本文通过比较两个程序执行过程中所执行的语句 ID 序列，即程序的完整路径谱，来确定两个执行是否相似，比较两个执行的完整路径谱我们可以简化为测量两个序列之间的距离，而本文使用最长共同子序列（LCS）作为距离度量，两个序列 a 和 b 的 LCS 是仅通过删除元素就能从 a 和 b 得到的最长序列，然后我们用以下公式将 LCS 长度归一化。

$$\text{distance}(a, b) = 1 - \frac{|LCS(a, b)|}{\max(|a|, |b|)} \quad (1)$$

3.4.2 测试距离测量器

测试距离测量器检查每一个生成的测试，并计算它与原始测试的距离。其结果是一个距离向量，其中每个元素代表一个生成的测试和一个原始测试之间的距离。

3.4.3 补丁距离测量器

补丁距离测量器检查每个原始和生成的测试，并计算它在原始程序和补丁程序上执行的距离。其结果是一个距离矢量，其中每个元素代表一个测试的距离。

3.5 结果分类

3.5.1 测试分类器

测试分类器将生成的测试结果分类为合格或不合格。有些生成的测试很难精确分类，我们会丢弃这些测试。让 $\text{result}(t)$ 表示原始测试 t 的测试结果，即，要么通过，要么失败或丢弃。让 $\text{distance}(t, t')$ 表示在原始程序上执行 t 和 t' 的距离。给定一个生成的测试 t' ，我们使用以下公式来确定其分类结果。

$$\text{classification}(t') = \begin{cases} \text{passing} & A_p < A_f \\ \text{failing} & A_p > A_f \\ \text{discarded} & A_p = A_f \end{cases} \quad (2)$$

$$A_p = \min(\{\text{distance}(t, t') \mid \text{classification}(t) = \text{passing}\})$$

$$A_f = \min(\{\text{distance}(t, t') \mid \text{classification}(t) = \text{failing}\})$$

3.5.2 补丁分类器

补丁分类器根据计算出的距离将一个补丁分类为正确或不正确。让 $\text{distance}_p(t)$ 表示在应用补丁 p 之前和之后执行测试的距离。

$$\text{classification}(p) = \begin{cases} \text{incorrect} & A_p \geq K_p \\ \text{incorrect} & A_p \geq A_f \\ \text{correct} & \text{otherwise} \end{cases} \quad (3)$$

$$A_p = \max(\{\text{distance}_p(t) \mid \text{classification}(t) = \text{passing}\})$$

$$A_f = \text{mean}(\{\text{distance}_p(t) \mid \text{classification}(t) = \text{failing}\})$$

我们计算出每个测试在源程序和补丁程序上的距离。观察 PATCH-SIM 的两个条件。第一，通过的测试应该有类似的行为，我们可以将通过测试的最大距离和一个设置好的阈值 K_p 相比来检查这个条件，用最大距离是因为一个不正确的补丁可能只影响几个通过的测试，用最大距离来关注这些测试，如果变化太大则该补丁不正确。第二，失败的测试应该表现得不同，但因为不同 bug 的修复方式不同，很难设定一个固定的阈值。检查失败的测试的平均距离是否大于所有通过的测试的距离，如果不是，则

