软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006924]

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

基于指针神经网络的细粒度缺陷定位*

王尚文[1], 刘 逵[2], 林 博[1], 黎 立[3], KLEINJacques[4], BISSYANDÉ Tegawendé François[4], 毛晓光[1]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](南京航空航天大学 计算机科学与技术学院, 江苏 南京 210016)

[3](Monash University, Clayton VIC 3800, Australia)

[4](University of Luxembourg, rue Richard Coudenhove-Kalergi L-1359, Luxembourg)

通信作者: 刘逵, E-mail: kui.liu@nuaa.edu.cn

摘 要: 软件缺陷定位是指找出与软件失效相关的程序元素. 当前的缺陷定位技术仅能产生函数级或语句级的定位结果. 这种粗粒度的定位结果会影响人工调试程序和软件缺陷自动修复的效率和效果. 专注于细粒度地识别导致软件缺陷的具体代码令牌, 为代码令牌建立抽象语法树路径, 提出基于指针神经网络的细粒度缺陷定位模型来预测出具体的缺陷代码令牌和修复该令牌的具体操作行为. 开源项目中的大量缺陷补丁数据集包含大量可供训练的数据, 且基于抽象语法树构建的路径可以有效捕获程序结构信息. 实验结果表明所训练出的模型能够准确预测缺陷代码令牌并显著优于基于统计的与基于机器学习的基线方法. 另外, 为了验证细粒度的缺陷定位结果可以贡献于缺陷自动修复, 基于细粒度的缺陷定位结果设计两种程序修复流程, 即代码补全工具去预测正确令牌的方法和启发式规则寻找合适代码修复元素的方法, 结果表明两种方法都能有效解决软件缺陷自动修复中的过拟合问题.

关键词: 缺陷定位; 缺陷自动修复; 神经网络

中图法分类号: TP311

中文引用格式: 王尚文, 刘逵, 林博, 黎立, KLEINJacques, BISSYANDÉ Tegawendé François, 毛晓光. 基于指针神经网络的细粒度缺陷定位. 软件学报. http://www.jos.org.cn/1000-9825/6924.htm

英文引用格式: Wang SW, Liu K, Lin B, Li L, Klein J, Bissyandé TF, Mao XG. Fine-grained Defect Localization Based on Pointer Neural Network. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6924.htm

Fine-grained Defect Localization Based on Pointer Neural Network

WANG Shang-Wen[1], LIU Kui[2], LIN Bo[1], LI Li[3], KLEIN Jacques[4], BISSYANDÉ Tegawendé François[4], MAO Xiao-Guang[1]

[1](College of Computer Science, National University of Defense Technology, Changsha 410073, China)

[2](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

[3](Monash University, Clayton VIC 3800, Australia)

[4](University of Luxembourg, rue Richard Coudenhove-Kalergi L-1359, Luxembourg)

Abstract: Software defect localization refers to the activity of finding program elements that are related to software failure. The existing defect localization techniques, however, can only produce localization results at the function or statement level. These coarse-grained localization results can affect the efficiency and effectiveness of manual debugging and automatic software defect repair. This study focuses on the fine-grained identification of specific code tokens that lead to software defects. The study establishes abstract syntax tree paths for code tokens and proposes a fine-grained defect localization model based on a pointer neural network to predict specific code tokens of defects and specific operation behaviors of repairing the tokens. A large number of defect patch data sets in open-source projects contain a large amount of trainable data, and the paths constructed based on abstract syntax trees can effectively capture the program's

* 基金项目: 国家自然科学基金 (62172214); 江苏省自然科学基金 (BK20210279)
收稿时间: 2021-11-16; 修改时间: 2022-06-05, 2022-11-03, 2023-02-15; 采用时间: 2023-03-05; jos 在线出版时间: 2023-08-23

structural information. Experimental results show that the model trained in this study can accurately predict defect code tokens and is significantly better than the baseline methods based on statistics and machine learning. In addition, in order to verify that fine-grained defect localization results can contribute to automatic defect repair, two kinds of program repair processes are designed based on the fine-grained defect localization results. The processes are implemented by using code completion tools to predict the correct token or by following heuristic rules to find appropriate code repair elements. The results show that both methods can effectively solve the overfitting problem in automatic software defect repair.

Key words: defect localization; automatic defect repair; neural network

随着软件规模不断增大, 软件出现缺陷变得不可避免. 软件缺陷定位技术是软件调试活动中非常重要的任务和研究课题. 准确的缺陷定位不仅可以减轻开发人员的工作负担, 而且作为缺陷自动修复技术的上游任务, 其准确性的提升也能够进一步促进缺陷自动修复研究工作的研发[1].

已有研究表明, 现有的缺陷定位技术尚未被工业界的开发人员们广泛使用. 其主要原因是现有定位技术不能够提供细粒度的定位结果[2]. 具体而言, 现有技术主要关注于代码函数或语句级别的缺陷定位, 而开发人员们认为仅提供一条可能出错的代码语句并不能够帮助他们快速准确地理解和修复缺陷. 理论上, 有两种可行的方案能够弥补当前缺陷定位技术的不足: 一种是在提供粗粒度定位结果的同时提供缺陷的精确描述 (例如, 在第 x 行出现了缓冲区溢出的错误); 另一种是提供细粒度的代码令牌 (code token) 级别的缺陷定位结果以及修改该令牌的操作行为 (例如, 更新位于第 x 行第 y 列的布尔值“true”). 本文的工作采用第 2 条路线来弥补现有缺陷定位技术的不足.

缺陷定位为缺陷自动修复提供需要进行修正的代码位置, 然而现有的缺陷定位粒度不能为缺陷自动修复提供很好的支撑. 一方面, 函数和语句级别的定位会使得缺陷修复工具在生成补丁时遇到搜索空间爆炸的问题, 导致修复效率低下[3]; 另一方面, 定位工具的不准确导致修复工具在非缺陷位置处生成测试过拟合的补丁 (即通过测试但不能正确修复缺陷的补丁). 这种补丁带来过拟合的问题[4−8]是缺陷自动修复技术面临的关键瓶颈[9]. 图 1 展示了源自 Defects4J 缺陷数据集[10]的 Closure-62 缺陷的标准补丁. 尽管当前缺陷定位工具能够定位到这条出错的条件语句, 由于不了解该语句中出现错误的位置, 开发人员仍需要调查大量的变更试验来修复该缺陷. 类似地, 一个常规的基于搜索的缺陷自动修复工具需要对该语句中的 12 个代码令牌 (即“excerpt”“equals”“LINE”“&&”“0”“<=”“charno”“&&”“charno”“<”“sourceExcerpt”“length”) 进行多种修复尝试. 在这一过程中, 对非缺陷代码元素进行试验将产生和验证众多无意义的补丁, 而编译与运行补丁通常极为耗时, 这将大大降低自动修复的效率. 此外, 修改非缺陷元素会增加产生过拟合补丁的可能性. 图 1 的下半部分给出了 jKali 工具[11]针对此缺陷生成的过拟合补丁, 它删除了整个条件表达式. 由于真实程序中的测试套件往往不能够完整刻画程序正确行为的规约, 该补丁仍通过了整个测试套件并造成过拟合补丁的产生. 总的来说, 缺陷定位是促进人工调试与缺陷自动修复的关键.

```
1  //   Closure-62的标准补丁
2  -    if (excerpt.equals(LINE) && 0 <= charno && charno < sourceExcerpt.length()) {
3  +    if (excerpt.equals(LINE) && 0 <= charno && charno <= sourceExcerpt.length()) {
4
5  //   由jKali生成的Closure-62的过拟合补丁
6  -    if (excerpt.equals(LINE) && 0 <= charno && charno < sourceExcerpt.length()) {
7  +    if (true) {
```

图 1　缺陷 Closure-62 的标准补丁以及 jKali 对其生成的过拟合补丁

针对上述问题, 本文提出一种基于深度学习的细粒度缺陷定位方法 BEEP (buggy code element predictor). 该方法意在准确定位缺陷程序中需要被修改的导致缺陷的代码令牌, 因此能够较当前的缺陷定位方法 (至多提供语句级别的定位结果) 提供更加细粒度的定位结果. 我们方法的基本假设与大多数基于机器学习的缺陷预测技术[12]的共同假设相似, 即, 具有相似程序结构的缺陷程序往往涉及相同的错误代码元素 (令牌). 因此, 我们采用了抽象语法树路径 (AST path) 的技术, 该技术已被证明能很好地学习到程序的语法结构特征[13−16]. 我们方法的工作流程如图 2 所示. 其总体上分为两个阶段, 即模型训练和令牌预测. 在训练阶段, 首先进行数据的预处理: 我们对大量的补丁数据进行分析, 提取其抽象语法树层面的代码更改, 针对每一个缺陷程序, 构建一组包含抽象语法树路径和修改

操作 (即更换 UPDATE、添加 INSERT、删除 DELETE) 的“操作路径”. 该组路径中含有一条或多条表示该补丁中代码更改的“标准操作路径”. 模型中采用了指针神经网络 (pointer network)[17], 其功能为从众多输入的操作路径中预测出标准操作路径, 该路径中抽象语法树路径指向的代码令牌为缺陷代码元素, 对应的修改操作为修复该缺陷代码元素所需的更改操作. 当应用于预测时, BEEP 的输入是一个缺陷函数, 其通过建立缺陷代码的抽象语法树, 提取抽象语法树路径并构建操作路径, 将一组操作路径送入已训练好的模型中, 返回结果即是细粒度的缺陷定位结果. 多组对比实验验证了本文所提出的方法在细粒度缺陷定位问题上的有效性. 此外, 基于细粒度缺陷定位结果, 我们设计了两种缺陷修复流程, 一种利用现成的代码补全工具来预测可以替换被 BEEP 识别出的缺陷代码令牌的正确内容; 另一种使用简单的启发式方法来搜索出可以替换被 BEEP 方法识别出的缺陷代码令牌的内容. 实验结果表明两种方法都能实现百分之百的正确率 (生成的通过所有测试套件的补丁都与开发人员提供的标准补丁在语义上等价, 即语义正确的) 并具有较高的修复效率.

