软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(2):377−396 [doi: 10.13328/j.cnki.jos.006347]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

自适应推导下的统一化调试加速技术*

娄一翎[1,2], 张令明[3], 郝 丹[1,2], 张皓天[4], 张 路[1,2]

[1](高可信软件技术教育部重点实验室(北京大学), 北京 100871)

[2](北京大学 信息科学技术学院, 北京 100871)

[3](Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois 61822, USA)

[4](蚂蚁金融服务集团, 浙江 杭州 310099)

通信作者: 郝丹, E-mail: haodan@pku.edu.cn

摘 要: 在传统调试过程中, 缺陷定位通常作为程序修复的前置步骤. 最近, 一种新型调试框架(统一化调试)被提出. 不同于传统调试中缺陷定位和程序修复的单向连接方式, 统一化调试首次建立了定位与修复之间的双向连接机制, 从而达到同时提升两个领域的效果. 作为首个统一化调试技术, ProFL 利用程序修复过程中伴随产生的大量补丁执行信息逆向地提升已有缺陷定位技术的效果. 统一化调试技术不仅修复了可被修复的缺陷, 而且也为不能被自动修复技术修复的缺陷提供了有效的调试线索. 虽然统一化调试是一个很有前景的研究方向, 但其在补丁验证过程中涉及到了大量的测试用例执行(比如百万量级的测试执行), 因此时间开销问题严重. 提出一种针对于统一化调试框架的加速技术(AUDE), 该技术通过减少对缺陷定位效果无提升的测试执行, 以提升统一化调试的效率. 具体来说, AUDE 首先通过马尔可夫链蒙特卡洛采样方法构建补丁执行的初始序列, 随后在补丁执行过程中将已执行的补丁信息作为反馈信息, 自适应性地估计每一个未执行补丁可能提供有效反馈信息的概率. 在广泛使用的数据集 Defects4J 上对该技术进行了验证, 发现 AUDE 在显著加速 ProFL 的同时, 并没有降低其在缺陷定位和程序修复的效果. 例如: 在减少了 ProFL 中 70.29%的测试执行的同时, AUDE 仍在 Top-1/Top-3/Top-5 指标上与 ProFL 保持了相同的定位效果.

关键词: 软件质量保障; 软件测试; 软件调试; 缺陷定位; 缺陷修复

中图法分类号: TP311

中文引用格式: 娄一翎, 张令明, 郝丹, 张皓天, 张路. 自适应推导下的统一化调试加速技术. 软件学报, 2022, 33(2): 377–396. http://www.jos.org.cn/1000-9825/6347.htm

英文引用格式: Lou YL, Zhang LM, Hao D, Zhang HT, Zhang L. Accelerating Unified Debugging via Adaptive Inference. Ruan Jian Xue Bao/Journal of Software, 2022, 33(2): 377–396 (in Chinese). http://www.jos.org.cn/1000-9825/6347.htm

Accelerating Unified Debugging via Adaptive Inference

LOU Yi-Ling[1,2], ZHANG Ling-Ming[3], HAO Dan[1,2], ZHANG Hao-Tian[4], ZHANG Lu[1,2]

[1](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

[2](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[3](Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois 61822, USA)

[4](Ant Financial Services Group, Hangzhou 310099, China)

Abstract: Fault localization and program repair techniques have been extensively studied for over decades; their only connection is that fault localization serves as a supplier diagnosing potential buggy code for automated program repair. Recently, unified debugging has been proposed to unify fault localization and program repair in the other direction for the first time to boost both areas. ProFL, the first work on unified debugging, opens a new dimension that the large volume of patch execution information during program repair in turn

* 基金项目: 国家重点研发计划(2017YFB1001803); 国家自然科学基金(61872008)

收稿时间: 2020-10-14; 修改时间: 2020-11-27, 2021-01-14; 采用时间: 2021-04-12; jos 在线出版时间: 2021-05-20

can help boost state-of-the-art fault localization. Note that even state-of-the-art program repair techniques can only fix a small ratio of real bugs (e.g., <20% for Defects4J) and simply abort for the vast majority of unfixed bugs. In contrast, unified debugging not only directly fixes bugs when possible, but also provides debugging hints for bugs that cannot be automatically fixed (e.g., the patch execution information from such unsuccessful repair can still help boost fault localization for manual repair). Although demonstrated to be a promising direction, unified debugging relies on massive test executions (i.e., million test executions) and can cost hours for execution. This work proposes AUDE to accelerate unified debugging by reducing test executions that provide little helpful feedback for improving fault localization. Specifically, AUDE first constructs an initial execution order of patches guided by Markov chain Monte Carlo sampling strategy, and then adaptively estimates the likelihood of each patch being informative during patch execution on-the-fly. The results on the widely-used Defejcts4J benchmark show that AUDE significantly accelerates ProFL by reducing 70.29% of test executions with negligible effectiveness drop in both fault localization and program repair, e.g., AUDE can localize the same number of bug methods at Top-1/Top-3/Top-5 as ProFL.

Key words: software quality assurance; software testing; software debugging; fault localization; program repair

软件缺陷(software bug)普遍存在于软件系统中，而软件缺陷调试(debugging)往往代价巨大且具有挑战性. 因此，为了降低软件缺陷调试过程中的人工代价，软件缺陷自动调试领域受到了非常广泛的关注. 其中，最主要的两个核心问题就是缺陷定位和程序修复. 缺陷定位技术，通过静态或者动态的程序分析技术来识别程序中可能出错的代码元素. 比较常见的缺陷定位技术包括基于频谱[1−10]、基于变异[11−13]和基于程序切片[10]等. 给定一个含有缺陷的程序，缺陷定位技术通常会产生一个依照出错可能性(通常被称为可疑度)降序排序的程序元素列表. 随后，开发者可通过从头遍历该列表，依次检查程序元素，从而加速人工修复的过程. 程序自动修复技术[14−26]旨在没有人工干预的情况下，自动化地修复程序中的缺陷. 通常来说，程序自动修复技术往往会先应用已有的缺陷定位技术来识别可能出错的代码元素，然后在此基础上进行后续的补丁生成. 例如，很多近期的程序自动修复技术[27−29]的第 1 步即是应用基于频谱的缺陷定位技术 Ochiai[30]来识别出错的代码元素. 缺陷定位和程序修复这两个领域自提出以来已被研究了很多年，在传统的调试场景下，缺陷定位和程序修复通常以这种单向的方式进行连接. 但是，目前无论是缺陷定位还是程序修复技术在实际中都仍未得到广泛的应用[31,32]：已有的缺陷定位技术在实际中的效果表现仍然非常有限[33,34]，而已有的程序自动修复技术只能修复很小的一部分(在数据集上 Defects4J[35]不到 20%)真实缺陷[27−29,36,37]或者某一类特定的程序缺陷[34].

最近，一种双向连接缺陷定位和程序修复的统一化调试方法[31,32]被提了出来. 作为首个统一化调试方法，ProFL[31,32]利用了大量修复过程中产生的补丁执行信息来逆向提升已有缺陷定位技术的效果. 该技术不仅将程序自动修复的适用范围拓展到了所有缺陷(不仅仅是原来小部分可以被自动修复技术成功解决的缺陷)，而且也大幅度提升了已有缺陷定位技术的效果. 例如，ProFL 通过利用修复工具 PraPR[27]在修复过程中产生的补丁信息，不仅比主流的基于频谱和基于变异的缺陷定位技术至少多将 37.6%的缺陷代码元素排到可疑度列表第一位，而且在效果上也超越主流的基于学习的非监督和监督缺陷定位技术[38,39].

虽然颇具前景，但统一化调试方法中仍存在着不容忽视的效率问题. ProFL 工作提到了其中存在的效率问题，并且提出了只使用部分补丁执行矩阵信息来辅助提升缺陷定位的建议. 例如：优先执行之前失败的测试用例然后再执行之前通过的测试用例，并且一旦有任意测试用例在当前补丁上执行失败就终止当前补丁的执行. 尽管应用了上述效率优化策略，但对于 Defects4J 中的大型项目(例如 Closure)，ProFL 仍然要花费很多个小时来收集必须的补丁执行信息[31,32,40]. 这个过程往往涉及到了百万级别的测试用例执行(值得注意的是：对于每一个补丁执行，往往存在多个测试执行).

考虑到统一化调试所依赖的大量测试执行已成为其实际应用中的一大难题，在本文中，我们提出了一种统一化调试加速技术，旨在更好地平衡统一化调试的效率和效果. 我们的基本思想是：在海量需要被执行的补丁中，存在着很多对提升缺陷定位没有帮助因此也没有必要被执行的补丁. 在本文中，我们将这一类补丁称为无信息性补丁. 相应地，对于那些可以提供改善定位效果的反馈信息的补丁，我们则称其为信息性补丁. 基于这一思想，提出了 AUDE (accelerating unified debugging)，一种基于补丁是否具有信息性以优化补丁执行的统一化调试加速技术. AUDE 主要分成两个部分，包括初始优化和补丁执行过程中的自适应性推导：在初始

优化阶段, 对于所有生成的补丁, AUDE 基于它们的可疑度和马尔可夫链蒙特卡洛采样方法[41,42]来决定所有补丁的初始执行顺序; 在自适应推导阶段, 根据补丁初始的执行顺序, AUDE 迭代地选择一个补丁并根据其具有信息性的可能性来决定是否执行它, 而该可能性是根据目前为止所有已经执行的补丁信息进行概率推导得出的. 自适应推导过程不断迭代重复, 直到似真补丁出现(plausible patch, 即通过所有测试的补丁)或没有剩余的待验证补丁.

我们在广泛使用的数据集 Defects4J 中所有 395 个真实缺陷上对 AUDE 进行了实验验证. 实验结果显示, AUDE 减少了 ProFL 中 70.29%的测试执行, 显著地加速了 ProFL; 并且在加速的同时, AUDE 没有在缺陷定位和自动修复中引入明显的效果损失. 例如: 与 ProFL 相比, 对于相同数目的缺陷, AUDE 仍将相同数目的缺陷代码排到了可疑度列表的第 1 位/第 3 位/第 5 位(Top-1/Top-3/Top-5). 值得注意的是, 相关工作[31,32]已经验证了 ProFL 的效果胜过已有的主流缺陷定位技术(甚至包括了最新的基于深度学习的缺陷定位技术 DeepFL[38]). 同时, 在被跳过的 70.29%的测试执行后, 只有很小的一部分(5.34%)似真修复被 AUDE 地丢失了.

总的来说, 本文做出了下述贡献:

- 方法: 第 1 个针对于统一化调试的加速方法(AUDE), 自适应地根据补丁具有信息性的概率来调整补丁执行的顺序和减少测试用例的执行;
- 实现: 一个建立在最新统一化调试框架 ProFL 上的方法实现.

1 背景介绍

1.1 传统调试

在传统调试场景下, 缺陷定位和程序修复通常是单向连接的: 缺陷定位为程序修复提供了后续生成补丁的出错代码位置.