该补丁不正确，用平均距离是因为打过补丁后所有失败的测试都应该出现改变。其余的补丁我们则分类到正确的。

4 复现细节

4.1 与已有开源代码对比

使用了作者的源码。

4.2 实验环境搭建

Java 1.7
Python3
unidiff (Python3 包)
defects4j 1.3

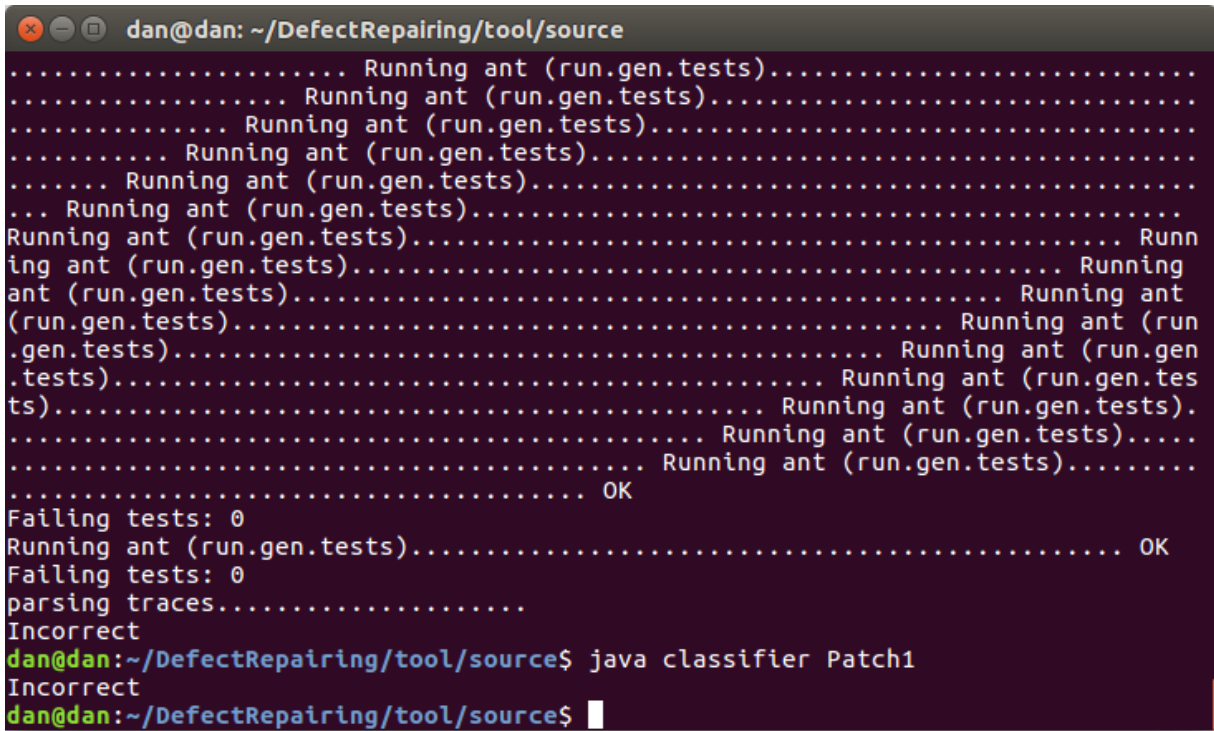
4.3 创新点

在对补丁进行分类时，使用了失败测试的平均距离，但如果某些失败测试的距离值与其他值相比偏大很多或偏小很多，则会对平均值产生较大的影响，考虑改变求平均值的算法来消除这一影响。

改 mean 函数为 trimmean 函数：计算平均值时，先除去数据内部的最大值和最小值，再求平均值。

5 实验结果分析

实验使用的数据集来自以前的程序修复系统产生的 139 个补丁，这些程序包括 jGenProg、Nopol、jKali、HDRepair 和 ACS。图 2 是使用论文方法对补丁进行分类的一个例子。



```
dan@dan: ~/DefectRepairing/tool/source
..... Running ant (run.gen.tests).....
..... Running ant (run.gen.tests).....
..... Running ant (run.gen.tests).....
..... Running ant (run.gen.tests).....
..... Running ant (run.gen.tests).....
... Running ant (run.gen.tests).....
Running ant (run.gen.tests)..... Running
ant (run.gen.tests)..... Running
ant (run.gen.tests)..... Running ant (run
.gen.tests)..... Running ant (run.gen
.tests)..... Running ant (run.gen.tes
ts)..... Running ant (run.gen.tests).
..... Running ant (run.gen.tests)....
..... Running ant (run.gen.tests).....
..... OK
Failing tests: 0
Running ant (run.gen.tests)..... OK
Failing tests: 0
parsing traces.....
Incorrect
dan@dan:~/DefectRepairing/tool/source$ java classifier Patch1
Incorrect
dan@dan:~/DefectRepairing/tool/source$
```

图 2: 复现实验结果示例

表 1 和表 2 分别显示了我们的方法在每个程序和每个项目的数据集上的表现。如表所示，我们的方法成功地过滤掉了 110 个错误的补丁中的 62 个，而没有过滤掉任何正确的补丁。此外，我们的方法在不同的工具和不同的项目上显示出类似的性能，表明我们的结果有可能被推广到不同类型的项目