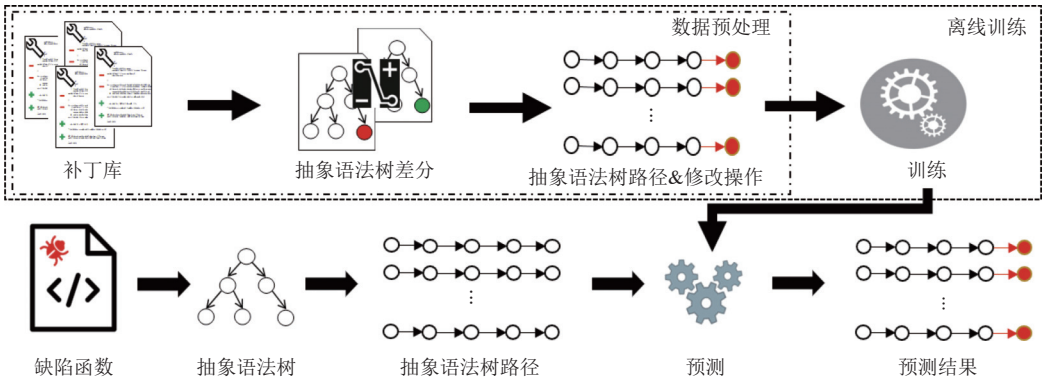


图 2 BEEP 细粒度缺陷定位技术工作流程图

本文第 1 节介绍了缺陷定位与程序修复的相关方法和研究现状. 第 2 节阐述了本文所需的背景知识, 即抽象语法树路径. 第 3 节介绍了本文构建的基于指针神经网络的细粒度缺陷定位模型. 第 4 节通过对比实验验证了所提模型的有效性. 第 5 节探讨了基于细粒度缺陷定位的程序修复的有效性与效率. 最后第 6 节对全文进行了总结.

1 缺陷定位与程序修复相关工作

本文所涉及内容主要与缺陷定位与程序修复有关, 下面就相关研究现状给予介绍.

1.1 缺陷定位

缺陷定位旨在精确诊断缺陷位置以方便程序调试[18]. 在众多缺陷定位技术中, 研究最广泛的是基于频谱的缺陷定位技术 (spectrum based fault localization, SBFL)[19]. 其通常利用成功和失败测试用例的执行轨迹来识别可疑代码元素 (例如代码行或方法). 其背后的机理是, 如果一个代码元素被更多的失败测试覆盖但却被较少的通过测试覆盖, 那么它有缺陷的可能性就偏大. 因此, 研究人员通常应用统计分析 (例如 Ochiai[20]和 Tarantula[21]) 来计算代码元素为缺陷可能性, 并按可能性大小的降序排列它们以表示缺陷定位结果. SBFL 方法的固有限制是执行了失败测试用例的代码元素不一定表明该元素与测试失败有关. 为了解决这个问题, 研究人员还提出了基于变异的缺陷定位 (mutation based fault localization, MBFL) 技术[22], 该技术对代码元素进行变异, 然后检查其对测试结果的影响. 除了 SBFL 和 MBFL, 研究人员还探索了各种缺陷定位技术, 例如基于代码切片[23]、基于统计分析[24]、基于程序状态[25]、基于学习[26]、基于数据挖掘[27]和基于模型[28]的技术. 也有研究人员提出利用程序修复阶段产生的补丁信息来优化缺陷定位结果[1,29]: 如果在识别到的缺陷位置处能生成通过所有测试用例的补丁, 则说明该定位结果是准确的; 反之则说明该定位结果不够准确. 在近期的缺陷定位相关研究中, 研究人员试图融入更先进的机器学习技术以获取更准确的定位结果. 例如, Zou 等人[30]利用排序学习 (learning to rank) 的方法将上文中提到的不同

类型缺陷定位方法的定位结果进行融合, 得到更为准确的结果; Lou 等人[31]设计了面向缺陷定位的图结构程序表示方法, 在抽象语法树基础上引入测试用例与程序语句间的覆盖关系, 并用图神经网络 (gated graph neural network) 进行特征学习并对程序语句进行排序; 为解决正例与负例 (即通过的测试用例与失效的测试用例) 数量不均衡的问题, Xie 等人[32]尝试通过自编码器 (auto-encoder) 自动生成合理的负例样本, 提升基于机器学习的缺陷定位方法的定位效果. 也有学者提出[33], 可以利用变量的运行时信息优化对包含数字、字符等类型的变量的赋值语句可疑度的计算过程.

尽管缺陷定位技术有较长的研究历史, 最先进的定位技术也仅能在代码语句的粒度上定位缺陷的位置. 因此在一些针对开发人员的调查显示, 现有的缺陷定位技术并没有受到他们的欢迎. Parnin 等人[2]发现自动调试技术做出的一些假设 (例如, “对开发人员来说, 检查一条独立的错误语句就足够了”) 对开发人员的实践来说并不成立. Xie 等人[34]发现简单地为开发人员提供可疑代码语句的排名实际上会降低他们的调试效率. Kochhar 等人[35]发现迄今为止最流行的缺陷定位粒度 (即方法级别) 仅获得了大约一半的被调查者的认可. 我们的工作针对缺陷语句中的导致缺陷的代码令牌进行识别, 从而为程序调试提供更细粒度的缺陷定位结果.

1.2 缺陷自动修复

缺陷自动修复旨在将补丁生成的过程自动化, 使开发人员摆脱手动调试的沉重负担. 缺陷自动修复流程以缺陷定位为起始步骤, 在获取到可能的缺陷位置后尝试不同的补丁生成策略生成并验证补丁, 最终输出能够通过验证的补丁. Liu 等人[36]指出, 现有缺陷自动修复工具的修复能力在一定程度上受到了缺陷定位工具的制约. 因此, 研究人员提出利用不同的方法对缺陷定位结果进行优化. 例如, SimFix[37]应用测试用例提纯的方法将一个测试用例拆分为多个测试用例从而更加充分地暴露缺陷位置.

长久以来, 缺陷自动修复领域的研究人员们探索了多种补丁生成的方法. 作为缺陷自动修复领域的里程碑, GenProg[38]借助于基因编程的思想, 通过变异生成一代又一代种子程序, 利用种子程序成功运行测试用例的数量衡量种子质量, 直至变异出能够成功通过所有测试用例的补丁. 在此基础上涌现出了各种缺陷自动修复技术, 根据补丁生成方法的不同, 大致可以分为 4 类: 基于启发式、基于约束、基于修复模板和基于学习的修复方法. 基于启发式的修复方法构建满足语法规约的程序搜索空间并在其中对可能的补丁进行搜索, 代表工作有利用相似代码对搜索空间进行限制的 SimFix[37]以及为修复动作与缺陷修复历史数据相匹配的补丁赋予更高优先级的 HDRepair[39]. 基于约束的修复方法通常利用特定技术 (例如约束求解) 来推断给定缺陷程序的约束, 这些约束被进一步用于指导补丁的生成和验证, 代表工作有利用测试用例信息刻画约束条件的 SemFix[40]以及利用轻量级约束条件显著提升此类方法的可扩展性的 Angelix[41]. 基于修复模板的修复方法总结从现实世界的补丁中识别出的修复模板, 这些模板可以为缺陷修复的代码更改行为提供具体的指导, 辅助机器生成与现实世界中的补丁更为相似的补丁, 代表工作包括融合了 35 种代码变更模式的 TBar[42]和利用静态分析工具总结可靠的、多样的修复模式的 AVATAR[43]. 近几年来, 也有部分工作关注于用学习的方法生成补丁, 将补丁生成的过程视为从缺陷代码到正确代码的转换过程. 例如, CoCoNut[44]提出根据上下文信息采用集成学习的方式将缺陷语句翻译为正确语句; CURE[45]提出在解码器一段增加约束条件, 提高翻译出正确语句的概率; Allamanis 等人[46]使用图神经网络解决程序中变量误用 (variable-misuse bugs) 的问题; Bhatia 等人[47]关注于小规模学生程序中的语法错误问题; Codit[48]使用基于抽象语法树的模型以便捕获代码中的结构信息并保证生成的补丁的语法正确性; Recoder[49]通过预测对代码的编辑操作而非直接生成补丁代码使模型对小规模的代码变换更加有效, 并通过生成占位符使补丁包含项目专有的标识符, 有效解决训练集中词汇表的局限问题.

尽管程序修复技术已被广泛研究, 其仍旧面临一个重要的问题: 补丁过拟合[4−8]. 具体而言, 现有技术无法保证工具生成的通过整个测试套件的补丁是语义正确的. ACS[50]针对条件语句运用变量依赖分析、自然语言处理、谓词选择等技术, 实现了 78% 的准确率; CapGen[51]将缺陷代码的上下文信息纳入考虑, 计算缺陷位置处代码与项目其他地方代码的相似度来选取可用于修复的原料, 实现了 84% 的准确率. 然而, 它们的效果离理想的准确度还有一定距离. 我们的研究通过为缺陷自动修复提供细粒度的缺陷定位信息, 在两种不同的修复策略上均实现了百分之百的准确率, 展示了细粒度缺陷定位可行性.

2 背景知识

本文所提出的方法主要建立在基于抽象语法树路径的代码表示技术之上. 该技术将程序代码转化为抽象语法树, 并在树结构上提取从根节点到叶节点的路径. 在该路径上的任意一对相邻节点都在树结构上具有父节点-子节点的相对关系, 因此所提取到的抽象语法树路径能够有效地对程序结构特征进行建模[12,15].

定义 1. 抽象语法树. 代码的抽象语法树定义为一个六元组: $<N,L,T,r,\Delta,W>$. 其中, N 是一组非叶子节点, L 是一组叶子节点, T 是一组叶子节点对应的代码令牌, $r\in N$ 代表根节点, $\delta\in\Delta: n\rightarrow n'$, $n\in N$, $n'\in(N\cup T)$ 是反映两个抽象语法树节点 n 与 n' 之间父节点-子节点关系的函数, $\omega\in W: l\rightarrow t$, $l\in L$, $t\in T$ 建立起每个叶子节点 l 与其相对应的代码令牌 t 间的对应关系.