缺陷定位[1−10]在近年来被广泛地加以研究, 通常用来为程序修复识别潜在的出错代码位置. 基于频谱的缺陷定位技术, 作为一种使用最为广泛的缺陷定位技术之一, 利用原来失败和通过测试用例在出错程序上的覆盖信息来计算程序元素(例如程序语句或函数)的可疑度(如上文所述, 可疑度衡量代码元素出错的可能性). 基于频谱的缺陷定位技术的核心思想是, 被更多失败测试用例覆盖的程序元素比被更多通过测试用例覆盖的程序元素更加可疑. 至今为止, 有大量的基于频谱的缺陷定位技术, 包括 Tarantula[43]、Ochiai[44]和 Ample[3]等等. 然而, 覆盖信息并不能完全准确地反映程序元素对于执行结果的影响. 为了弥补覆盖信息和影响信息之间的差距, 基于变异的缺陷定位技术[5,11−13]被提了出来, 这项技术基于变异测试的思想对程序代码进行变换, 从而模拟变换对执行结果的影响. 例如 Metallaxis[13], 作为一项主流的基于变异的缺陷定位技术, 在计算可疑度值时就是基于变异前后所有类型的失败信息受到的影响. 除了基于频谱和基于变异的缺陷定位技术以外, 大量其他信息也被用在缺陷定位技术当中, 如程序中的不变量[45]、缺陷报告[46]、谓词翻转[47]、代码复杂度[48]以及程序切片[10]等等. 除此之外, 研究人员还提出基于学习的定位技术[38,49,50], 同时整合利用上述多维度的信息来源.

程序自动修复[14−17,51]旨在不需要人力参与的条件下, 自动化地修复软件中的缺陷. 程序自动修复领域自提出以来, 一直都被密切地关注和讨论. 主流的程序自动修复技术通常先借助现成的缺陷定位技术来识别可能出错的程序元素列表, 随后通过依次修改可疑度列表中的程序元素来为其生成补丁, 同时, 每个生成的补丁都会通过执行测试用例而被验证.

至今为止, 研究人员提出了各种不同的策略, 通过尝试性地修改程序代码找到出错程序的似真补丁(即通过所有测试用例的补丁). 基于语义的修复技术[21,23]使用语义分析(例如符号执行)来综合生成程序条件语句或者更为复杂的代码片段, 使其可以通过所有的测试用例. 基于搜索的自动修复技术[23,28,29,52]往往基于某种生成补丁的规则, 在所有候选补丁中搜索似真补丁. 后者由于其在大型实际软件系统中的可拓展性, 受到了研究人员更多的关注. 例如, 最新字节码自动修复技术 PraPR[27]成功地为 Defects4J 数据集中的 43 个缺陷成功生成了正确补丁.

尽管调试技术在过去几十年里被广泛地研究, 但到目前为止, 已有的缺陷定位技术在实际中仍然表现出比较有限的效果; 同时, 已有的自动修复技术也只能修复很小一部分的真实缺陷或者某种特定类型的缺陷[53]. 例如, 一些主流的自动修复技术[27,28]只能修复 Defects4J 中不到 20%的真实缺陷.

1.2 统一化调试

最近, 双向连接缺陷定位和程序修复的统一化调试[31,32]技术被提了出来. 其中, Lou 等人[31,32]提出了首个统一化调试技术 ProFL. 该技术逆向连接缺陷定位和程序修复, 通过分析程序修复过程中的补丁信息来帮助提升已有的缺陷定位结果. 其实验结果表明, 程序修复可以显著地提升缺陷定位的效果. 基于最新的自动修复工具 PraPR, ProFL 不仅可以比主流的基于频谱和基于变异的缺陷定位技术至少多将 37.6%的缺陷代码元素排到可疑度列表第一位, 而且在效果上也超越主流的基于学习的非监督和监督缺陷定位技术. 除了 ProFL 以外, Xu 等人也提出了基于类似思想的定位改善技术 Restore[54]. 与 ProFL 不同, Restore 依赖于开发人员提供出错的函数作为输入, 进而利用补丁执行信息来改善语句级别的缺陷定位效果.

这种在缺陷定位和程序修复之间所建立的双向连接机制, 充分利用了那些不能被自动修复技术所修复的缺陷(在传统调试中, 这一情况被视为失败的修复继而直接退出), 将自动修复的应用范围拓展到所有可能的缺陷, 并且也提升了缺陷定位的效果. 虽然这是一个有前景的方向, 统一化调试由于涉及到大量的测试用例执行, 其中仍然存在着不容忽视的效率问题. 例如: 尽管 ProFL 已经采用了一定的效率优化策略(优先执行原来失败的测试用例然后执行原来成功的测试用例, 终止当前补丁的验证当有任意测试用例在当前补丁上执行失败时), 它仍然需要花费很多时间来收集百万测试用例执行信息来达到改善缺陷定位的目的.

在本文的工作中, 我们提出了第一个加速统一化调试技术的加速方法 AUDE. 具体来说, AUDE 在不降低定位和修复效果的同时, 大幅地减少所需执行的测试用例数目, 从而显著加速当前统一化调试技术 ProFL. 值得注意的是: Mike 等人[55]和 Li 等人[49]曾提出一些简单的针对基于变异的缺陷定位加速技术, 其中, Mike 等人随机地执行一部分变异体, 而 Li 等人则只执行在被原来失败测试用例覆盖的程序元素上生成的变异体. Li 等人的策略在相关工作实验中被证明优于 Mike 等人的优化策略, 而前者已经默认在 AUDE 中被使用. 并且, 之前针对基于变异的缺陷定位加速方法对于每个变异体会执行所有测试用例, 而 AUDE 则不会执行所有的测试用例. 换言之, 当任意测试用例在当前补丁上执行失败时, AUDE 会终止当前补丁的验证.

2 示 例

在本节中, 我们用一个真实缺陷(Closure-32)作为例子来阐述现有统一化调试方法 ProFL 中存在的效率问题, 并且解释我们方法背后的动机. Clossure-32 是软件 Google Closure Compiler 在被广泛使用的数据集 Defects4J[20]上的第 32 个错误版本. 表 1 呈现了该缺陷的具体细节. 由于空间限制, 我们只展现一部分涉及到的关键函数和测试用例来帮助解释.

- ProFL 是如何工作的.

给定一个出错程序及其失败测试用例集合, ProFL 首先使用现成的缺陷定位技术(例如基于频谱的缺陷定位技术)来为程序元素(例如函数)生成基于可疑度降序排列的初始排序列表; 其次, ProFL 采用最新的程序自动修复技术(即 PraPR)来生成和验证补丁; 最后, ProFL 将返回似真补丁(若存在的话)和通过补丁执行信息精化过的缺陷定位列表. 首先, ProFL 使用 Ochiai 公式[30]计算每个语句的可疑度, 随后, 对于每个函数采用其包含的所有语句的可疑度最大值作为其可疑度(即聚集策略[48]). 在表 1 中, 列“SBFL”代表了基于频谱缺陷定位计算出的初始可疑度, 列“MID”代表了每个函数的唯一标识符(这些函数按照它们的初始可疑度进行降序排列). 由于基于频谱的缺陷定位技术本身的局限性, 真正有错的函数 extractMultilineTextualBlock 一开始便被排在可疑度列表的第 5 名(即 Me_5). 其次, ProFL 使用最新的程序自动修复技术 PraPR 对 Closure-32 进行缺陷修复. 然而, 由于修复这个缺陷需要涉及到多行代码的复杂修改, 因此没有正确或者似真补丁被生成. 这样的失败修复案例, 在传统的缺陷调试场景下往往直接被报告为失败修复并结束整个调试过程. 但在统一化调试中, 整个修复过程中的所有补丁执行信息会被收集并用来逆向改进缺陷定位的结果. 列“PID”代表了每一个生成补丁的

唯一标识符, 而列“原来失败的测试用例”和列“原来通过的测试用例”则分别代表了原来失败的测试和原来通过的测试在该补丁上的执行结果, 其中, 空白代表了该测试用例并未覆盖产生补丁的代码位置, 因此出于效率考虑而不在该补丁上被执行. 举例来说, $P1$ 被 4 个原来失败的测试用例(即 ft_1 到 ft_4)和两个原来通过的测试用例(即 pt_1 到 pt_2)所覆盖, 并且测试用例 ft_1 到 ft_4 都在 $P1$ 执行通过. 最后, ProFL 利用测试执行结果对函数进行分类, 从而改进原有的定位效果. 列“补丁分类”则展现了相应的分类结果. 一个补丁可以是:

(1) CleanFix(正向补丁): 当至少有一个原来失败的测试用例在该补丁上执行通过且没有任何原来通过的测试用例在该补丁上执行失败时(例如 $P6_1$);

(2) NoisyFix(交错补丁): 当至少有一个原来失败的测试用例在该补丁上执行通过且存在原来通过的测试用例在该补丁上执行失败时(例如 $P6_2$);

(3) NonFix(空白补丁): 不存在任何测试用例在该补丁上的执行结果发生改变(例如 $P2$);

(4) NegFix(负向补丁): 当不存在任何原来失败的测试用例在该补丁上执行通过但至少有一个原来通过的测试用例在该补丁上执行失败时(例如 $P5$).

根据预先定义的不同类别补丁之间的偏序关系(CleanFix 正向补丁>NoisyFix 交错补丁>NonFix 空白补丁>NegFix 负向补丁), 每个函数根据在其产生所有补丁中的优先级最高的补丁类型作为其类型. 在这个基础上, 函数之间按照它们所属的补丁类型进行二次排序, 而属于同一个类别的函数则根据其初始可疑度(即基于频谱定位技术计算得出的可疑度)进行排序. 例如: 通过重新排序, 出错的函数 Me_5 从原来的可疑度第 5 名排到了第 1 名, 因为它是一个属于 CleanFix 正向补丁的函数.

- 效率问题.

ProFL 将程序自动修复的可应用范围拓展为所有缺陷(不仅仅是那些可以被自动修复技术解决的缺陷), 同时也改进了缺陷定位的效果. 尽管非常有效, 但 ProFL 由于涉及到了大量的测试用例执行, 其中存在着不容忽视的效率问题. 在 Closure-32 中, PraPR 总共产生了 152 867 个待执行的补丁. 基于效率上的考虑, 对于每一个补丁, ProFL 优先执行原来失败的测试用例然后再执行原来通过的测试用例, 并且终止当前补丁的执行当有任意测试用例在当前补丁上执行失败时. 上述两点都是程序修复在实际应用时常用的提高效率的策略. 尽管如此, ProFL 仍然进行了 159 351 次测试执行, 而其中很大一部分并没有提供对改进缺陷定位有帮助的信息. 例如: 在 $P1$ 被执行之后, $P2$ 的执行结果对于 Me_1 的最终可疑度排序并不会产生影响, $P2$ 所属的补丁类型(NonFix 空白补丁)优先级不如 $P1$(NoisyFix 交错补丁)的高. 又例如: $P6_2$、$P6_3$ 和 $P6_4$(即交错补丁)也都返回了冗余的反馈信息, 在改进 Me_5 的最终定位排名上效果重叠.