和不同类型的程序上。

表 1: 每个程序的补丁的总体效果

Tool	Incorrect	Correct	Incorrect Excluded	Correct Excluded
jGenprog	17	5	8(47.1%)	0
jKali	17	1	9(52.9%)	0
Nopol2015	24	5	16(66.7%)	0
Nopol2017	40	0	22(55.0%)	0
ACS	6	15	3(50.0%)	0
HDRepair	6	3	4(66.7%)	0
Total	110	29	62(56.3%)	0

表 2: 每个项目的补丁的总体效果

Project	Incorrect	Correct	Incorrect Excluded	Correct Excluded
Chart	23	3	14(60.9%)	0
Lang	11	4	6(54.5%)	0
Math	63	20	33(52.4%)	0
Time	13	2	9(69.2%)	0
Total	110	29	62(56.3%)	0

图 3 是对同一补丁使用论文方法和改进方法进行分类的对比。

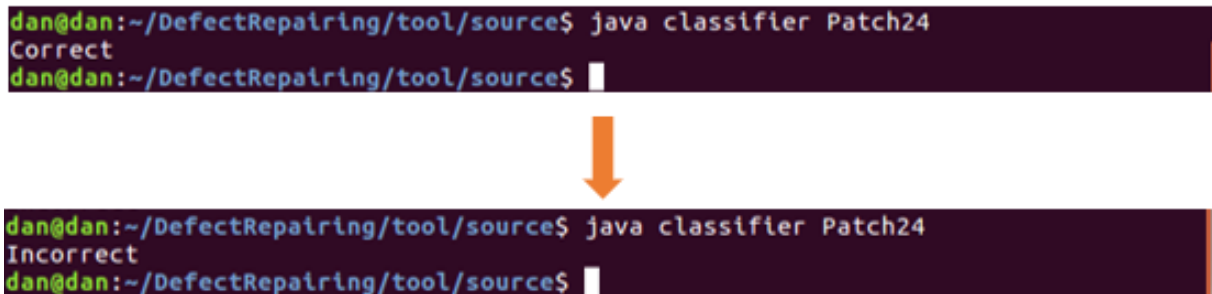


图 3: 改进实验结果示例

如表 3 所示，在使用改进后的方法对同一数据集补丁进行分类后，改进后的方法相比原方法多过滤掉了 4 个错误的补丁，同时也没有过滤掉任何正确的补丁。

表 3: 改进方法的总体效果

	Incorrect	Correct	Incorrect Excluded	Correct Excluded
Total	110	29	66(60.0%)	0

6 总结与展望

本部分对整个文档的内容进行归纳并分析目前实现过程中的不足以及未来可进一步进行研究的
方向。根据论文提出的方法，对给定数据集的补丁进行分类，成功复现文章的结果。但在复现过程中，

因个人水平所限，提出的改进方法不甚有效。在未来可以考虑更换不同的指标来测量序列的距离，如编辑距离或汉明距离，以期更好的实验效果。

参考文献

- [1] XIN Q, REISS S P. Identifying Test-Suite-Overfitted Patches through Test Case Generation[J/OL]. ISSTA 2017 2017: 226-236. <https://doi.org/10.1145/3092703.3092718>. DOI: 10.1145/3092703.3092718.
- [2] YANG J, ZHIKHARTSEV A, LIU Y, et al. Better Test Cases for Better Automated Program Repair [J/OL]. ESEC/FSE 2017 2017: 831-841. <https://doi.org/10.1145/3106237.3106274>. DOI: 10.1145/3106237.3106274.
- [3] TAN S H, YOSHIDA H, PRASAD M R, et al. Anti-Patterns in Search-Based Program Repair[J/OL]. FSE 2016 2016: 727-738. <https://doi.org/10.1145/2950290.2950295>. DOI: 10.1145/2950290.2950295.
- [4] YU Z, MARTINEZ M, DANGLOT B, et al. Test case generation for program repair: A study of feasibility and effectiveness[J]. arXiv preprint arXiv:1703.00198, 2017.
- [5] WEIMER W, FORREST S, KIM M, et al. Trusted software repair for system resiliency[J]., 2016: 238-241.