定义 2. 抽象语法树路径. 一条抽象语法树路径是一条从根节点 r 出发到一个叶子节点 l 为止的路径. 路径上的任意两个相邻节点间都满足父节点-子节点的关系, 其被定义为一个四元组: $p=<r,l,N',\Delta'>$, 其中 $l\in L$, $N'\subset N$, $\Delta'\subset\Delta$.

定义 3. 操作路径. 一条操作路径是在一条抽象语法树路径的基础上加上代码令牌的变换操作, 被定义为一个三元组: $op=<t,p,o>$, 其中 t 是抽象语法树路径 p 上叶子节点 l 所对应的代码令牌, $o\in$ {UPDATE, DELETE, INSERT} 是一种代码变换原子操作, 其含义是修复该缺陷需要对代码令牌 t 进行何种形式的代码变换.

图 3 展示了图 1 中正确补丁的抽象语法树以及其对应的代码变换操作. MethodDeclaration 是这棵抽象语法树的根节点, 其他具有灰色背景的非叶子节点, 而所有椭圆形透明背景的节点是叶子节点. 每个叶子节点对应的代码令牌展现在长方形中. 为了便于展示, 其他与缺陷代码元素无关的节点没有在图中完全展示出来. 对于缺陷代码令牌"<", 其相应的抽象语法树路径 p 为: "MethodDeclaration→Block→IfStatement→IfStatement→InfixExpression→InfixExpression→InfixExpression→Operator", 即图 3 中用红色虚线标出的路径. 修复这个缺陷的补丁将"<"替换成了"<=", 因此其操作路径为 op=<"<", p, UPDATE>.

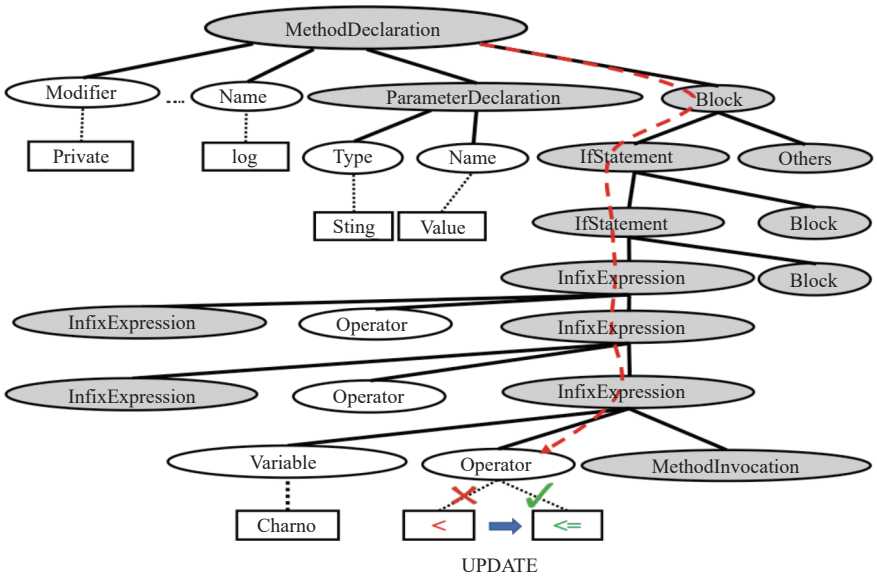


图 3 图 1 中正确补丁的抽象语法树与变换操作

3 基于指针神经网络的细粒度缺陷定位方法

鉴于神经网络强大的特征提取能力, 本文提出一种基于指针神经网络的细粒度缺陷定位方法, 该方法主要由训练阶段和预测阶段组成.

在训练阶段, 我们先对大量的补丁数据进行预处理. 针对每一个补丁, 我们利用抽象语法树差分技术分析其代

码变换的内容, 提取此补丁的标准操作路径, 并人为地构建缺陷程序包含的其他操作路径. 在训练时, 人为构建得到的操作路径为噪音信息, 而通过抽象语法树差分技术得到的标准操作路径为神经网络的预期输出. 在完成数据预处理后, 开始对模型进行训练. 我们首先对每条输入的操作路径进行编码, 将路径中的信息映射到高维数字向量空间. 随后我们采用指针神经网络从多个特征中筛选出所需的特征, 帮助我们从众多输入的操作路径中筛选出标准操作路径, 即修复该缺陷所对应的操作路径. 从该路径中我们便可获取到细粒度的缺陷代码令牌信息以及对其进行修复所需操作.

第 2 阶段为预测阶段, 对于给定的缺陷函数, 我们将其转化为对应的抽象语法树并抽取出所有的抽象语法树路径. 随后人为地将每条抽象语法树路径与 3 种代码更改原子操作结合, 构建出该缺陷代码片段所有可能的操作路径. 我们将得到的一组操作路径送入上一阶段已训练好的模型当中, 输出按照可能性从高到低排列的操作路径序列, 也即 BEEP 的预测结果. 通过解析操作路径中包含的代码令牌和变换操作, 可以得知缺陷函数可能的缺陷令牌及修复该缺陷所需的代码变换操作.

3.1 数据预处理

我们使用历史补丁作为训练数据集. 对于数据集中的每个补丁, 我们使用 GumTree[52]抽象语法树差分技术来计算缺陷程序与正确补丁之间在抽象语法树层面的代码变更差异. 通过得到的差异, 我们可以识别到抽象语法树路径 (即从根节点到被修改的叶子节点的路径) 以及操作路径 (即抽象语法树路径、代码令牌和代码变换操作). 需要注意的是, 如果补丁中代码变换操作为插入, 则对应的操作路径内容应包括插入位置前一处未被更改的代码令牌 (即前一条未被更改的语句的最后一个代码令牌)、其相应的抽象语法树路径以及 INSERT 操作. 通过抽象语法树差分得到的操作路径仅是这个补丁所对应的正样例. 我们考虑每个缺陷方法中所有指向非缺陷叶子节点的抽象语法树路径, 交替为其填补上 3 个更改操作中的每一个, 使一条抽象语法树路径衍生出 3 条操作路径, 这些新得到的操作路径在训练过程中被标记为负样例, 因为它们代表着不应被预测为有缺陷的路径.

此外, 在数据预处理中, 我们收集每条抽象语法树叶子节点所对应的代码令牌信息, 它们在模型训练中能够一定程度上提供程序语义信息. 一些近期的研究工作表明, 将代码令牌拆分成代码子令牌 (code sub-tokens) 能够显著降低模型中词表的大小并更加有效地捕获语义信息[13,44], 因此我们也进行了类似的操作. 具体而言, 代码令牌依据驼峰命名法和下划线命名规约[53,54]被拆分为代码子令牌, 得到的代码子令牌被进一步转化为小写形式.

3.2 模型训练

图 4 展示了我们提出的 BEEP 训练-预测模型的整体结构. 我们的深度神经网络主要由编码器-解码器结构和指针网络组成. 编码器-解码器结构已经大大提升了解决自然语言中出现的语法错误的能力[55,56], 同时受软件自然性[57]的启发, 程序中的缺陷部分也应该能够利用大数据的深度学习检测出来. 指针网络是注意力模型的一个简单变种, 给定一条输入的序列, 序列中各元素是离散的 (即不存在顺序关系), 指针网络能够学习到输出序列的概率条件. 近期研究表明, 当输出的元素是从输入的元素中进行选择时, 指针网络具有非常好的效果[58,59]. 在定位细粒度的缺陷代码令牌的场景下, 预测结果 (即操作路径) 是离散元素, 因此我们采用指针网络作为模型预测缺陷代码令牌及相关的代码变换操作的框架.

• 操作路径编码. 对于一条操作路径 $op_i =< t_i, p_i, o_i >$, 其中 $p_i = \{n_1^i, n_2^i, \ldots, n_{l_i}^i\}$ 是对应的抽象语法树路径, t_i 是代码令牌, o_i 是代码令牌修改操作. 我们首先对其中的代码子令牌、抽象语法树路径以及代码变换操作进行编码, 这一过程被已有的代码表示技术 (如 code2seq[13]) 广泛采用. 我们利用不同编码矩阵分别对拆分后的代码子令牌和修改操作进行编码, 这一过程可以表示为:

$$V_t = \sum_{t_s \in T_s} E_t(t_s),$$

$$V_o = E_o(o),$$

其中, $E_t(\cdot)$ 和 $E_o(\cdot)$ 分别代表针对子令牌和修改操作的不同编码矩阵. T_s 是代码令牌 t 的子令牌流, V_t 代表令牌 t 的向量表示, V_o 代表更改操作 o 的向量表示.

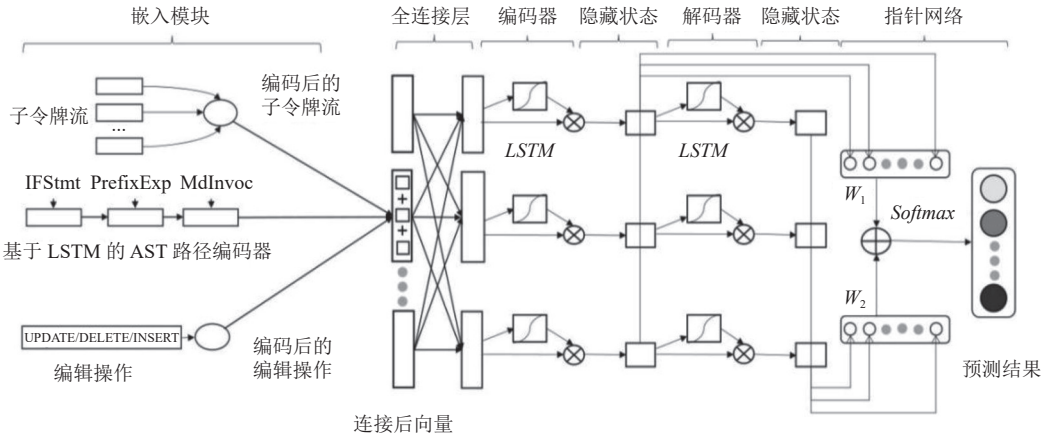


图 4 BEEP 模型结构图

抽象语法树路径由多个抽象语法树节点组成. 在对抽象语法树路径进行编码时, 我们先使用了一个编码矩阵对每一个节点进行编码, 随后为了捕获抽象语法树路径中相邻节点所蕴含的结构特征, 我们需要利用路径中的序列信息. 因此我们使用双向长短期记忆网络 (BiLSTM) 对整个路径进行编码, 神经网络最终的输出为对抽象语法树路径进行编码的结果. 该过程可表示为:

$$V_p = LSTM\left(E_p(n_1), E_p(n_2), \ldots, E_p(n_l)\right),$$

其中, $E_p(\cdot)$ 是用于编码节点的嵌入矩阵, *LSTM* 代表我们采用的双向 *LSTM* 模型, V_p 代表抽象语法树路径 p 最终的向量表示.