表 1 Closure-32 示例

MID	可疑方法	SBFL	补丁		原来失败的测试用例				原来通过的测试用例				
			PID	补丁分类	ft_1	ft_2	ft_3	ft_4	pt_1	pt_2	pt_3	pt_4	pt_5
Me_1	2530: private boolean *parse*(·){ . . .	函数可疑度=0.84 (函数补丁分类=NoisyFix)											
	2536: if (*preserve*. *length*(·)>0) { . . .	0.84	$P1$	NoisyFix	✓	✓	✓	✓	✗	✗			
	2547: if (*fileLevelJsDocBuilder*!=*null*) . . .	0.60	$P2$	NonFix	✗	✗			✓	✓			
Me_2	1206: public void *setLicense*(·) { . . .	函数可疑度=0.79 (函数补丁分类=NonFix)											
	1207: *lazyInitInfo*(·);	0.79	$P3$	NonFix	✗	✗		✗	✓	✓			
…													
Me_5	1329: private void *extractMultilineTextualBlock*(·){ . . .	函数可疑度=0.57 (函数补丁分类=CleanFix)											
	1370: if (*option*!=*WhitespaceOption*. *SINGL_LINE*) {	0.41	$P4_1$ $P4_2$	NonFix NonFix	✗ ✗	✗ ✗			✓ ✓	✓ ✗			
	1371: *builder.append*(“*n*”); . . .	0.20	$P5$	NegFix		✗				✗			
	1408: if (*builder.length*(·)>0){ . . .	0.55	$P6_1$ $P6_2$ $P6_3$ $P6_4$	CleanFix NoisyFix NoisyFix NoisyFix	✓ ✓ ✓ ✓	✗ ✗ ✗ ✗	✗ ✗ ✗ ✗	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	✓ ✗ ✗ ✗	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓
	1409: *builder.append*(“.”); ←出错代码	0.57	$P7$	CleanFix	✓	✓	✗	✗	✓	✓	✓	✓	

加速 ProFL 的关键在于只执行能为改进缺陷定位提供一定帮助信息的补丁(在本文中称为信息性补丁)而跳过执行不能提供有效信息的补丁(相应地, 无信息性补丁). 总的来说, 随着越来越多的补丁在程序自动修复过程中被执行, 不断增长的补丁执行信息可以为后续未执行的补丁是否具有信息性提供推断的依据和线索. 因此在本文中, 我们将自适应地分析不断积累的补丁执行信息, 从而在未执行补丁中识别出信息性和无信息性补丁.

3 方法描述

我们提出了针对统一化调试的加速技术 AUDE, 通过动态地分析已有的补丁执行信息来判断未执行补丁是否具有信息性, 从而优化补丁的执行. 图 1 表示了 AUDE 加速统一化调试的整个过程. AUDE 内包含了两部分内容: 初始优化和自适应性推导. 前者在所有补丁执行前构建了初始的补丁执行序列; 在补丁执行时, 后者基于已执行补丁信息进行概率推导, 而自适应地执行或者跳过序列中的补丁.

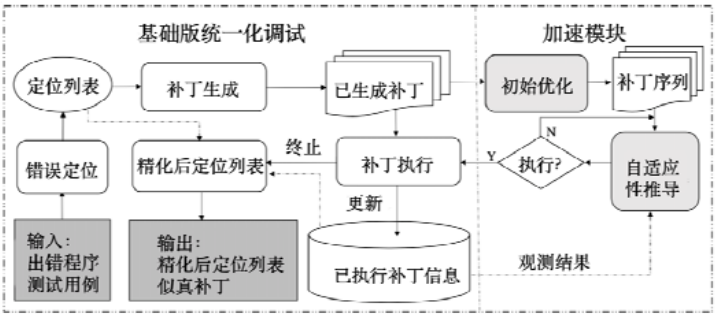

图 1 统一化调试加速技术方法流程

在初始优化阶段, AUDE 基于马尔可夫链蒙特卡洛采样方法(MCMC)[41,56]迭代地从未被选择的补丁集合中选出补丁, 追加到补丁执行序列尾部. 通过这种方式, AUDE 产生了一个满足以下两个条件的初始补丁执行序列: (1) 发生在高可疑度代码片段的补丁更可能被优先执行; (2) 发生在低可疑度代码片段的补丁仍然有被执行的低可能性. 关于这一部分的更多细节可参见第 3.1 节.

在自适应推导阶段, AUDE 以迭代的方式依次基于补丁初始执行序列中每个补丁有信息性的可能性来判断它们是否要被执行. 信息性补丁会被执行, 并且它的执行结果会被收集. 被判断为无信息性的补丁会从执行序列中被移除. AUDE 不断重复这个过程, 直到出现似真补丁或执行序列中已经没有待执行的补丁. 特别地, 我们将上述判断补丁是否具有信息性的过程转化为一个概率推导问题. 对于每个补丁, 其是否具有信息性的可能性作为一个随机变量. 对于一个未执行的补丁 P, 所有当前已被执行的补丁信息可被作为观测现象, 而补丁 P 与已执行补丁之间可以建立条件概率. 推导出来的后验概率即表征了在当前补丁执行结果的基础上, P 具有信息性的可能性. AUDE 倾向于执行后验概率高的补丁, 根据新的执行结果不断更新观测现象和先验概率. 关于这一部分的更多细节可参见第 3.2 节.

3.1 初始优化

在自适应推导阶段之前, AUDE 在该阶段构建了初始补丁执行序列. 在许多主流的程序自动修复技术中, 补丁的执行顺序是完全按照它们改变的代码片段可疑度降序排列的. 这样的执行顺序并不能很好地适用于我们的场景: (1) 修改相近代码的补丁会被频繁地执行, 从而降低了即时反馈信息的多样性和推导的稳定性; (2) 修改可疑度高的代码的补丁总是比修改可疑度低的代码的补丁优先执行, 从而降低了初始可疑度低的出错函数被重新排列的可能性(尤其是考虑到实际修复过程往往在似真补丁被发现后就被整体终止). 因此, 为了保证在修复过程中补丁执行信息以一种平衡和有信息增益的方式逐渐积累, 修改可疑度高的代码的补丁应当有较大的但并非决定性大的概率被优先执行. 而该问题可以进一步抽象为给定概率分布下的采样问题.

为了解决这一问题, 在AUDE中, 我们采用了马尔可夫链蒙特卡洛采样方法(MCMC)[41,42], 通过构建马尔可夫链趋近期望的概率分布, 从而迭代式地构建序列. 更确切来说, AUDE 首先根据补丁的初始可疑度将其分入不同的优先队列; 随后采用 MCMC 采样方法迭代式地决定一个优先队列, 并从中随机选取一个补丁加入补丁执行序列末端. AUDE 不断重复第 2 个步骤, 直到完成包含所有补丁的执行序列构建.

- 优先队列.

所有生成的补丁会根据其生成处代码的初始可疑度(即应用 Ochiai 公式的基于频谱的缺陷定位技术)被分入不同的优先队列中. 具有同样初始可疑度的补丁会被分入同一个优先队列中, 而所有优先队列会按照其所在的可疑度进行降序排列. 为了更好地解释方法, S 代表了所有优先队列的排序列表, $Pos(s)$则返回优先队列 s 在排序列表中 S 的下标.

- MCMC 采样.

AUDE 应用一种广泛使用的 MCMC 方法——Metropolis-Hastings 算法[41,42]来迭代式地决定下一步选择的优先队列 s. 采样的过程是迭代式的, 在当前第θ轮迭代时, 下一个样本(即下一个被选择的优先队列 s)的选择依赖于上一轮θ–1 迭代时的 s 选择. 整个迭代选择过程类似于 Bernoulli 实验. 给定每一轮实验成功的可能性 p, 第 k 轮实验是第 1 次成功实验的概率可以通过下面公式(1)计算得到:

$$P_{Bernoulli}(X=k)=(1-p)^{(k-1)}p(k=1,2,3,\ldots,N_S) \tag{1}$$

其中, N_S 代表了所有优先队列 S 的总数目.

我们将在上一轮(第θ–1 轮)被选择的优先队列记为 $s_{\theta-1}$, 在当前轮(第θ轮)被选择的优先队列记为 s_θ. Metropolis-Hastings 算法按照公式(2)来计算状态从 $s_{\theta-1}$ 迁移到 s_θ之间的成功率:

$$P_{acc}(S_{\theta-1}\to S_\theta)=\min\left(1,\frac{P_{Bernoulli}(X=Pos(S_\theta))}{P_{Bernoulli}(X=Pos(S_{\theta-1}))}\right) \tag{2}$$

该计算公式可进一步简化为公式(3):

$$P_{acc}(S_{\theta-1}\to S_\theta)=\min(1,(1-p)^{Pos(S_\theta)-Pos(S_{\theta-1})}) \tag{3}$$

基于这种方式, 状态更容易从低优先级队列转移到高优先级队列, 而从高优先级到低优先级的转移虽然发生的概率较低但仍然存在, 并且它取决于高低优先级之间的排名差距. 一旦当前轮的优先队列 s 被决定, AUDE 就从该优先队列中随机选择一个未被执行的补丁, 并将其追加到初始补丁执行序列末端. AUDE 不断重复选择一个优先队列和从中选择一个补丁的过程, 直到完成初始补丁执行序列的构建.

类似于之前的相关研究[57,58], 我们遵循以下 3 点规则来估计每轮实验的成功率 p 的数值: (1) 所有实验成功率接近 1, 即 $0.95\leqslant\sum_{k=1}^{N_s}(1-p)^{(k-1)}p\leqslant 1$; (2) 选择第 1 个优先队列的概率大于 $1/N_S$, 即 $p>\frac{1}{N_s}$; (3) 仍然存在选择最低概率的优先队列的可能性, 即 $0.01\leqslant(1-p)^{N_s-1}p$. 在此基础上, p 的取值空间限定于$[p_{\min},p_{\max}]$, 我们采用中间值作为 p 的最后估值.

3.2 自适应性推导

对于初始执行序列中的每一个补丁, AUDE 根据已执行补丁信息来推导其具有信息性的概率, 并根据该概率决定执行或跳过该补丁.

根据 ProFL 的信息反馈机制, 当至少有一个原来失败的测试用例在某补丁上执行通过时, 该补丁 P 即被认为是有信息性的; 否则, 执行该补丁对于改进定位结果和修复效果只能提供非常有限的帮助. 换言之, 避免执行这样的补丁, 可以在不损失效果的情况下提升效率. 基于这样的设定, AUDE 采用了贝叶斯推导理论[59]公式(4), 根据已有的补丁执行结果来推导一个补丁是否具有信息性, 而不是直接执行所有未执行补丁:

$$p(X=1\mid\mathcal{O})=\frac{p(\mathcal{O}\mid X=1)p(X=1)}{p(\mathcal{O})} \tag{4}$$

其中, 随机变量 X 表示补丁 P 是否有信息性. X 是一个二元变量, 具体地, X=1 说明 P 具有信息性, X=0 说明 P

不具有信息性. $p(X=1|\mathcal{O})$代表了随机变量 X 基于当前观测结果$\mathcal{O}$上其具有信息性的后验概率, 它可以通过先验概率 $p(X=1)$、条件概率 $p(\mathcal{O}|X=1)$和概率 $p(\mathcal{O})$推导得到. 概率 $p(\mathcal{O})$标准化了联合后验概率, 可通过全概率公式(5)计算得到:

$$p(\mathcal{O})=p(\mathcal{O}|X=1)p(X=1)+p(\mathcal{O})p(X=0)p(X=0) \tag{5}$$

接下来, 我们来解释如何形式化地表示涉及到的概率.

- 先验概率.

$p(X=k)$(其中, $k=0$ 或 1)代表了观测到任何现象之前的初始先验概率分布. 考虑到一个补丁不具有信息性当没有任何原来失败的测试用例在该补丁上执行通过时, 其定义如等式(6)所示:

$$\begin{cases} p(X=1)=1-(1-p_{pass})^{n_f} \\ p(X=0)=(1-p_{pass})^{n_f} \end{cases} \tag{6}$$

其中, p_{pass} 代表了测试用例 t 在补丁 P 上通过的概率. 因为在推导的初始阶段我们并没有针对该测试用例执行结果的任何观测指标和先验知识, 因此对于所有测试用例 p_{pass}, 初始值为 0.5(即失败和通过的可能性是相等的). 这也是初始估计先验概率时的常见做法[60,61]. n_f 则代表了需要在补丁 P 上执行的原来失败的测试用例的总数目.

- 观测现象.

AUDE 通过相似补丁分组, 将当前所有补丁执行信息总结为一组观测现象$\mathcal{O}$. 考虑到效率问题, 在本文中, 我们将推导的范围限制在函数内部. 换言之, 当推导补丁 P 的执行结果时, 我们只考虑和 P 修改同一个函数的其他已执行补丁信息.

AUDE 首先根据测试用例覆盖情况, 将补丁分成 n_g 个组 $g_1,g_2,...,g_{n_g}$, 即被同样测试用例集合覆盖的补丁会被分到同一个组. 这里提到的测试用例覆盖补丁, 指的是测试用例覆盖了该补丁修改的代码位置. 我们将覆盖补丁 P 的测试用例集合记为 $T_{cov}(P)$. 对于任意在同一个补丁相似组 g_k 的两个补丁 P_i 和 P_j, 覆盖它们的测试用例集合是一样的, 即 $T_{cov}(P_i)=T_{cov}(P_j)$.

对于每个补丁相似组 g_k, 我们用观测结果 o_k 代表是否存在任一原来失败的测试用例在该组任一补丁上执行通过, 计算方式如公式(7):

$$o_k=\begin{cases} 1, \ \exists P\in g_k, \ \exists t\in T_{\text{cov}}(P)\cap T_f, \ P[t]=\text{pass} \\ 0, \ \forall P\in g_k, \ \forall t\in T_{\text{cov}}(P)\cap T_f, \ P[t]=\text{fail} \end{cases} \tag{7}$$

T_f 代表了原来失败的测试用例集合, $P[t]$代表了测试用例 t 在补丁 p 上的执行结果(失败或通过). 给定函数 Me 中的 n_g 个补丁相似组, 观测结果合集$\mathcal{O}$则是各个补丁组上观测结果的总集, $\mathcal{O}=\{o_1,o_2,o_3,...,o_{n_g}\}$.

- 条件概率估计.

因为存在多个组的观测结果, 条件概率 $p(\mathcal{O}|X=k)$可以被看成是相关事件的联合概率, 如公式(8):

$$p(\mathcal{O}\mid X=k)=p(o_1o_2...o_{n_g}\mid X=k)=\prod_{i=1}^{n_g}p(o_i\mid X=k) \tag{8}$$

根据公式(8)和公式(5), 最基础的推导公式(4)可以被进一步拓展为公式(9):

$$p(X=1\mid\mathcal{O})=\frac{\prod_{i=1}^{n_g}p(o_i\mid X=1)p(X=1)}{\prod_{i=1}^{n_g}p(o_i\mid X=1)p(X=1)+\prod_{i=1}^{n_g}p(o_i\mid X=0)p(X=0)} \tag{9}$$

$p(o_i|X=k)$代表了观测结果 o_i 在给定 $X=k$ 时的条件概率. 举例来说, $p(o_i=1|X=1)$则表示了当给定补丁 p 具有信息性时, 至少在补丁相似组 g_i 存在一个信息性补丁时的概率. 根据以下两点假设, 公式(10)定义了可能性函数 $p(o_i|X=k)$: (1) 两个补丁相似组之间越相近, 它们就有更大的可能性具有相同的观测结果且有更小的可能性

具有相反的观测结果; (2) 这两个补丁组中执行的补丁数目越多, 假设 1 的可信度就越高. 特别地, 观测结果 o_j 和给定补丁 P 都具有信息性 $p(o_j=1|X=1)$或都不具有信息性 $p(o_j=0|X=0)$的可能性正向关于(1)补丁 P 所在的补丁相似组 g_i 与 o_j 所在的补丁相似组 g_j 之间的相似度和可信度 $Con(g_j)$. 相反的可能性 $p(o_j=0|X=1)$和 $p(o_j=1|X=0)$则与这两个补丁相似组的相似度成反比:

$$\begin{cases} p(o_j=1|X=1)=\dfrac{1+(1-2e)Sim(g_j,g_i)Con(g_j)}{2} \\ p(o_j=0|X=1)=1-p(o_j=1|X=1) \\ p(o_j=0|X=0)=\dfrac{1+(1-2e)Sim(g_i,g_j)Con(g_j)}{2} \\ p(o_j=1|X=0)=1-p(o_j=0|X=0) \end{cases} \tag{10}$$

在公式(10)中, g_j 代表了观测结果 o_j 所在的补丁相似组, 而 g_i 代表了给定补丁 P 所属的补丁相似组. $Sim(g_i, g_j)$根据覆盖补丁的原失败测试用例集合计算了补丁相似组 g_i 对于 g_j 的相似度, 见公式(11):

$$Sim(g_i,g_j)=\frac{|T_{\text{cov}}(g_i)\cap T_{\text{cov}}(g_j)\cap T_f|}{|T_{\text{cov}}(g_j)\cap T_f|} \tag{11}$$

$Sim(g_i,g_j)$代表了相对相似性, 因此它具有不对称性, 即 $Sim(g_i, g_j)\neq Sim(g_j,g_i)$. $Con(g_j)$代表了观测结果 o_j 的可信度, 见公式(12):

$$Con(g_j)=\frac{|\{P\,|\,P\in g_j\wedge P\in P_{Exe}\}|}{|\{P\,|\,P\in g_j\}|} \tag{12}$$

其中, P_{Exe} 代表了当前执行的补丁. 当越来越多 g_j 内的补丁被执行时, $Con(g_j)$就会越大. 微量偏移 e(例如 0.05)将可能性 $p(o_j|X)$标准化到区间$[e,1-e]$, 从而避免公式(9)中出现除 0. 不直接使用 0.0 或者 1.0 而使用接近它们的数值, 也是概率推导中的常见做法[60,61].

在每一轮迭代结束后, AUDE 会根据针对补丁 p 推导得出的后验概率来概率性地执行 p, 并同时收集它的执行结果(如果 p 最终被执行了). 收集的新执行结果会同时用来重新更新观测结果和之前的概率, 从而为下一轮推导做准备.

4 实验设定

4.1 研究问题

- RQ1. 加速程度: AUDE 能够在多大程度上减少统一化调试中的测试执行?
- RQ2. 效果损失: AUDE 在缺陷定位和程序修复上引入了多少效果损失?
- RQ3. 方法开销: AUDE 中自身存在的时间开销是多大?
- RQ4. 敏感性分析: AUDE 方法中每一部分对于结果的提升各有多少帮助?
- RQ5. 参数分析:
  - RQ5a. 参数 e 的取值对 AUDE 的效果影响有多大?
  - RQ5b. AUDE 在不同的初始定位技术上表现如何?

4.2 验证数据集

我们在数据集 Defects4J 上所有的 395 个真实软件缺陷上对 AUDE 进行了实验验证. 我们选用了 Defects4J, 因为它是一个在缺陷定位领域和程序自动修复领域都被广泛使用的数据集. 表 2 描述了所涉及项目的基本信息, 其中, 列“ID”代表了项目的标识符, 列“项目名”和列“缺陷数目”则分别描述了项目的全名称及其所涉及到的缺陷版本总数, 列“行数”和列“测试数目”则罗列了该项目首版本的代码行数和测试用例个数.

表 2 验证数据集信息

ID	项目名	缺陷数目	测试数目	行数
Lang	Apache commons-lang	65	2 245	22K
Math	Apache commons-math	106	3 602	85K
Time	Joda-Time	27	4 130	28K
Chart	JFreeChart	26	2 205	96K
Closure	Google Closure compiler	133	7 927	90K
Mockito	Mockito	38	1 366	23K
Defects4J (1.2.0)		395	21 475	344K

4.3 实验自变量

为了衡量方法加速效果和因加速引入的效果损失(RQ1 和 RQ2), 我们将 AUDE 与以下几种策略进行比较.

(1) ProFL: 我们采用 ProFL 作为基础对比方法, 根据其相关工作描述[31,32], ProFL 按照补丁所修改代码的可疑度(由采用 Ochiai 公式的基于频谱的缺陷定位技术计算得出)降序执行每个补丁, 对于每个补丁, 优先执行原来失败的测试用例再执行原来通过的测试用例, 终止当前补丁执行当有任一测试用例当前执行的发生失败时;

(2) $\text{ProFL}_{\text{plau}}$: $\text{ProFL}_{\text{plau}}$ 在 ProFL 的基础上引入提早终止策略, 即: 当找到任一似真补丁时, 整个修复过程终止. 需要注意的是: 对于 ProFL, 即使有似真补丁被发现, 整个修复过程仍然会继续进行, 直到所有补丁都被执行;

(3) $\text{ProFL}_{\text{rand}}$: $\text{ProFL}_{\text{rand}}$ 是 ProFL 的一个变式, 它按照随机顺序执行所有补丁, 当其执行了与 AUDE 相同数目的补丁时, 终止整个修复过程;

(4) $\text{ProFL}_{\text{sbfl}}$: $\text{ProFL}_{\text{sbfl}}$ 也是 ProFL 的一个变式, 它按照可疑度降序来执行每个补丁, 当它执行了与 AUDE 相同数目的补丁时, 终止整个修复过程,.

$\text{ProFL}_{\text{plau}}$, $\text{ProFL}_{\text{rand}}$ 和 $\text{ProFL}_{\text{sbfl}}$ 都是为了与 AUDE 进行有针对性的比较而在本文中首次提出的对比策略.

为了衡量 AUDE 中每一部分对整体效果的贡献(RQ4), 我们进一步设计了 AUDE 的 4 种变式, 通过将 AUDE 中原有策略依次进行替换, 从而帮助进行敏感性分析.

(1) $\text{AUDE}_{\text{Random+Infer}}$: 在初始优化阶段, 我们将 MCMC 采样策略替换为随机采样策略, 即按照随机顺序构建初始补丁执行序列;

(2) $\text{AUDE}_{\text{SBFL+Infer}}$: 在初始优化阶段, 我们将 MCMC 采样策略替换成启发式策略, 即严格按照补丁修改的代码元素的初始可疑度值进行降序排列, 从而构建初始补丁执行序列;

(3) $\text{AUDE}_{\text{MCMC+NoInfer}}$: 在自适应性推导阶段, 我们移除概率推导策略, 而直接执行所有补丁;

(4) $\text{AUDE}_{\text{MCMC+RDInfer}}$: 在自适应性推导阶段, 我们将概率推导策略替换成随机推导策略, 即随机判定一个未执行的补丁是否具有信息性.

因此, 通过将 AUDE 与 $\text{AUDE}_{\text{Random+Infer}}$、$\text{AUDE}_{\text{SBFL+Infer}}$ 进行比较, 我们可以探索在初始优化阶段设计的 MCMC 采样方法的有效性; 通过将 AUDE 与 $\text{AUDE}_{\text{MCMC+NoInfer}}$、$\text{AUDE}_{\text{MCMC+RDInfer}}$ 进行比较, 我们可以探索在自适应性推导阶段设计的贝叶斯推导方法的有效性.

为了衡量 AUDE 中唯一的可调节参数 e 对整体效果的影响, 在 RQ5a 中, 我们研究了 e 分别在 0.05、0.025、0.050、0.075、0.010 时 AUDE 的效果. 在其他 RQ 中, 我们默认采用 e=0.050.

4.4 实验因变量

我们使用下列因变量来衡量 AUDE 的效果和效率.

- 执行缩减率.

为了衡量 AUDE 的加速能力, 分别计算: (1) AUDE 比 ProFL 少执行的补丁数目与 ProFL 执行的补丁总数目的比例; (2) AUDE 比 ProFL 少执行的测试数目与 ProFL 执行的测试数目的比例.