在得到了操作路径 op 中 3 部分的向量表示后, 我们将 3 部分的向量表示连接在一起作为该操作路径的向量表示: $V_{op} = \left[V_t; V_p; V_o\right]$.

• 利用编码器-解码器网络提取特征. 假定输入的操作路径共有 k 条, 即 $\{op_1, op_2, \ldots, op_k\}$, 在对每条操作路径进行编码后, 为了提取更多的隐藏信息, 我们将其输入一个全连接层, 随后再利用基于 *LSTM* 的编码器-解码器结构捕获特征, 这一过程可被表示为:

$$\begin{aligned} z_i &= \tanh\left(W_{\text{in}}\left[V_t; V_p; V_o\right]\right), \\ (e_1, \ldots, e_k) &= LSTM_{\text{encoder}}(z_1, \ldots, z_k), \\ (b_1, \ldots, b_k) &= LSTM_{\text{decoder}}(e_1, \ldots, e_k), \end{aligned}$$

其中, W_{in} 是维度为 $\left(d_t + 2d_p + d_o\right) \times d_{\text{hidden}}$ 的全连接层权重矩阵, $(e_1, \ldots, e_k)$ 与 $(b_1, \ldots, b_k)$ 分别是编码器与解码器的隐藏层状态.

• 利用指针网络获得输出. 在得到了编码器隐藏层状态 $(e_1, \ldots, e_k)$ 与解码器隐藏层状态 $(b_1, \ldots, b_k)$ 后, 我们能够利用下式计算注意力向量:

$$u_j = v^{\mathrm{T}} \tanh\left(W_1 e_j + W_2 b_j\right),$$

其中, $j \in (1, \ldots, k)$, v, W_1, W_2 均为在模型训练过程中需要学习的参数. u_j 的数值即被用作第 j 个输入的注意力权重:

$$p\left(op_j \mid op_1, \ldots, op_k\right) = Softmax\left(u_j\right).$$

Softmax 函数将向量 $u = [u_1, \ldots, u_k]$ 归一化为输入的操作路径的概率分布. 模型最终的输出为依据分布权重由高到低排列的操作路径列表.

• 模型训练. 我们使用交叉熵损失函数[60]对模型进行训练:

$$Loss = \sum_{y_i \in Y} - \sum_{op \in y_i} \left[Y_{op} \cdot \log\left(P_{op}\right) + \left(1 - Y_{op}\right) \cdot \log\left(1 - P_{op}\right)\right],$$

其中, $Y_{op}=\{1,0\}$ 表示该条操作路径是否为正例, P_{op} 是模型输出的该条操作路径的权重. 在模型训练过程中, 我们使用 Adam 方法来降低上述式子中的 *Loss* 值.

3.3 在线预测

当模型经过训练后, 就可以用来预测给定缺陷函数的操作路径. 首先在缺陷函数中提取所有指向叶子节点的抽象语法树路径, 并为每条抽象语法树路径交替分配 3 个操作运算符中的每一个, 从而为该缺陷方法生成所有可能的操作路径. 随后根据第 3.2 节中介绍的操作路径编码方法为每条操作路径获取向量表示形式, 在将这些操作路径送入训练好的模型后, 模型即可返回预测结果 (即按照权重排序的操作路径列表), 对返回的预测结果进行解析即可获取预测到的缺陷令牌、缺陷位置以及修复该缺陷所需进行的代码修改操作信息.

4 实验分析

本节首先介绍了着重研究的 3 个问题、实验所用的数据集和评价标准, 然后设计实验对本文的方法进行验证并对实验结果进行分析讨论.

4.1 研究问题

为了评估基于指针神经网络的细粒度缺陷定位方法的有效性, 我们研究了以下 3 个问题.

- RQ1: BEEP 能否有效识别出真实程序中的缺陷代码令牌?
- RQ2: BEEP 能否准确地同时预测缺陷代码令牌及修复缺陷代码令牌所需进行的代码修改操作?
- RQ3: 不同的模块对 BEEP 方法的有效性产生怎样的影响?

4.2 实验数据集

为了训练我们提出的神经网络模型, 我们收集了来自 CoCoNut[44]工具训练集中的补丁; 为了评估我们模型的效果, 我们收集了来自 ManySStuBs4J[61]和 Defects4J[10]数据集中的补丁. 这些数据集被我们的研究所采用的原因在于: (1) CoCoNut 和 ManySStuBs4J 都是大规模补丁数据集, 用它们可以很好地检验我们模型的泛化能力; (2) Defects4J 数据集是软件缺陷自动修复领域最常用的基准数据集. 我们对数据集中的案例进行了去重, 并且由于我们的研究关注于源代码中的缺陷定位, 我们移除了跟测试代码相关的补丁. 最终, 我们选用的 3 个数据集分别包含 436 676, 26 406, 以及 393 个缺陷.

4.3 评价指标

在评估方法的定位效果时, 我们使用两个指标进行度量.

recall@Top-n 是在缺陷定位中常常被使用的评价指标[30,62,63], 它衡量的是标准答案在预测列表中排在前 n 个的普遍程度, 其计算方式为: $recall@Top\text{-}n=\dfrac{\text{缺陷令牌出现在预测结果}Top\text{-}n\text{个中的缺陷数}}{\text{缺陷总数}}$. n 越小代表定位效果越精确, *recall@Top-n* 值越大代表缺陷定位的效果越好. 在我们的实验中, 我们选取 n 的值为 1, 3, 5, 10, 20. 在我们的评估数据集中, 一个缺陷可能拥有多个缺陷代码令牌 (例如, 一个补丁更改了两个或者多个代码令牌的情况). 依据基于谱的缺陷定位技术中的处理方式[1], 我们认为只要其中任意一个缺陷代码令牌的排名位于 *Top-n* 内, 这个缺陷即被成功定位于 *Top-n* 内.

Mean first rank (*MFR*) 用于计算预测出的可疑代码元素排名列表中第一个缺陷代码元素的平均排名值[1], 其计算方式为: $MFR=\dfrac{\sum_{i=1}^{m}Rank(i)}{m}$, 其中, m 表示缺陷总数, $Rank(i)$ 表示第 i 个缺陷中缺陷令牌在预测结果中的最高排名. *MFR* 的值越低, 表明缺陷定位的精度越高.

4.4 实验设置

在本节中我们介绍实验设置的 3 方面: 一是我们的方法是如何实现并微调的, 二是我们是如何实现基准方法与我们的方法进行比较的, 三是我们评估实验的设计.

• 方法实现与微调. 我们的模型是在 PyTorch 版本的指针网络 (https://github.com/shirgur/PointerNet) 以及 code2seq 技术的代码框架 (https://github.com/tech-srl/code2seq) 上实现的. 代码已发布在 http://doi.org/10.5281/zenodo.4717352. 模型的训练在两台拥有 1080Ti GPU 配置的服务器上进行. 作为神经网络的模型, 我们对方法在应用前进行了微调. 在微调过程中, 我们从 CoCoNut 数据集中随机选取 43 000 个补丁用作验证集, 其余补丁则用于训练与测试. 我们关注于以下几个模型的超参数: 用于表征子令牌的向量长度 (64, 128, 256); 学习率 (0.001, 0.002, 0.005); 训练轮数 (20, 40, 50); 批量大小 (64, 128, 256); 抽象语法树路径最大长度 (10, 15, 20); 以及输入模型的操作路径最大数量 (100, 120, 150, 180, 200). 我们将列出的超参数值进行组合并试验了每种组合的效果, 以在验证集上损失值最小的数值组合作为最佳的超参数配置, 模型最终的超参数值在表 1 中列出.

表 1　模型超参数取值

子令牌向量长度	学习率	训练轮数	批量大小	路径最大长度	路径最大数量
128	0.001	40	256	15	120

• 基准方法设计. 我们的方法是第一个关注于代码令牌级别缺陷定位的, 因此没有现有的方法与之进行比较. 我们提出了两个基准方法与我们的方法进行比较.

第 1 个基准是基于统计的方法. 在 Liu 等人[64]关于真实程序中补丁的分析中, 他们发现一些特定的代码元素相较于其他代码元素更倾向于存在缺陷. 我们提出利用这一研究公布出的不同类别的代码令牌出现缺陷的频率建立一个简单的基准方法, 即依据代码令牌出现缺陷的频率由高到低对令牌进行排序. 对于相同类型的代码令牌, 它们的顺序由它们在缺陷方法的代码令牌流中出现的顺序决定.

第 2 个基准是基于机器学习的方法. 对于缺陷方法中的代码令牌, 我们提取 4 种特征, 即该令牌在令牌流中的排序、该令牌所在语句的类型 (例如, 返回语句或条件语句)、该令牌所包含的字符数以及该令牌所包含的子令牌数. 我们假设缺陷令牌与正常令牌可能在这些特征上存在区别, 因此我们训练了随机森林模型[65]并依据这些特征预测缺陷令牌.

• 评估实验. 为了回答本文提出的研究问题, 我们不仅将 CoCoNut 数据集上训练好的模型在 ManySStuBs4J 和 Defects4J 数据集上进行测评, 同时也在 CoCoNut 数据集上进行了十折交叉验证, 这是因为 CoCoNut 数据集包含了远多于其他两个数据集的补丁, 在该数据集上进行实验评估有助于衡量我们方法的普适性. 我们模型的微调过程已在前文中介绍, 而基准方法中随机森林模型的超参数则复用于机器学习开源库 sklearn (https://scikit-learn.org/stable/). 我们的实验是在内存为 64 GB、GPU 为 1660 Ti 的服务器上进行的. 在 CoCoNut 数据集上训练一个 epoch 的用时约为 8 min, 因此, 十折交叉验证中的每一折大约花费 5 h.