- 效果降低率.

根据之前的相关研究[31,32,45], 我们在实验中采用下述指标衡量缺陷定位在 AUDE 加速前后的效果.

(1) Top-*N*: Top-*N* 指的是至少有一个缺陷代码元素被排在可疑度列表前 *N* 位的缺陷版本总数. 根据相关工作显示[45], 开发者通常只检查位于可疑度列表前几位的代码元素. 例如, 73.58%的开发者只检查前 Top-5 的代码元素[62]. 因此, 遵循之前的工作[38,49,63], 我们在实验中采用 Top-*N* 指标(其中, *N*=1,3,5);

(2) 首要平均排名(mean first rank, MFR): 对于每个项目, MFR 代表了其所有缺陷版本中出错代码元素在可疑度列表中最靠前排名的平均值;

(3) 综合平均排名(mean average rank, MAR): 对于每个项目, MAR 代表了其所有缺陷版本中所有出错代码元素在可疑度列表中平均排名的平均值.

MFR 关注出错代码元素的最优排名, 而 MAR 关注出错代码的平均排名, 这对于包含多处出错代码的缺陷来说尤其重要. 对于可疑度相同的代码元素, 根据之前的工作[24,25,33], 我们采用其中的最差排名. 例如: 一个出错代码元素和一个正确代码元素都以同样的可疑度值并列第 k 位, 则二者的排名都作为第 k+1 位. 对于程序自动修复的效率, 我们使用成功生成了似真补丁的缺陷总数来衡量. 因使用 AUDE 而导致失去似真补丁的缺陷个数和 ProFL 中原来所有拥有似真补丁的缺陷个数的比例, 则代表了修复效果损失.

- 方法开销.

我们分别记录了在初始优化阶段(计算优先队列和 MCMC 采样)和自适应性推导(概率推导)过程中引入的时间开销, 作为 AUDE 的执行开销.

4.5 方法实现

AUDE 在最新统一化调试方法 ProFL 上加以实现. ProFL 已经开发成 Maven 插件, 它集成了最新的字节码级别程序修复技术 PraPR, 为出错程序生成和执行补丁, 最后返回似真补丁和精化后的缺陷定位列表. 所有的实验都在设备 Dell Workstation with Intel(R) Xeon(R) Gold 6138 CPU@2.00 GHz、330 GB RAM、Ubuntu 18.04.1 LTS 上进行.

4.6 有效性威胁分析

内部有效性的威胁主要来自于我们技术的实现以及对其他已有技术的重现. 为了减轻该威胁, 我们直接重用了 ProFL 的相关代码, 并在此基础上实现了 AUDE; 同时, 我们再三检查了代码. 构建有效性威胁主要来自实验中选取的衡量指标, 为了减少该威胁, 我们采用了大量在之前工作广泛使用的指标. 遵循之前在缺陷定位和程序修复领域的工作[38,39,48,49], 我们选择了 Defects4J 作为本文的验证数据集. Defects4J 是在缺陷定位和程序修复领域使用最为广泛的数据集, 同时也包含了上百个软件开发过程中的真实缺陷. 该做法可以在一定程度上缓解数据集所带来的通用性问题.

5 实验结果

表 3 汇总了在 RQ1 和 RQ2 上的实验结果. 我们分别展示了 AUDE 在每一个项目和总体上的性能和效果. 为了展示统一化调试技术在精化缺陷定位上的提升, 我们同时还展示了主流基于频谱的缺陷定位技术 SBFL(采用 Ochiai 公式和聚集策略)和主流的基于变异的缺陷定位技术 MBFL(Metallaxis)结果. 其中,

- 列“技术”表示所有涉及到的比较技术, 包括 ProFL;
- 列“执行加速”分别展示了缩减的执行补丁数目和执行测试数目, 其中, 列“#补丁”代表了执行的补丁数目, 后面紧跟着的列“减少%”则计算了缩减的补丁执行数目占 ProFL 执行补丁数目的比例(因此这项指标对于 ProFL 来说总是 0);
- 相似地, 列“#测试”代表了执行测试的总数, 后面紧跟着的列“减少%”则计算了缩减的测试执行数目占 ProFL 执行测试数目的比例;
- 列“缺陷定位效果损失”代表了因为加速而引入的在 Top-1、Top-3、Top-5、MFR、MAR 上的效果损失, 以及这些损失对比原来 ProFL 效果所占的比例(即列“δ”). 特别地, 向上的箭头代表了效果的提

升，向下的箭头代表了效果的降低;

- 列“修复效果损失”代表了产生似真补丁的缺陷数目，以及比 ProFL 原有似真修复数目降低的比例.

考虑到 MCMC 采样和推导过程中存在的随机性，所有的实验结果都是执行 10 遍之后的平均值.

表 3 执行加速和效果损失

项目	技术	执行加速				修复效果损失	
		#补丁	减少(%)	#测试	减少(%)	#似真	δ
Lang	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	710.76	0.00	740.02	0.00	18.00	0.00
	$\text{ProFL}_{\text{plau}}$	592.00	17.52	606.35	17.29	18.00	0.00
	$\text{ProFL}_{\text{rand}}$	209.18	71.55	212.48	71.64	7.67	↓10.33
	$\text{ProFL}_{\text{sbfl}}$	217.79	70.04	222.18	70.01	4.00	↓14.00
	AUDE	217.33	69.74	223.47	69.43	15.67	↓2.33
Math	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	3 401.23	0.00	3 896.85	0.00	37.00	0.00
	$\text{ProFL}_{\text{plau}}$	1 871.16	30.26	2 069.13	30.59	37.00	0.00
	$\text{ProFL}_{\text{rand}}$	641.03	73.94	725.5	73.72	14.00	↓23.00
	$\text{ProFL}_{\text{sbfl}}$	661.19	73.28	743.44	73.47	13.00	↓24.00
	AUDE	732.58	71.43	831.96	71.50	35.00	↓2.00
Mockito	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	3 396.42	0.00	3 651.47	0.00	5.00	0.00
	$\text{ProFL}_{\text{plau}}$	3 060.69	11.86	3 268.89	11.83	5.00	0.00
	$\text{ProFL}_{\text{rand}}$	1 304.56	62.27	1 385.79	62.38	1.00	↓4.00
	$\text{ProFL}_{\text{sbfl}}$	1 286.36	62.93	1 421.22	62.21	3.00	↓2.00
	AUDE	1 307.10	61.92	1 463.38	60.76	4.33	↓0.67
Time	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	3 979.92	0.00	4 607.19	0.00	5.00	0.00
	$\text{ProFL}_{\text{plau}}$	2874.69	18.99	3 434.19	18.97	5.00	0.00
	$\text{ProFL}_{\text{rand}}$	1 160.57	70.38	1 373.01	70.63	0.00	↓5.00
	$\text{ProFL}_{\text{sbfl}}$	1 154.15	70.61	1 399.12	70.54	3.00	↓2.00
	AUDE	1 161.57	70.10	1 337.87	70.56	5.00	0.00
Chart	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	5 697.32	0.00	6 398.80	0.00	14.00	0.00
	$\text{ProFL}_{\text{plau}}$	817.52	48.74	856.52	47.83	14.00	0.00
	$\text{ProFL}_{\text{rand}}$	487.29	74.77	502.29	74.98	3.33	↓10.67
	$\text{ProFL}_{\text{sbfl}}$	433.68	75.21	458.64	74.47	10.00	↓4.00
	AUDE	510.25	73.33	543.40	72.35	12.33	↓1.67
Closure	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	31 920.72	0.00	39 817.15	0.00	52.00	0.00
	$\text{ProFL}_{\text{plau}}$	18 920.93	38.29	22 058.20	37.16	52.00	0.00
	$\text{ProFL}_{\text{rand}}$	8 027.92	73.82	9 211.96	73.89	7.67	↓44.33
	$\text{ProFL}_{\text{sbfl}}$	7 735.32	74.53	9 532.55	73.50	20.00	↓32.00
	AUDE	8 167.25	73.38	10 126.00	71.97	51.67	↓0.33
All	SBFL	–	–	–	–	–	–
	Metallaxis	–	–	–	–	–	–
	ProFL	12 883.98	0.00	15 828.54	0.00	131.00	0.00
	$\text{ProFL}_{\text{plau}}$	7 592.55	29.61	8 778.77	29.20	131.00	0.00
	$\text{ProFL}_{\text{rand}}$	3 179.88	72.21	3 630.19	72.23	33.67	↓97.33
	$\text{ProFL}_{\text{sbfl}}$	3 081.05	72.13	3 748.21	71.70	53.00	↓78.00
	AUDE	3 254.82	70.90	3 979.78	70.29	124.00	↓7.00

表 3 执行加速和效果损失(续)

项目	技术	定位效果损失									
		Top-1	δ	Top-3	δ	Top-5	δ	MFR	δ	MAR	δ
Lang	SBFL	30.00	–	50.00	–	54.00	–	3.16	–	3.44	–
	Metallaxis	31.00	–	51.00	–	56.00	–	3.30	–	3.66	–
	ProFL	40.00	0.00	55.00	0.00	59.00	0.00	1.95	0.00	2.37	0.00
	$\text{ProFL}_{\text{plau}}$	40.00	0.00	55.00	0.00	59.00	0.00	1.95	0.00	2.37	0.00
	$\text{ProFL}_{\text{rand}}$	36.33	↓3.67	55.00	0.00	58.33	↓0.67	2.20	↓0.25	2.58	↓0.21
	$\text{ProFL}_{\text{sbfl}}$	35.00	↓5.00	52.00	↓3.00	57.00	↓2.00	2.55	↓0.60	3.00	↓0.63
	AUDE	38.00	↓2.00	54.67	↓0.33	58.33	↓0.67	2.10	↓0.15	2.55	↓0.18
Math	SBFL	31.00	–	61.00	–	76.00	–	5.74	–	5.99	–
	Metallaxis	21.00	–	52.00	–	68.00	–	8.18	–	7.19	–
	ProFL	46.00	0.00	71.00	0.00	80.00	0.00	5.18	0.00	5.44	0.00
	$\text{ProFL}_{\text{plau}}$	47.00	↑1.00	69.00	↓2.00	77.00	↓3.00	5.05	↑0.13	5.26	↑0.18
	$\text{ProFL}_{\text{rand}}$	39.33	↓6.67	65.00	↓6.00	77.00	↓3.00	5.10	↑0.08	5.39	↑0.05
	$\text{ProFL}_{\text{sbfl}}$	44.00	↓2.00	66.00	↓5.00	76.00	↓4.00	5.23	↓0.05	5.51	↓0.07
	AUDE	46.33	↑0.33	67.33	↓3.67	79.00	↓1.00	4.93	↑0.25	5.16	↑0.28
Mockito	SBFL	11.00	–	18.00	–	23.00	–	9.08	–	13.32	–
	Metallaxis	5.00	–	9.00	–	13.00	–	35.94	–	44.62	–
	ProFL	10.00	0.00	21.00	0.00	25.00	0.00	6.89	0.00	10.24	0.00
	$\text{ProFL}_{\text{plau}}$	1.00	0.00	20.00	↓1.00	24.00	↓1.00	6.78	↑0.11	10.13	↑0.11
	$\text{ProFL}_{\text{rand}}$	9.67	↓0.33	21.33	↑0.33	26.33	↑0.133	6.60	↑0.29	10.97	↓0.73
	$\text{ProFL}_{\text{sbfl}}$	10.00	0.00	21.00	0.00	27.00	↑2.00	6.00	↑0.89	10.77	↓0.53
	AUDE	10.67	↑0.67	23.00	↑2.00	26.00	↑1.00	7.10	↓0.21	11.01	↓0.77
Time	SBFL	7.00	–	15.00	–	18.00	–	11.31	–	12.65	–
	Metallaxis	7.00	–	12.00	–	15.00	–	11.80	–	14.36	–
	ProFL	9.00	0.00	13.00	0.00	15.00	0.00	11.77	0.00	13.39	0.00
	$\text{ProFL}_{\text{plau}}$	9.00	0.00	13.00	0.00	16.00	↑1.00	11.62	↑0.15	13.24	↑0.15
	$\text{ProFL}_{\text{rand}}$	5.33	↓3.67	10.67	↓2.33	17.67	↑2.67	11.38	↑0.39	12.89	↑0.50
	$\text{ProFL}_{\text{sbfl}}$	7.00	↓2.00	14.00	↑1.00	18.00	↑3.00	12.73	↓0.96	14.10	↓0.71
	AUDE	8.67	↓0.33	14.67	↑1.67	16.33	↑1.33	11.33	↑0.44	12.87	↑0.52
Chart	SBFL	7.00	–	16.00	–	17.00	–	5.80	–	6.56	–
	Metallaxis	8.00	–	16.00	–	18.00	–	12.04	–	12.66	–
	ProFL	9.00	0.00	19.00	0.00	21.00	0.00	3.92	0.00	4.66	0.00
	$\text{ProFL}_{\text{plau}}$	12.00	↑3.00	17.00	↓2.00	19.00	↓2.00	5.04	↓1.12	5.88	↓1.22
	$\text{ProFL}_{\text{rand}}$	8.67	↓0.33	16.67	↓2.33	18.67	↓2.33	5.21	↓1.29	6.05	↓1.39
	$\text{ProFL}_{\text{sbfl}}$	11.00	↑2.00	18.00	↓1.00	18.00	↓3.00	5.40	↓1.48	6.22	↓1.56
	AUDE	12.33	↑3.33	18.33	↓0.67	20.00	↓1.00	4.16	↓0.24	4.98	↓0.32
Closure	SBFL	31.00	–	59.00	–	71.00	–	44.20	–	58.32	–
	Metallaxis	20.00	–	45.00	–	61.00	–	22.07	–	26.05	–
	ProFL	43.00	0.00	63.00	0.00	81.00	0.00	18.15	0.00	29.78	0.00
	$\text{ProFL}_{\text{plau}}$	44.00	↑1.00	61.00	↓2.00	76.00	↓5.00	30.84	↓12.69	42.29	↓12.51
	$\text{ProFL}_{\text{rand}}$	30.00	↓13.00	57.33	↓5.67	71.33	↓9.67	31.73	↓13.58	44.24	↓14.16
	$\text{ProFL}_{\text{sbfl}}$	36.00	↓7.00	60.00	↓3.00	78.00	↓3.00	44.71	↓26.56	57.26	↓27.48
	AUDE	45.00	↑2.00	64.67	↑1.67	80.33	↓0.67	24.65	↓6.50	36.73	↓6.95
All	SBFL	117.00	–	219.00	–	259.00	–	19.15	–	24.63	–
	Metallaxis	94.00	–	191.00	–	244.00	–	14.28	–	16.93	–
	ProFL	157.00	0.00	242.00	0.00	281.00	0.00	9.61	0.00	14.20	0.00
	$\text{ProFL}_{\text{plau}}$	162.00	↑5.00	235.00	↓7.00	271.00	↓10.00	13.96	↓4.35	18.48	↓4.28
	$\text{ProFL}_{\text{rand}}$	129.33	↓27.67	226.00	↓16.00	269.33	↓11.67	14.29	↓4.68	19.28	↓5.08
	$\text{ProFL}_{\text{sbfl}}$	143.00	↓14.00	231.00	↓11.00	274.00	↓7.00	18.86	↓9.25	23.90	↓9.70
	AUDE	161.00	↑4.00	242.67	↑0.67	280.00	↓1.00	11.79	↓2.17	16.58	↓2.38

5.1 加速程度和效果损失(RQ1和RQ2)

表 3 说明, AUDE 显著地加速了统一化调试方法 ProFL. 特别地, 与 ProFL 相比, AUDE 平均减少了其70.29%的测试执行; 与此同时, 从 Top-1、Top-3、Top-5 和似真修复的降低程度(例如 5.34%)来看, AUDE 在缺陷定位和程序修复上仍然保持了与 ProFL 同样突出的效果. 总的来说, 实验结果说明了 AUDE 在只引入了可以忽略的效果损失的情况下, 非常显著地加速了统一化调试方法. 我们将从以下几个角度进一步分析和阐述相关的实验结论.

- 执行缩减.

总的来说, AUDE 减少了 70.29%的测试执行却保持了与 ProFL 相当的效果, 这说明,在统一化调试原有的补丁执行信息中存在大量的冗余信息. 这进一步支持了本文一开始的研究动机, 即在统一化调试场景下, 只有一小部分补丁执行信息可以为改善缺陷定位提供有效的反馈信息.

- 缺陷定位效果.

总的来说, AUDE 和 ProFL 在缺陷定位上有着相当的表现, 两者都显著超过了主流的基于频谱和基于变异的缺陷定位技术. 例如, 至少有 44 多个缺陷被提升到了 Top-1 的定位. 特别地, AUDE 在 Top-1、Top-3 和 Top-5 上与 ProFL 的表现极其相近, 甚至在只执行不到 ProFL 的 30%测试数目的前提下, AUDE 比 ProFL 还增加了 4 个排到 Top-1 的缺陷. 我们进一步人工检查了这些被 AUDE 排到 Top-1 而没有被 ProFL 排到 Top-1 的缺陷, 发现 AUDE 跳过了那些发生在正确代码上的正向补丁的执行而保留了发生在错误代码上的正向补丁的执行, 而这恰恰就可以将错误代码在可疑度列表上排到比正确代码更靠前的位置. 例如, Chart-26 中出错的函数是 drawLabel, 它在初始阶段被基于频谱的缺陷定位技术排到了第 22 名(可疑度为 0.74). ProFL 重新对该函数进行了排名调整: 因为真正出错的函数 drawLabel 上存在正向补丁(*P*1, 同时也是一个似真补丁)而正确函数 draw (可疑度为 0.84)也存在正向补丁(*P*2), 因此正确函数 draw 被重新排到了 Top-1, 而出错函数 drawLabel 被排到了 Top-2 (drawLabel 的初始可疑度小于 draw 的初始可疑度). 但在 AUDE 中, MCMC 采样过程将补丁 *P*1 的执行排到了 *P*2 之前, 所以整个修复过程在 *P*1 执行后(似真补丁被发现)就终止了. 因此, 真正出错的函数 drawLabel 作为唯一拥有正向补丁的函数, 被 AUDE 排到了可疑度列表的第 1 名. 这个例子充分说明了初始优化补丁执行序列的必要性: 如果补丁总是按照初始的可疑度降序排列执行(即当前 ProFL 采取的方式), 正确的函数则会一直比真正错误的函数有更大或者相等的机会发现正向补丁, 因而就更容易被排到靠前的位置. 换言之, 该实验结果说明, 存在一些补丁执行信息对反馈调节缺陷定位效果起到了负面影响, 而 AUDE 可以通过跳过这些执行, 从而缓解其所带来的负面影响.

同时, 我们也发现 AUDE 在 MFR 和 MAR 上存在微小的效果下降. 例如, MFR 从 14.20 增加到了 16.58. 其实, 缺陷的平均排名(即 MFR 和 MAR)的降低是由于似真补丁触发的提早退出机制引入的. 例如, 在 Closure 中, $\text{ProFL}_{\text{plau}}$ 的 MAR 比 ProFL 显著变差(从 29.78 变到了 42.29), 而该降低程度实际上在 AUDE 中反而有所缓解. 同时, 尽管存在着效果的降低, 但在 MAR 和 MFR 上, AUDE 仍然显著优于主流的其他缺陷定位技术(例如, 基于频谱和基于变异的缺陷定位技术). 根据之前的相关工作[62], 大多数开发者只愿意检查推荐列表前 5 名之前的程序元素. 因此, 基于 Top-1、Top-3 和 Top-5, 我们认为, 当前在 MAR 和 MFR 上的损失是可以接受的.

- 程序修复效果损失.

根据列“似真修复”, 在 131 个可以被 ProFL 似真修复的缺陷中, 有 7 个被 AUDE 错误地丢弃了, 即对于每个项目平均少了 1.16 个可以被似真修复的缺陷. 因为自动地进行正确补丁的识别在实现上非常具有挑战性, 因此我们人为地检查了这 7 个被丢失的似真补丁, 发现这其中包含了两个与正确补丁完全等价的补丁. 换言之, 与 ProFL 相比, AUDE 以少修复两个缺陷作为代价, 减少了 70.29%的测试执行. 并且, 我们进一步检查了这 7 个在 AUDE 中无法被似真修复的缺陷, 发现它们被跳过执行的似真补丁所在的函数还有其他被执行的正向补丁, 这说明 AUDE 仍收集了有利于改善缺陷定位的反馈信息. 考虑到 ProFL 背后的核心思想是利用不成功的修复来逆向改进不精准的定位, 在统一化调试的场景下, 这样的修复效果降低是可以接受的.

- 加速程度和效果损失的平衡.

与其他对比加速策略相比较($\text{ProFL}_{\text{plau}}$、$\text{ProFL}_{\text{rand}}$ 和 $\text{ProFL}_{\text{sbfl}}$), AUDE 以最小的效果损失取得了最大的效率提升. 例如, $\text{ProFL}_{\text{plau}}$ 只减少了 29.20%的测试用例执行(不到 AUDE 加速比的一半). 在 MFR 上, AUDE 以 15.54%优于 $\text{ProFL}_{\text{plau}}$, 但少了 7 个似真修复. 值得注意的是: $\text{ProFL}_{\text{plau}}$ 这种策略从来不会跳过似真补丁的执行, 因为它并没有引入由似真补丁出发的提前退出修复机制. 比较 $\text{ProFL}_{\text{plau}}$ 和 AUDE, 说明了在效果相近的情况下, AUDE 更加显著地加速了 ProFL ($\text{ProFL}_{\text{plau}}$ 提升了 2.44 倍). 至于 $\text{ProFL}_{\text{rand}}$ 和 $\text{ProFL}_{\text{sbfl}}$, 当它们的补丁执行

数目被控制为与 AUDE 相同时(即控制加速程度一致), 它们的效果明显差于 AUDE. 以缺陷定位的效果为例, 它们的 Top-*N* 全都下降了 10–20 左右, 并且它们的似真修复损失也非常巨大. 例如, $\text{ProFL}_{\text{sbfl}}$ 错误地跳过了 78 个缺陷的似真修复(59.54%), 而 $\text{ProFL}_{\text{rand}}$ 则错误地跳过了 97.33 个缺陷的似真修复(74.30%). 该结果说明, AUDE 在所有加速策略中是效果保持和效率提升平衡得最好的加速技术.

总的来说, AUDE 在显著加速 ProFL 的同时保持了与其相当的缺陷定位和程序修复效果, 并且在效果保持和效率提升上显著优于其他加速策略.