我们的实验考虑了两种场景: 提供给每种方法整个缺陷方法以及在缺陷方法的基础上定位到了具体的缺陷语句. 这两种条件都能被当前基于程序谱的缺陷定位方法所提供. 在进行第 2 个场景的实验时, 我们忽视缺陷方法中在缺陷语句之外的代码令牌的可疑值, 仅对缺陷语句中包含的代码令牌进行排序.

• 标准答案的获取. 针对每一个缺陷程序, 我们使用 GumTree 抽象语法树差分技术来计算缺陷程序与正确补丁之间在抽象语法树层面的代码变更差异. 通过得到的差异, 我们可以识别到抽象语法树路径 (即从根节点到被修改的叶子节点的路径) 以及操作路径 (即抽象语法树路径、该叶子节点对应的代码令牌和代码变换操作). 通过抽象语法树差分得到的操作路径即被视为标准答案, 也即需要被预测出来的路径, 这些路径中对应的代码令牌即为需要预测的缺陷代码令牌.

4.5 实验结果与分析

4.5.1 RQ1: BEEP 能否有效识别真实程序中的缺陷代码令牌?

为了验证这个问题的结果, 我们将 BEEP 方法与两种基准方法在不同粒度的输入下 (即输入为缺陷语句与输入为缺陷方法) 进行了对比实验, 在解析 BEEP 返回的结果时, 我们仅关注操作路径中的代码令牌信息 (即忽略操作路径中的代码变换操作信息, 如果一条操作路径中的代码令牌与实际缺陷令牌一致即被视为是正确的预测). 实验的结果见表 2 与表 3.

表 2 缺陷语句作为输入时的定位效果

数据集	方法	*Top*-1 (%)	*Top*-3 (%)	*Top*-5 (%)	*MFR*
CoCoNut	基准方法#1	26.1	65.9	86.7	3.2
	基准方法#2	61.7	76.5	89.9	2.7
	BEEP	**63.8**	**83.9**	**91.2**	**2.3**
ManySStuBs4J	基准方法#1	36.7	84.1	94.0	2.5
	基准方法#2	36.9	86.3	97.2	1.7
	BEEP	**85.2**	**96.3**	**100**	**1.3**
Defects4J	基准方法#1	18.2	49.4	83.1	2.8
	基准方法#2	23.9	61.8	86.7	1.9
	BEEP	**86.7**	**91.4**	**96.8**	**1.4**

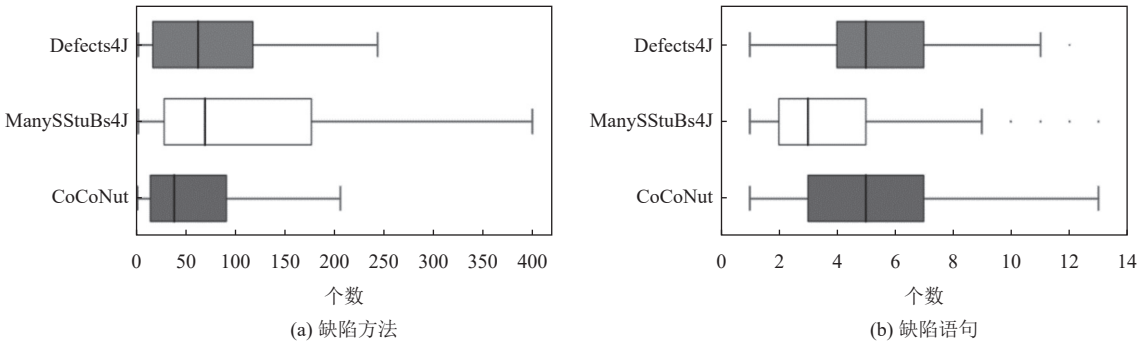
表 3 缺陷方法作为输入时的定位效果

数据集	方法	*Top*-1 (%)	*Top*-5 (%)	*Top*-10 (%)	*Top*-20 (%)	*MFR*
CoCoNut	基准方法#1	3.7	25.3	40.9	57.9	38.7
	基准方法#2	14.0	31.9	41.9	48.3	32.1
	BEEP	**46.9**	**74.2**	**85.5**	**95.2**	**3.9**
ManySStuBs4J	基准方法#1	0.7	10.4	22.3	40.5	42.0
	基准方法#2	8.1	20.9	33.6	46.2	28.9
	BEEP	**30.7**	**56.4**	**72.6**	**90.1**	**6.7**
Defects4J	基准方法#1	1.3	13.3	25.3	45.3	65.1
	基准方法#2	3.0	23.9	29.8	31.3	32.1
	BEEP	**34.9**	**57.1**	**68.2**	**87.3**	**6.5**

我们注意到基准方法#1 的表现与 Liu 等人的观察较为一致, 能够提供较为有效的定位结果, 基准方法#2 的表现在大体上也要优于基准方法#1. 然而, 在 3 个数据集上, BEEP 在所有评价指标下的表现都要优于两个基准方法. 当输入是缺陷方法时, 对于 46.9%/30.7%/34.9% 的来自于 CoCoNut/ManySStuBs4J/Defects4J 数据集的补丁, BEEP 能够准确地将缺陷代码令牌排在第 1 位. 与之对应的是, 在这 3 个数据集上, 基准方法#1 仅能分别将 3.7%/0.7%/1.3% 的缺陷代码令牌排在第 1 位; 基准方法#2 仅能分别将 14.0%/8.1%/3.0% 的缺陷代码令牌排在第 1 位. 此外, 我们也注意到, BEEP 在绝大多数缺陷程序中 (超过 90%) 能够将缺陷代码令牌排在输出结果的前 20 位. 特别是, 对于 ManySStuBs4J 数据集中的所有缺陷, 当输入为缺陷语句时, BEEP 都能将缺陷令牌排在输出结果的前 5 位 (表 2). BEEP 的 *MFR* 值较基准方法领先了一个数量级. 例如, 在 Defects4J 数据集中, BEEP 的 *MFR* 值为 6.5, 而基准方法#1 的 *MFR* 值为 65.1. 当输入是缺陷语句时, 我们也可以观察到相似的现象. 值得注意的是, 在此条件的输入下, BEEP 能够将 ManySStuBs4J 数据集中所有缺陷程序的缺陷代码令牌排在输出列表的前 5 位.

综上所述, BEEP 能够有效预测缺陷程序中的缺陷令牌, 并显著优于一种基于统计的基准方法与一种基于机器学习的基准方法.

为了探究使用 BEEP 进行细粒度的缺陷定位能为开发人员节省多少人工调试的开销, 我们进一步调查了 3 个数据集中缺陷方法与缺陷语句中所含代码令牌的数量. 在进行数据统计时我们对于 Java 语言的关键词 (例如: if, int 等) 不作考虑. 图 5 展示了统计结果.



(a) 缺陷方法 (b) 缺陷语句

图 5 每个缺陷方法与缺陷语句中的代码令牌数

对于 CoCoNut/ManySStuBs4J/Defects4J 数据集来说, 每个缺陷方法中所包含的代码令牌数的中位数为 38/72/67. 根据表 3 中的数据, BEEP 在 3 个数据集上的 *MFR* 值分别为 3.9/6.7/6.5. 这一结果说明缺陷代码令牌在 BEEP 预测结果中很靠前的位置即可被找到 (通常位于所有代码令牌的前 10%). 对于每行缺陷语句中的代码令牌

数, 3 个数据集上的中位数分别为 5/3/5. 根据表 2 的结果, BEEP 在 3 个数据集上的 *MFR* 分别为 2.3/1.3/1.4. 这意味着通常情况下 BEEP 能将缺陷元素排在其输出列表的前半部分.

综上所述, 在提供缺陷方法或是缺陷语句的情况下, BEEP 的预测结果都能够将缺陷代码令牌排在靠前的位置.

4.5.2 RQ2: BEEP 能否准确地同时预测缺陷代码令牌及修复缺陷代码令牌所需进行的代码变换操作?

为了回答这个问题, 我们分析 BEEP 方法返回的操作路径信息并计算每个缺陷的标准操作路径在返回列表中的排名情况, 结果见表 4.

表 4 缺陷方法作为输入时 BEEP 的操作路径预测效果

数据集	*Top*-1 (%)	*Top*-5 (%)	*Top*-10 (%)	*Top*-20 (%)	*MFR*
CoCoNut	44.6	67.8	77.9	87.5	7.5
ManySStuBs4J	29.7	50.8	63.7	79.3	11.5
Defects4J	29.2	51.8	57.7	65.5	12.3

我们注意到, 与单纯的缺陷代码令牌预测结果 (表 3) 相比, 考虑整个操作路径的内容后, BEEP 的预测结果仅仅小幅度下降. 例如, 在 ManySStuBs4J 数据集上 *Top*-1 的召回率仅由 30.7% 下降至 29.7%. 同样, 在 3 个数据集上的 *MFR* 值也未见明显的大幅下降, 依然保持在较高水准. 该结果表明, 当 BEEP 成功预测到缺陷代码令牌后, 其大体上可以准确预测出与该令牌相关的代码变换操作.

综上所述, BEEP 不仅可以定位与缺陷有关的代码令牌, 也可以准确预测修复该缺陷令牌需要进行的代码更改操作.

4.5.3 RQ3: 不同的模块对方法的有效性产生怎样的影响?

为了研究我们模型中不同模块对模型有效性带来的影响, 我们进行了消融实验. 具体地, 我们关注于模型中的 3 个设计: (1) 将令牌拆分成子令牌; (2) 基于节点的抽象语法树路径编码方法; (3) 全连接层. 在第一个实验中, 我们移除将令牌拆分为子令牌的过程, 直接将代码令牌送入模型的令牌嵌入矩阵 $E_t(\cdot)$ 中得到向量表示, 值得注意的是, 由于令牌嵌入矩阵是随着模型其他部分一同训练的, 这一实验设置下训练出的令牌嵌入矩阵与前文得到的并不相同; 在第 2 个实验中, 我们不使用经过编码后的节点序列去表示抽象语法树路径, 相反, 我们直接用独热向量去直接表示抽象语法树路径; 在第 3 个实验中, 我们移除模型中原有的全连接层, 将经过编码的操作路径直接送入到编码器中. 实验结果展示在表 5 中.