5.2 方法开销(RQ3)

考虑到 AUDE 直接建立在 ProFL 之上, 并且比起 ProFL, AUDE 并不需要其他额外的输入信息(例如补丁执行和测试覆盖信息同时都是 ProFL 的输入, 因此天然地被 ProFL 所收集), AUDE 并不涉及到信息收集的额外开销. 因此, 我们将计算时间开销作为 AUDE 的执行开销, 并将结果呈现于表 4. 其中, 行“初始优化”代表了 AUDE 在构建初始序列时, 由于计算优先队列分组和 MCMC 采样时引入的时间开销; 行“自适应性推导”则代表了在自适应性推导阶段, 由于概率推导引入的时间开销; 行“总时间”则计算了上述二者的总和. 从表 4 可以看到, AUDE 的总体开销是以秒来计算的. 对于 Closure(数据集中规模最大的项目), AUDE 的计算代价也总共花费了 15.17 s, 这个时间开销与 Closure 进行统一化调试的整体时间(两个小时[32,33])相比可以忽略不计. 实验结果说明, AUDE 是一项开销很小的技术.

表 4 AUDE 时间开销 (s)

时间	Lang	Math	Mockito	Time	Chart	Closure	All
初始优化	0.01	0.07	0.16	0.30	0.41	8.11	2.85
自适应性推导	0.00	0.03	0.21	0.12	0.02	7.05	2.44
总时间	0.01	0.10	0.37	0.42	0.43	15.16	5.29

5.3 敏感性分析(RQ4)

为了探索 AUDE 各个部分的有效性(即初始优化阶段的 MCMC 采样方法和自适应性推导阶段的贝叶斯推导方法), 我们通过依次替换这些策略为其他策略, 生成 4 个 AUDE 的变化策略, 通过比较它们的性能来进行敏感性分析. 表 5 分别呈现了这些策略在加速效果、缺陷定位效果和程序修复效果上的表现.

表 5 敏感性分析

技术	执行加速				修复效果损失	
	#补丁	减少(%)	#测试	减少(%)	#似真	δ
AUDE	3 254.82	70.90	3 979.78	70.29	124.00	↓7.00
$\text{AUDE}_{\text{Random+Infer}}$	3 499.18	70.44	4 402.09	69.73	93.50	↓37.50
$\text{AUDE}_{\text{SBFL+Infer}}$	3 480.49	68.47	4 358.32	67.60	110.00	↓21.00
$\text{AUDE}_{\text{MCMC+NoInfer}}$	7 707.94	30.22	8 776.72	30.09	131.00	0.00
$\text{AUDE}_{\text{MCMC+RDInfer}}$	4 682.03	55.07	5 315.23	55.09	106.50	↓25.40

表 5 敏感性分析(续)

技术	定位效果损失									
	Top-1	δ	Top-3	δ	Top-5	δ	MFR	δ	MAR	δ
AUDE	161.00	↑4.00	242.67	0.67	280.00	↓1.00	11.79	↓2.17	16.58	↓2.38
$\text{AUDE}_{\text{Random+Infer}}$	143.70	↓13.30	230.20	↓11.80	268.00	↓13.00	13.95	↓4.34	19.05	↓4.85
$\text{AUDE}_{\text{SBFL+Infer}}$	152.70	↓4.30	235.00	↓7.00	273.50	↓7.50	13.84	↓4.23	18.90	↓4.70
$\text{AUDE}_{\text{MCMC+NoInfer}}$	163.70	↑6.30	239.50	↓2.50	278.50	↓2.50	13.24	↓3.63	17.26	↓3.06
$\text{AUDE}_{\text{MCMC+RDInfer}}$	148.40	↓8.60	231.20	↓10.80	274.00	↓7.00	13.53	↓3.92	18.61	↓4.41

为了综合各项指标进行更全面的比较, 对于每一个指标, 我们进一步计算了它在 4 个技术和 AUDE 上的比例, 并绘制成如图 2 和图 3 所示. 在图中, AUDE 被视为 100%的基准方法(即, 它所有的指标都是 100%). 总体上, 我们可以发现, AUDE 比所有的变式方法在大多数指标上都要更好, 说明 AUDE 中每个部分的策略都是有效的, 并且有自己的贡献. 我们接下来对每一部分进行更详细的分析.

- MCMC 采样策略的有效性.

当将初始优化阶段的 MCMC 采样策略替换成启发式和随机策略之后(这些策略的具体描述可参考第 4.3 节), $AUDE_{SBFL+Infer}$ 和 $AUDE_{Random+Infer}$ 在缺陷定位和程序修复上的效果都变差了. 特别地, 从表 5 中可以发现, $AUDE_{SBFL+Infer}$ 和 $AUDE_{Random+Infer}$ 比 AUDE 分别少产生了 21 和 37.5 个似真修复, 在图 2 显示将近 10%和 20%. 另外, $AUDE_{SBFL+Infer}$ 和 $AUDE_{Random+Infer}$ 在缺陷平均排名(MAR 和 MFR)上也有轻微的效果降低. 另一方面, 根据图 2, $AUDE_{SBFL+Infer}$ 和 $AUDE_{Random+Infer}$ 在执行加速上与 AUDE 的表现相近, 这说明, MCMC 采样策略的主要贡献是保持 ProFL 原有的定位和修复效果.

- 概率推导策略的有效性.

相似地, 为了探索在自适应性推导阶段贝叶斯推导策略的有效性, 我们将其替换成了直接执行($AUDE_{MCMC+NoInfer}$)和随机推导策略($AUDE_{MCMC+RDInfer}$), 并且将它们的性能与 AUDE 进行对比. 根据表 5 和图 3, 将贝叶斯推导替换为其他策略, 很大程度上降低了加速效果, 说明该推导策略的主要贡献在于缩减测试执行. 并且, 将推导策略替换成随机推导会导致缺陷定位(降低 12.60 个 Top-1 缺陷)和自动修复效果(降低 17.50 个似真修复)变差. 为了确认这一现象, 对于所有技术, 我们对缺陷的排名和执行加速比例进行了基于 Bonferroni corrections[64]的 Wilcoxon signed-rank 检验[65]. 检验结果显示: 在显著水平为 0.05 时, 它们存在着显著性差异.

总的来说, AUDE 中的两个部分都对 AUDE 的整体效果起到了重要的作用: MCMC 采样方法在保持缺陷定位和自动修复效果上起到了重要的作用, 而推导策略则主要加剧了测试执行缩减.

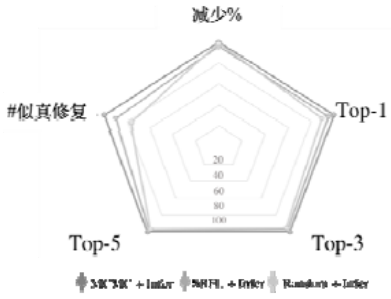

图 2 MCMC 采样敏感性分析

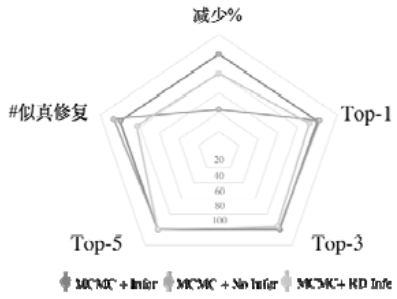

图 3 自适应推导敏感性分析

5.4 参数分析(RQ5)

本研究问题进而研究参数 e 和初始定位技术对于 AUDE 效果的影响.

- RQ5a.

本文对 e 在取值为{0.05,0.025,0.05,0.075,0.010}时, 对 AUDE 的加速程度和效果损失(Top-1、Top-3、Top-5、MAR、MFR 和似真修复数目)进行了比较. 结果发现, 各项指标表现接近, 其中, 加速比为 69.78% (σ=2.15), Top-1 为 160.60 (σ=2.07), 似真修复数目为 123.00 (σ=1.58). 同样地, 对于不同的 e, 我们在每个项目上进行了基于 Bonferroni corrections 的 Wilcoxon signed-rank 检验. 检验结果显示, 它们在各项目内部均不存在显著性差异;

- RQ5b.

上文中, 本文遵循 ProFL 的初始设定, 采用 Ochiai 作为初始定位函数. 本文进而研究 AUDE 分别以已有 34 种基于频谱的定位技术作为初始定位函数的结果. 图 4 展示了 34 种技术各自的加速比, 其中, 横轴为 34 种频谱公式, 纵轴为加速比. 平均加速比为 69.52%.

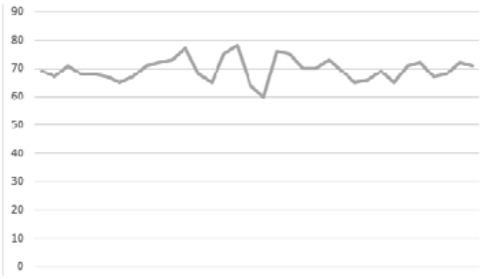

图 4　不同频谱公式的加速结果

6 结　论

在本文中, 我们提出了首个针对统一化调试方法的加速技术 AUDE, 通过减少对改善缺陷定位无帮助的补丁执行, 从而在保持统一化调试效果的同时实现大幅加速. 具体来说:

- 首先, 基于对多样性和可疑度优先级的同时考虑, AUDE 应用了基于 MCMC 采样方法构建了初始补丁执行序列;
- 其次, 对于序列中每一个补丁, AUDE 动态地分析当前已执行的补丁信息以推导其具有信息性的可能性. 我们在数据集 Defects4J 中的 395 个缺陷上进行了实验验证, 结果表明, AUDE 减少了 ProFL 中 70.29%的测试执行, 显著地加速了 ProFL; 同时, 在缺陷定位和程序修复上只引入了微弱的效果损失. 例如: 对于与 ProFL 相近数目的缺陷, AUDE 可以把出错的代码元素排到 Top-1、Top-3 和 Top-5; 同时, AUDE 只引入了秒级的时间开销, 说明其是一种高效的加速方法;
- 最后, 针对于方法各部分策略所进行的敏感性分析, 也说明了 AUDE 中每一部分的策略都对 AUDE 的整体效果起到了重要的作用.

References:

[1] Lou YL, Zhu QH, Dong JH, Li X, Sun ZY, Hao D, Zhang L, Zhang LM. Boosting coverage-based fault localization via graph-based representation learning. In Proc. of the ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (FSE 2021). ACM, 2021. 664–676. https://doi.org/10.1145/3468264.3468580

[2] Briand LC, van Labiche Y, Liu XT. Using machine learning to support debugging with tarantula. In: Proc. of the ISSRE. 2007. 137−146.

[3] Dallmeier V, Lindig C, Zeller A. Lightweight defect localization for Java. In: Proc. of the ECOOP. 2005. 528−550.

[4] Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. 2005. In: Proc. of the Scalable Statistical Bug Isolation. 2005. 15−26.

[5] Moon SY, Kim YH, Kim MZ, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. In: Proc. of the 7th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST 2014). IEEE, 2014. 153−162.

[6] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. In: Proc. of the 39th Int'l Conf. on Software Engineering. 2017. 609−620.

[7] Roychowdhury S, Khurshid S. A novel framework for locating software faults using latent divergences. In: Proc. of the ECML. 2011. 49−64.

[8] Roychowdhury S, Khurshid S. Software fault localization using feature selection. In: Proc. of the Int'l Workshop on Machine Learning Technologies in Software Engineering. 2011. 11−18.

[9] Roychowdhury S, Khurshid S. A family of generalized entropies and its application to software fault localization. In: Proc. of the Int'l Conf. on Intelligent Systems. 2012. 368−373.

[10] Xuan JF, Monperrus M. Test case purification for improving fault localization. In: Proc. of the FSE. 2014. 52−63.