表 5 消融实验结果

预测目标	模型设计	*MFR*		
		CoCoNut	ManySStuBs4J	Defects4J
缺陷代码令牌	–令牌拆分	4.7	7.2	7.3
	–基于节点的路径编码	**3.9**	7.2	7.2
	–全连接层	4.0	7.0	6.7
	BEEP	**3.9**	**6.7**	**6.5**
操作路径	–令牌拆分	13.5	16.4	16.4
	–基于节点的路径编码	7.6	12.3	14.6
	–全连接层	8.3	12.6	13.7
	BEEP	**7.5**	**11.5**	**12.3**

在表 5 中我们展示了去掉不同模块后方法的有效性 (以 *MFR* 的数值体现), 我们注意到, 在每一组实验中, 去掉令牌拆分后的 *MFR* 值始终是最高的. 例如, 在对 CoCoNut 数据集上的操作路径进行预测时, 无令牌拆分的模型 *MFR* 值达到 13.5 而其他 3 种模型的 *MFR* 值均在 8 左右. 在 3 个数据集上, 一些模型的变体可以取得与 BEEP 相接近的 *MFR* 值, 例如在 CoCoNut 数据集上预测缺陷代码令牌时, 去掉基于节点的路径编码方式后的模型 *MFR* 值与 BEEP 的一致, 均为 3.9, 但是 BEEP 的 *MFR* 值在每一组实验中均为最低的, 这说明我们的各个模块共同作用使得 BEEP 达到了最优的效果.

综上所述, BEEP 模型中的各个模块共同作用使该模型的有效性达到最优, 其中对代码令牌进行拆分后再进行编码对模型有效性的提升最明显.

5 分析与讨论

在评估了 BEEP 方法在细粒度缺陷定位上的有效性后, 本节着重讨论如何构建基于细粒度缺陷定位的缺陷自动修复技术并分析其有效性与效率.

5.1 基于细粒度缺陷定位的缺陷自动修复

在对当前的缺陷自动修复技术进行了系统调研后, 我们发现, 已有技术[37,42,45]与当前缺陷定位技术有着高耦合关系, 也即当前修复技术仅接收现有的语句级别的定位结果为输入, 后运用不同的算法生成新的正确语句. 因此, 我们提出两种基于细粒度缺陷定位信息 (缺陷代码令牌) 的修复方案, 其工作流程如下.

● 输入. 我们假定缺陷方法是已知的. 由于近期的方法级缺陷定位技术[1,31]在定位效果上取得了较大进展, 因此这是一个较为合理的假设.

● 缺陷定位. 我们使用 BEEP 对输入的缺陷方法进行预测, 得到一个排序后的操作路径列表.

● 操作路径选择. 已有研究表明, 搜索空间过大会降低修复工具的效率, 由表 3 的结果, 绝大部分缺陷的标准操作路径都出现在 BEEP 预测结果的前 20 条中, 因此我们在这一步中仅考虑前 20 条预测结果. 随后进行不断的尝试, 每次按照排名顺序选取每条操作路径, 将其输入到下一阶段的补丁生成过程中, 如果尝试了所有 20 条路径但并没有生成能够通过所有测试用例的补丁, 则尝试将操作路径两两组合 (例如, 将预测结果中的第一条和第二条操作路径一同送入下一阶段). 为了限制搜索空间在合理的范围内, 本文提出的方法目前最多将两条路径组合在一起. 当生成能够通过所有测试用例的补丁后, 则结束此过程.

● 补丁生成. 在此阶段, 补丁通过一个代码补全技术或一套简单的启发式生成. 我们将在整个工作流程介绍完后详细介绍这两种补丁生成方法.

● 补丁验证. 生成的补丁通过执行测试用例进行验证. 当一个能够通过所有测试用例的补丁被生成后, 整个工作流程被结束. 我们随后人工检验该补丁的正确性. 该过程通过人工对生成的补丁与开发人员提供的标准补丁进行比对, 依据 Liu 等人[3]提出的补丁正确性评判规约, 检验生成的补丁是否与标准补丁具有语义一致性.

下面我们详细介绍两种补丁生成方法.

● 基于代码补全的补丁生成. Alon 等人[14]提出了一种利用软件自然性的静态代码补全方法, AnyCodeGen, 能够像预测自然语言中缺失的文本一样预测程序中缺失的代码元素. 实验结果表明该方法在 Java 语言上的补全能力显著优于包括序列到序列模型 (Seq2Seq) 在内的早期方法. 因此, 我们使用该方法生成补丁. 在获取到操作路径后, 我们解析出其中的缺陷代码令牌与变换操作. 如果变换操作为 DELETE, 我们直接删除该令牌; 如果变换操作为 UPDATE, 我们移除原令牌后将缺陷方法作为输入传给 AnyCodeGen, 将其返回的代码补全结果作为补丁; 如果变换操作为 INSERT, 我们在该令牌后添加占位符并调用 AnyCodeGen.

● 基于启发式的补丁生成. 我们根据 BEEP 预测的缺陷代码令牌的类别设计了一套启发式规则用以生成补丁.

• 如果缺陷令牌是一个运算符, 将其变为另一个运算符 ($==\rightarrow!=$, $>=\rightarrow>$).

• 如果缺陷令牌是一个布尔值, 将其变为相反的布尔值 (false $\rightarrow$ true).

• 如果缺陷令牌代表一种数据类型, 将其变为另一种数据类型 (int $\rightarrow$ float).

• 如果缺陷令牌是一个普通标识符, 从缺陷方法的代码令牌序列中选取距离其最近的 5 个用于生成补丁. 由于 AnyCodeGen 针对代码补全仅返回 5 个候选补丁, 因此此处也仅考虑 5 个距离最近的代码令牌.

5.2 缺陷自动修复实验设置

为了分析基于细粒度缺陷定位的程序自动修复方案的修复效果, 我们在 4 个常用的 Java 语言缺陷数据集上进行实验, 分别为 Defects4J[10], Bears[66], QuixBugs[67]以及 Bugs.jar[68]. 已有修复方法广泛地于这些数据集上进行测评, 因此我们可以直接将修复结果与已有方法进行比较.

在评价指标上, 我们关注修复有效性和效率两方面. 在有效性方面, 除了广泛讨论的召回率 (*recall*, 即能对多少个缺陷生成语义正确的补丁), 我们着重关注近年来逐渐被研究人员重视的另一个方面, 修复的准确率 (*precision*, 即生成的通过所有测试用例的补丁中有多少是真正正确的). 假定一个程序修复工具对 x 个缺陷程序生成了能够通过所有测试用例的补丁, 其中 y 个补丁是与开发人员提供的标准补丁语义一致的, 而缺陷数据集中共有 z 个缺陷, 则其准确率为 $precision = \frac{y}{x}$, 其召回率为 $recall = \frac{y}{z}$. 在修复效率方面, Liu 等人[3]指出时间开销这一指标与机器配置、实验设置等因素具有强关联关系, 因此使用候选补丁数 (number of patch candidates, NPC) 即修复工具在搜索到第 1 个通过所有测试用例的补丁之前产生的候选补丁的数量, 是一个更加合理的选择. 因此, 在本文中我们选用 NPC 值作为衡量修复效率的指标.

我们关注修复准确率和效率的原因在于, 缺陷修复准确率和效率能够从侧面反映出缺陷定位的有效性: 一方面, 已有研究表明[69]超过 90% 的修复工具产生的正确补丁是在缺陷位置处生成的, 因此, 倘若缺陷令牌能够获得较高的排名, 则修复工具能够较早生成正确补丁, 即取得高修复准确率; 反之, 若缺陷令牌排名靠后, 修复工具不能够较早生成正确补丁, 且易于在排名靠前的非缺陷位置处生成过拟合补丁, 使修复准确率下降. 另一方面, 本文设计的修复方法根据可疑值大小迭代式地依次对代码令牌进行修复尝试, 搜索空间较大且存在组合爆炸的可能, 将缺陷令牌排在靠前的位置将大大提升修复方法的效率.

5.3 缺陷修复有效性分析

表 6 展示了基于细粒度缺陷定位信息的缺陷修复方法的修复有效性. 表中每个数据集下括号中的数字代表该缺陷数据集包含的缺陷数, 诸如“*m*/*n*”形式的数据表示某一修复方法在某一数据集下对 n 个缺陷生成了可疑补丁, 其中 m 个补丁是语义正确的. 根据第 5.2 节中的评价指标相关内容, m 值越大代表该修复工具的召回率越高, 而 $\frac{m}{n}$ 的值即代表该修复工具的准确率. 我们将两种方法同两类已有技术进行比较, 一类是当前技术中具有较高准确率的 (如 CapGen、ACS), 一类是当前技术中具有较高召回率的 (如 TBar、CURE). 结果显示两种基于细粒度缺陷定位信息的修复流水线都能够达到 100% 的准确率, 并且在人工检验生成的补丁的过程中, 我们发现由这两种方法生成的能够通过所有测试用例的补丁都与标准补丁在相同位置完成修改. 进一步分析表明, 基于细粒度缺陷定位信息的修复流水线能够取得 100% 准确率的关键因素在于其能够准确定位到缺陷令牌并进行令牌层面的操作, 从而避免修改其他与缺陷无关的内容. 图 1 中所示的缺陷是一个相应的案例, BEEP 成功将包含缺陷令牌“<”以及代码变换操作 UPDATE 的操作路径排在所有操作路径的第 3 位, 随后, 基于启发式的补丁生成方法将“<”替换为“<=”, 从而生成了正确的补丁, 避免了过拟合补丁的产生. 此处需要注意的是, 我们将代码中的每个操作符也视为代码令牌、为每个操作符构建了操作路径 (一个直观的例子如图 3 所示) 并与其他构建的操作路径一同进行训练, 因此我们的方法也可以准确定位缺陷操作符. 我们将在第 5.5 节案例分析中分析更多被基于细粒度缺陷定位信息的修复流水线成功修复的案例. 我们还注意到, 与第 1 类已有工具相比, 我们提出的修复方案能够修复数量大致相当的缺陷, 即拥有大致相当的召回率. 例如, ACS 能够正确修复 18 个缺陷, 而我们的方案 (BEEP + 启发式) 能够正确修复 21 个缺陷. 另一方面, 尽管第 2 类已有工具在能够修复的补丁数量上优于我们提出的方案, 即拥有较高的召回率, 它们的准确率却低了很多. 例如, DLFix 的准确率甚至低于 50%, 说明在其生成的能通过所有测试用例的补丁中, 不正确补丁要多于正确补丁. 进一步分析表明, 对于 21 个能被基于 BEEP 与启发式的修复流水线正确修复的缺陷, 表中所示的 4 个第 2 类修复工具均对其中 2–3 个缺陷生成了过拟合补丁. 由于在实际中开发人员在将补丁集成到系统中之前往往要检查补丁的正确性, 因此过低的准确率会加重开发人员使用程序修复工具的负担.

表 6 中括号里列出的数字代表提出的修复方案在该缺陷数据集上能够修复的未被已有修复方法 (Jiang 等人[45] 所列出的 26 个) 解决的缺陷个数. 我们注意到, 即使是在修复领域最广为使用的 Defects4J 数据集上, 我们提出的修复方案依然能够成功修复 2 个不能被当前任何一种修复技术解决的缺陷. 这表明, 基于令牌的程序修复在未来是值得探索的方向. 此外, 在 4 个数据集上, 基于启发式的方案比基于代码补全的方案多成功修复了 5 个缺陷, 这表明, 对于程序修复而言, 在缺陷方法内部寻找合适的用于生成补丁的修复原料可能会优于利用大数据预测缺陷代码中缺失的部分.

表 6 基于细粒度缺陷定位的修复方法在真实程序上的修复有效性

修复工具	Defects4J (393)	Bears (251)	QuixBugs (40)	Bugs.jar (1 158)	*precision* (%)
JAID[70]	25/31	—	—	—	80.6
CapGen[51]	21/25	—	—	—	84.0
ACS[50]	18/23	—	—	—	78.3
FixMiner[71]	25/31	—	—	—	80.6
SimFix[37]	34/56	—	—	—	60.7
DLFix[72]	30/65	—	—	—	46.2
TBar[42]	43/81	—	—	—	53.1
CURE[45]	57/104	—	26/35	—	59.7
BEEP + 代码补全	**16/16 (2)**	**2/2 (2)**	**4/4 (0)**	**5/5 (4)**	**100**
BEEP + 启发式	**21/21 (2)**	**0/0**	**5/5 (0)**	**6/6 (4)**	**100**

综上, 基于 BEEP 预测结果的缺陷自动修复能够具有高准确率, 为解决修复领域的补丁过拟合问题提供了新的思路.

5.4 修复效率分析

Liu 等人[3]的研究提供了 16 种已有方法在两种输入下 (传统的基于谱的缺陷定位结果以及直接输入标准补丁中修改的语句) 在 Defects4J 数据集上的 NPC 值. 为了进行合理的对比, 我们假定缺陷语句已知, 将基于 BEEP 预测结果与启发式规则的修复方案的 NPC 值与已有方法进行对比, 结果展示在图 6 中. 数据显示, 本文提出的方案的 NPC 值中位数为 2, 显著优于基于模板的程序自动修复工具 (如 TBar 和 kPAR[36]) 以及最新的基于启发式的修复技术 (如 SimFix). 我们观察到有 6 个工具的修复效率要优于我们提出的方案, 其中的 3 个 (DynaMoth[73], Nopol[74], ACS) 使用了约束求解与程序综合的技术, 使得生成的补丁有很大的概率能够满足测试用例提供的规约; 另外 3 个 (jKali[11], Kali-A[75], jMutRepair[11]) 在补丁生成的过程中仅运用了简单的修复策略, 大大简化了搜索空间, jKali 与 Kali-A 仅将条件语句的条件部分变为 true 或 false, jMutRepair 仅变更操作符. 值得注意的是, 本文提出的修复方案的 NPC 中位数小于 5, 而该方案对每个定位到的缺陷标识符尝试产生 5 个补丁, 因此该方案取得高修复效率的主要原因是 BEEP 能够准确定位到标准操作路径. 同时也应注意到, 我们提出的基于启发式的规则严格限制了补丁搜索空间, 使该修复方案失去了修复更多缺陷的机会, 因此, 如何利用细粒度缺陷定位结果在保持高修复效率的同时修复更多缺陷, 值得后续研究的深入探索.

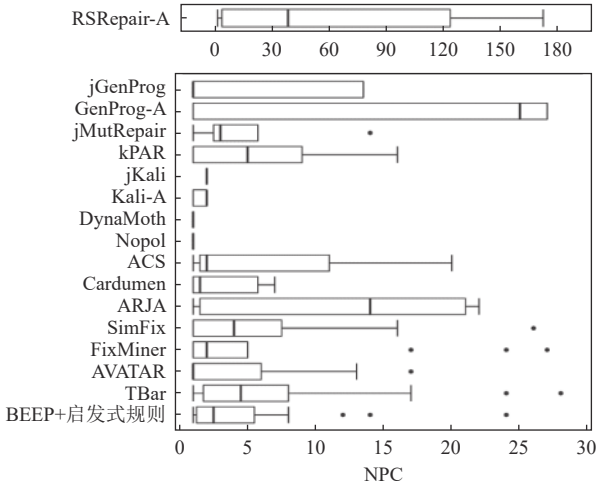


图 6 几种方法的 NPC 值比较结果

综上, 对缺陷代码令牌和修复所必需的代码变换操作的成功预测降低了 NPC 值, 从而提高了补丁生成阶段的效率.

5.5 案例分析

本节中我们将分析两个案例. 首先, 我们通过一个实例展示基于细粒度缺陷定位信息的缺陷自动修复方案能够准确定位并修改缺陷令牌, 从而避免产生过拟合补丁; 随后我们通过另一个实例展示基于细粒度缺陷定位信息的缺陷自动修复方案能够与已有修复技术互补 (即能够修复当前技术无法解决的缺陷) 的原因.

图 7 展示了 Defects4J 数据集中缺陷 Math-80 的标准补丁以及本文提出的修复方案与 SimFix 工具针对此缺陷产生的补丁. 可以看到, 此缺陷是由一条赋值语句中的数值错误引起的, BEEP 能够准确定位到出错的数值, 并从相近的代码令牌中选取合适的令牌 (即 4), 生成了与标准答案语义一致的补丁 (int $j = 4 \times n - 4$). 而 SimFix 则是在语句级别对补丁进行搜索, 尽管缺陷语句的可疑度排名较高, 但其并没有找到与缺陷语句相似度较高的代码用于生成补丁; 相反, SimFix 在非缺陷处生成了一个过拟合补丁. 这个案例表明, 本文提出的修复方案取得高准确率的原因在于, 精准的令牌级缺陷定位与修复可以避免过拟合补丁的产生.

```
// Math-80的标准补丁

 private boolean flipIfWarranted(final int n, final int step) {
    if (1.5 * work[pingPong] < work[4 * (n - 1) + pingPong]) {
-       int j = 4 * n - 1;
+       int j = 4 * (n - 1);
        for (int k = 0; k < 4; k += step) {
            final double tmp = work[i + k];

// BEEP + 启发式 生成的补丁

-       int j = 4 * n - 1;
+       int j = 4 * n - 4;

// SimFix生成的过拟合补丁

-       for (int k = 0; k < 4; k += step) {
+       for (int k = 0; k < 0; k += step) {
```

图 7　缺陷 Math-80 的标准补丁与两种修复方法生成的补丁

图 8 展示了 Defects4J 数据集中缺陷 Math-79 的标准补丁. 修复这个缺陷需要将源程序中两个数据类型 int 转化为 double. 这是一个多点缺陷[76]的案例, 即修复该缺陷需对源程序的多个地方进行更改. 已有技术在定位到缺陷语句后, 往往忽视其中的变量类型, 仅针对变量的赋值内容等搜索补丁, 缺乏专门针对数据类型的变异操作, 因而不能解决这一缺陷. 在基于 BEEP 预测结果与启发式规则的修复方案中, 两条标准操作路径排在 BEEP 预测结果的前两位. 在将操作路径两两组合后, 根据设计的启发式规则, 本文提出的方案对数据类型进行了变换, 因而成功生成了正确补丁. 该案例表明, 细粒度的缺陷定位可以关注到更多种缺陷的可能情况, 暴露出传统技术的盲区.

```
// Math-79的标准补丁

 public static double distance(int[] p1, int[] p2) {
-       int sum = 0;
+       double sum = 0;
        for (int i = 0; i < p1.length; i++) {
-           final int dp = p1[i] - p2[i];
+           final double dp = p1[i] - p2[i];
            sum += dp * dp;
        }
        return Math.sqrt(sum);
    }
```

图 8　缺陷 Math-79 的标准补丁

5.6 未来工作

缺陷定位的目标是辅助开发人员进行代码调试. 本文中仅从定量的角度出发衡量我们提出的 BEEP 方法对于人工调试的帮助, 即分析 BEEP 的定位结果是否能够将缺陷代码元素排在靠前位置, 但是缺少了定性分析的环节, 即考察 BEEP 的预测结果能够帮助开发人员更快更好地修复缺陷. 一种合理的实验设置是, 将 BEEP 的预测结果展示给一部分开发人员, 而对于另一部分开发人员则不展示任何信息, 观察两组开发人员在修复相同缺陷时的表现. 然而, 受到诸如工业界合作等因素的限制, 本文未能包含此部分内容, 故将其留作未来工作, 以便后续研究更好地探索细粒度缺陷定位对开发人员的调试工作带来的影响.