[11] Zhang LM, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. In: Proc. of the OOPSLA. 2013. 765−784

[12] Papadakis M, Le Traon Y. Using mutants to locate “unknown” faults. In: Proc. of the 5th IEEE Int’l Conf. on Software Testing, Verification and Validation (ICST 2012). IEEE, 2012. 691−700.

[13] Papadakis M, Le Traon Y. Metallaxis-FL: Mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5-7): 605−628.

[14] Chen LS, Pei Y, Furia CA. Contract-based program repair without the contracts. In: Proc. of the 32nd IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE 2017). Urbana-Champaign, Piscataway: IEEE, 2017. 637−647. http://dl.acm.org/citation.cfm?id=3155562.3155642

[15] Dallmeier V, Zeller A, Meyer B. Generating fixes from object behavior anomalies. In: Proc. of the 2009 IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE 2009). Washington: IEEE Computer Society, 2009. 550−554. https://doi.org/10.1109/ASE.2009.15

[16] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. IEEE Trans. on Software Engineering, 2017, 99: 1. https://doi.org/10.1109/TSE.2017.2755013

[17] Gopinath D, Malik MZ, Khurshid S. Specification-based program repair using SAT. In: Proc. of the 17th Int’l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011/ETAPS 2011). Saarbrcken, Berlin, Heidelberg: Springer-Verlag, 2011. 173−188. http://dl.acm.org/citation.cfm?id=1987389.1987408

[18] Long F, Rinard M. Staged program repair with condition synthesis. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). Bergamo, 2015. 166−178. https://doi.org/10.1145/2786805.2786811

[19] Long F, Rinard M. Automatic patch generation by learning correct code. In: Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2016). St. Petersburg, 2016. 298−312. https://doi.org/10.1145/2837614.2837617

[20] Martinez M, Durieux T, Sommerard R, Xuan JF, Monperrus M. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. Empirical Software Engineering, 2017, 22(4): 1936−1964.

[21] Mechtaev S, Yi JY, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proc. of the 38th Int’l Conf. on Software Engineering (ICSE 2016). Austin, 2016. 691−701. https://doi.org/10.1145/2884781.2884807

[22] Monperrus M. Automatic software repair: A bibliography. ACM Computing Surveys, 2018, 51(1): 24. https://doi.org/10.1145/3105906

[23] Nguyen HDT, Qi DW, Roychoudhury A, Chandra S. SemFix: Program repair via semantic analysis. In: Proc. of the 35th Int’l Conf. on Software Engineering (ICSE 2013). San Francisco, 2013. 772−781.

[24] Pei Y, Furia CA, Nordio M, Wei Y, Meyer B, Zeller A. Automated fixing of programs with contracts. IEEE Trans. on Software Engineering, 2014, 40(5): 427−449. https://doi.org/10.1109/TSE.2014.2312918

[25] Weimer W. Patches as better bug reports. In: Proc. of the 5th Int’l Conf. on Generative Programming and Component Engineering (GPCE 2006). New York: ACM, 2006. 181−190. https://doi.org/10.1145/1173706.1173734

[26] Xuan JF, Martinez M, Demarco F, Clement M, Lamelas Marcote SR, Durieux T, Le Berre D, Monperrus M. NOPOL: Automatic repair of conditional statement bugs in Java programs. IEEE Trans. on Software Engineering, 2017, 43(1): 34−55. https://doi.org/10.1109/TSE.2016.2560811

[27] Ghanbari A, Benton S, Zhang LM. Practical program repair via bytecode mutation. In: Zhang DM, Møller A, eds. Proc. of the 28th ACM SIGSOFT Int’l Symp. on Software Testing and Analysis (ISSTA 2019). Beijing: ACM, 2019. 19−30. https://doi.org/10.1145/3293882.3330559

[28] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Tip F, Bodden E, eds. Proc. of the 27th ACM SIGSOFT Int’l Symp. on Software Testing and Analysis (ISSTA 2018). Amsterdam: ACM, 2018. 298−309. https://doi.org/10.1145/3213846.3213871

[29] Wen M, Chen JJ, Wu RX, Hao D, Cheung SC. Context-aware patch generation for better automated program repair. In: Chaudron M, Crnkovic I, Chechik M, Harman M, eds. Proc. of the 40th Int’l Conf. on Software Engineering (ICSE 2018). Gothenburg: ACM, 2018. 1−11.

[30] Rui A, Zoeteweij P, Van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Proc. of the Testing: Academic and Industrial Conf. on Practice and Research Techniques (MUTATION 2007). TAICPART-MUTATION, 2007.

[31] Lou YL, Ghanbari A, Li X, Zhang LM, Hao D, Zhang L. Can automated program repair refine fault localization? arXiv: 1910.01270, 2019. http://arxiv.org/abs/1910.01270

[32] Lou YL, Ghanbari A, Li X, Zhang LM, Zhang HT, Hao D, Zhang L. Can automatd program repair refine fault localization? A unified debugging approach. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2020). Los Angeles, 2020. 12.

[33] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Dwyer MB, Tip F, eds. Proc. of the 20th Int'l Symp. on Software Testing and Analysis (ISSTA 2011). Toronto, ACM, 2011. 199–209. https://doi.org/10.1145/2001420.2001445

[34] Xie XY, Liu ZC, Song S, Chen ZY, Xuan JF, Xu BW. Revisit of automatic debugging via human focus-tracking analysis. In: Dillon LK, Visser W, Williams L, eds. Proc. of the 38th Int'l Conf. on Software Engineering (ICSE 2016). Austin: ACM, 2016. 808–819.

[35] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis (ISSTA 2014). San Jose, New York: ACM, 2014. 437–440. https://doi.org/10.1145/2610384.2628055

[36] Liu K, Wang SW, Koyuncu A, Kim K, Bissyandé TF, Kim DS, Wu P, Klein J, Mao XG, Le Traon Y. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In: Proc. of the 42th Int'l Conf. on Software Engineering (ICSE 2020). 2020.

[37] Marginean A, Bader J, Chandra S, Harman M, Jia Y, Mao K, Mols A, Scott A. SapFix: Automated end-to-end repair at scale. In: Sharp H, Whalen M, eds. Proc. of the 41st Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE (SEIP) 2019). Montreal: IEEE/ACM, 2019. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[38] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the ISSTA. 2019. 169–180.

[39] Zhang MS, Li X, Zhang LM, Khurshid S. Boosting spectrum-based fault localization using PageRank. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2017. 261–272.

[40] Benton S, Li X, Lou YL, Zhang LM. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In: Proc. of the ASE. 2020.

[41] Chib S, Greenberg E. Understanding the metropolis-hastings algorithm. The American Statistician, 1995, 49(4). [doi: 10.2307/2684568]

[42] Metropolis N. Equation of state calculations by fast computing machines. Journal of Chemical Physics, 1953, 21(1953).

[43] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proc. of the ICSE. 2002. 467–477.

[44] Abreu R, Zoeteweij P, Gemund A. On the accuracy of 1155 spectrum-based fault localization. In: Proc. of the Testing: Academic and Industrial Conf. on Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), Vol. 1156. IEEE, 2007. 89–98.

[45] DuyBLe T, Lo D, Goues C, Grunske L. Alearning-to-rank based fault localization approach using likely invariants. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. ACM, 2016. 177–188.

[46] Koyuncu A, Liu K, Bissyandé TF, Kim DS, Monperrus M, Klein J, Le Traon Y. iFixR: Bug report driven program repair. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2019. 314–325.

[47] Zhang XY, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proc. of the 28th Int'l Conf. on Software Engineering. ACM, 2006. 272–281.

[48] Sohn JJ, Yoo S. FLUCCS: Using code and change metrics to improve fault localization. In: Bultan T, Sen K, eds. Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 273–283. https://doi.org/10.1145/3092703.3092717

[49] Li X, Zhang LM. Transforming programs and tests in tandem for fault localization. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): Article 92. https://doi.org/10.1145/3133916

[50] Xuan JF, Monperrus M. Learning to combine multiple ranking metrics for fault localization. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 191–200.

[51] Li Y, Wang SH, Nguyen TN. DLFix: Context-based code transformation learning for automated program repair. In: Proc. of the 42th Int'l Conf. on Software Engineering (ICSE 2020). 2020.

[52] Debroy V, Wong WE. Using mutation to automatically suggest fixes for faulty programs. In: Proc. of the 3rd Int'l Conf. on Software Testing, Verification and Validation. 2010. 65–74. https://doi.org/10.1109/ICST.2010.66

[53] Le Goues C, Vu Nguyen T, Forrest S, Weimer W. GenProg: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2012, 38(1): 54−72. https://doi.org/10.1109/TSE.2011.104

[54] Xu TT, Chen LS, Pei Y, Zhang T, Pan MX, Furia CA. RESTORE: Retrospective fault localization enhancing automated program repair. IEEE Trans. on Software Engineering, 2019.

[55] Papadakis M, Le Traon Y. Effective fault localization via mutation analysis: A selective mutation approach. In: Proc. of the 29th Annual ACM Symp. 2014.

[56] Schkufza E, Sharma R, Aiken A. Stochastic superoptimization. In: Sarkar V, Bodík R, eds. Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013). Houston: ACM, 2013. 305−316. https://doi.org/10.1145/2451116.2451150

[57] Chen JJ, Han JQ, Sun PY, Zhang LM, Hao D, Zhang L. Compiler bug isolation via effective witness test program generation. In: Dumas M, Pfahl D, Apel S, Russo A, eds. Proc. of the ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2019). Tallinn: ACM, 2019. 223−234. https://doi.org/10.1145/3338906.3338957

[58] Chen YT, Su T, Sun CN, Su ZD, Zhao JJ. Coverage-directed differential testing of JVM implementations. In: Krintz C, Berger E, eds. Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2016). Santa Barbara: ACM, 2016. 85−99.

[59] Nicholls EM, Stark AE. Bayes' theorem. Medical Journal of Australia 2, 1971, 26(1971): 1335.

[60] Xie XY, Liu ZC, Song S, Chen ZY, Xuan JF, Xu BW. Revisit of automatic debugging via human focus-tracking analysis. In: Dillon LK, Visser W, Williams L, eds. Proc. of the 38th Int'l Conf. on Software Engineering (ICSE 2016). Austin: ACM, 2016. 808−819.

[61] Xu ZG, Ma SQ, Zhang XY, Zhu SF, Xu BW. Debugging with intelligence via probabilistic inference. In: Chaudron M, Crnkovic I, Chechik M, Harman M, eds. Proc. of the 40th Int'l Conf. on Software Engineering (ICSE 2018). Gothenburg: ACM, 2018. 1171−1181.

[62] Kochhar PS, Xia X, Lo D, Li SP. Practitioners' expectations on automated fault localization. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. ACM, 2016. 165−176.

[63] Zou DM, Liang JJ, Xiong YF, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Trans. on Software Engineering, 2019.

[64] Wikipedia Contributors. Wilcoxon signed-rank test—Wikipedia, the free encyclopedia. 2019. https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.AccessedApril-25-2020

[65] Dunn OJ. Multiple comparisons among means. Journal of the American Statistical Association, 1961, 56(293): 52−64.

娄一翎(1993－), 女, 博士, CCF 学生会员, 主要研究领域为软件工程.

张皓天(1981－), 男, 硕士, CCF 专业会员, 主要研究领域为软件工程, 程序分析, 机器学习.

张令明(1986－), 男, 博士, 助理教授, 博士生导师, 主要研究领域为软件工程, 程序语言.

张路(1973－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件分析, 软件测试, 软件维护与演化.

郝丹(1979－), 女, 博士, 长聘副教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件工程, 软件测试与排错.