6 总 结

缺陷定位方法旨在自动地找到与程序失效相关的代码元素, 从而在代码调试阶段为开发人员提供帮助. 本文提出了一种基于指针网络的细粒度缺陷定位方法. 针对现有缺陷定位方法不能够为开发人员提供细粒度的缺陷信息这一问题, 本文提出利用神经网络和大数据预测与缺陷相关的代码令牌, 帮助开发人员更准确地定位缺陷代码. 本文基于指针神经网络, 通过将缺陷程序表示为一组抽象语法树路径, 利用程序结构信息获取预测结果. 在 3 个包含大量真实程序中补丁案例的数据集上的实验表明, 本文提出的细粒度缺陷定位方法不仅可以有效预测缺陷代码令牌以及修复该缺陷所必需的代码变换类型, 还显著优于基于统计信息和基于机器学习的基准方法. 此外, 在细粒度缺陷定位信息的基础上, 我们设计了基于代码补全与基于启发式的补丁生成策略从而形成完整的程序修复方案. 在多个缺陷数据集上的结果表明, 基于令牌级别缺陷定位信息的程序修复在有效性和效率两方面都具有较好前景.

References:

[1] Lou YL, Ghanbari A, Li X, Zhang LM, Zhang HT, Hao D, Zhang L. Can automated program repair refine fault localization? A unified debugging approach. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 75–87. [doi: 10.1145/3395363.3397351]

[2] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 199–209. [doi: 10.1145/2001420.2001445]

[3] Liu K, Wang SW, Koyuncu A, Kim K, Bissyandé T F, Kim D, 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 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 615–627. [doi: 10.1145/3377811.3380338]

[4] Long F, Rinard M. An analysis of the search spaces for generate and validate patch generation systems. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 702–713. [doi: 10.1145/2884781.2884872]

[5] Smith EK, Barr ET, Le Goues C, Brun Y. Is the cure worse than the disease? Overfitting in automated program repair. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 532–543. [doi: 10.1145/2786805.2786825]

[6] Tian HY, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF. Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: ACM, 2020. 981–992. [doi: 10.1145/3324884.3416532]

[7] Wang SW, Wen M, Lin B, Wu HJ, Qin YH, Zou DQ, Mao XG, Jin H. Automated patch correctness assessment: How far are we? In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: ACM, 2020. 968–980. [doi: 10.1145/3324884.3416590]

[8] Xiong YF, Liu XY, Zeng MH, Zhang L, Huang G. Identifying patch correctness in test-based program repair. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 789–799. [doi: 10.1145/3180155.3180182]

[9] Tao YD, Kim J, Kim S, Xu C. Automatically generated patches as debugging aids: A human study. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 64–74. [doi: 10.1145/2635868.2635873]

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

[11] Martinez M, Monperrus M. ASTOR: A program repair library for Java (demo). In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 441–444. [doi: 10.1145/2931037.2948705]

[12] Li Y, Wang SH, Nguyen TN, van Nguyen S. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 162. [doi: 10.1145/3360588]

[13] Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. In: Proc. of the 7th Int'l Conf. on Learning Representations. New Orleans: ICLR, 2019.

[14] Alon U, Sadaka R, Levy O, Yahav E. Structural language models of code. In: Proc. of the 37th Int'l Conf. on Machine Learning. PMLR, 2020. 245–256.

[15] Alon U, Zilberstein M, Levy O, Yahav E. A general path-based representation for predicting program properties. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Philadelphia: ACM, 2018. 404–419. [doi: 10.1145/3192366.3192412]

[16] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 40. [doi: 10.1145/3290353]

[17] Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 2692–2700.

[18] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016, 42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]

[19] 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. Buenos Aires: IEEE, 2017. 609–620. [doi: 10.1109/ICSE.2017.62]

[20] Abreu R, Zoeteweij P, Van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conf. Practice and Research Techniques-MUTATION. Windsor: IEEE, 2007. 89–98. [doi: 10.1109/TAIC.PART.2007.13]

[21] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proc. of the 24th Int'l Conf. on Software Engineering. Orlando: ACM, 2002. 467–477. [doi: 10.1145/581339.581397]

[22] Papadakis M, Le Traon Y. Metallaxis-FL: Mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5–7): 605–628. [doi: 10.1002/stvr.1509]

[23] Zhang XY, Gupta N, Gupta R. A study of effectiveness of dynamic slicing in locating real faults. Empirical Software Engineering, 2007, 12(2): 143–160. [doi: 10.1007/s10664-006-9007-3]

[24] Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementatio. Chicago: ACM, 2005. 15–26. [doi: 10.1145/1065010.1065014]

[25] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans. on Software Engineering, 2002, 28(2): 183–200. [doi: 10.1109/32.988498]

[26] Wong WE, Debroy V, Golden R, Xu XF, Thuraisingham B. Effective software fault localization using an RBF neural network. IEEE Trans. on Reliability, 2012, 61(1): 149–169. [doi: 10.1109/TR.2011.2172031]

[27] Koyuncu A, Liu K, Bissyandé TF, Kim D, 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 the Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 314–325. [doi: 10.1145/3338906.3338935]

[28] Mayer W, Stumptner M. Evaluating models for model-based debugging. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. L'Aquila: IEEE, 2008. 128–137. [doi: 10.1109/ASE.2008.23]

[29] 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 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: ACM, 2020. 907–918. [doi: 10.1145/3324884.3416566]

[30] 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, 2021, 47(2): 332–347. [doi: 10.1109/TSE.2019.2892102]

[31] 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 29th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 664–676. [doi: 10.1145/3468264.3468580]

[32] Xie H, Lei Y, Yan M, Yu Y, Xia X, Mao XG. A universal data augmentation approach for fault localization. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 48–60. [doi: 10.1145/3510003.3510136]

[33] Küçük Y, Henderson TAD, Podgurski A. Improving fault localization by integrating value and predicate based causal inference techniques. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 649–660. [doi: 10.1109/ICSE43902.2021.00066]

[34] Xie XY, Liu ZC, Song S, Chen ZY, Xuan JF, Xu BW. Revisit of automatic debugging via human focus-tracking analysis. In: Proc. of the

38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 808–819. [doi: 10.1145/2884781.2884834]

[35] 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. Saarbrucken: ACM, 2016. 165–176. [doi: 10.1145/2931037.2931051]

[36] Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Traon YL. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification. Xi'an: IEEE, 2019. 102–113. [doi: 10.1109/ICST.2019.00020]

[37] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 298–309. [doi: 10.1145/3213846.3213871]

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

[39] Le XBD, Lo D, Goues CL. History driven program repair. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Osaka: IEEE, 2016. 213–224. [doi: 10.1109/SANER.2016.76]

[40] 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. San Francisco: IEEE, 2013. 772–781. [doi: 10.1109/ICSE.2013.6606623]

[41] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 691–701. [doi: 10.1145/2884781.2884807]

[42] Liu K, Koyuncu A, Kim D, Bissyandé TF. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 31–42. [doi: 10.1145/3293882.3330577]

[43] Liu K, Koyuncu A, Kim D, Bissyandè TF. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In: Proc. of the IEEE 26th Int'l Conf. on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019. 456–467. [doi: 10.1109/SANER.2019.8667970]

[44] Lutellier T, Pham HV, Pang L, Li YT, Wei MS, Tan L. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 101–114. [doi: 10.1145/3395363.3397369]

[45] Jiang N, Lutellier T, Tan L. CURE: Code-aware neural machine translation for automatic program repair. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1161–1173. [doi: 10.1109/ICSE43902.2021.00107]

[46] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: ICLR, 2018.

[47] Bhatia S, Kohli P, Singh R. Neuro-symbolic program corrector for introductory programming assignments. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM. 2018. 60–70. [doi: 10.1145/3180155.3180219]

[48] Chakraborty S, Ding YRB, Allamanis M, Ray B. Codit: Code editing with tree-based neural models. IEEE Trans. on Software Engineering, 2022, 48(4): 1385–1399. [doi: 10.1109/TSE.2020.3020502]

[49] Zhu QH, Sun ZY, Xiao YA, Zhang WJ, Yuan K, Xiong YF, Zhang L. A syntax-guided edit decoder for neural program repair. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 341–353. [doi: 10.1145/3468264.3468544]

[50] Xiong YF, Wang J, Yan RF, Zhang JC, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 416–426. [doi: 10.1109/ICSE.2017.45]

[51] Wen M, Chen JJ, Wu RX, Hao D, Cheung SC. Context-aware patch generation for better automated program repair. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 1–11. [doi: 10.1145/3180155.3180233]

[52] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 313–324. [doi: 10.1145/2642937.2642982]

[53] Binkley D, Davis M, Lawrie D, Morrell C. To camelcase or under_score. In: Proc. of the 17th IEEE Int'l Conf. on Program Comprehension. Vancouver: IEEE, 2009. 158–167. [doi: 10.1109/ICPC.2009.5090039]

[54] Hill E, Binkley D, Lawrie D, Pollock L, Vijay-Shanker K. An empirical study of identifier splitting techniques. Empirical Software Engineering, 2014, 19(6): 1754–1780. [doi: 10.1007/s10664-013-9261-0]

[55] Rozovskaya A, Roth D. Grammatical error correction: Machine translation and classifiers. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. Berlin: ACL, 2016. 2205–2215. [doi: 10.18653/v1/p16-1208]

[56] Wang LH, Zheng XQ. Improving grammatical error correction models with purpose-built adversarial examples. In: Proc. of the 2020 Conf. on Empirical Methods in Natural Language Processing. ACL, 2020. 2858–2869. [doi: 10.18653/v1/2020.emnlp-main.228]

[57] Hindle A, Barr ET, Su ZD, Gabel M, Devanbu P. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 837–847. [doi: 10.1109/ICSE.2012.6227135]

[58] Brody S, Alon U, Yahav E. A structural model for contextual code changes. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 1–28. [doi: 10.1145/3428283]

[59] Li J, Sun AX, Han JL, Li CL. A survey on deep learning for named entity recognition. IEEE Trans. on Knowledge and Data Engineering, 2022, 34(1): 50–70. [doi: 10.1109/tkde.2020.2981314]

[60] Mannor S, Peleg D, Rubinstein R. The cross entropy method for classification. In: Proc. of the 22nd Int'l Conf. on Machine Learning. Bonn: ACM, 2005. 561–568. [doi: 10.1145/1102351.1102422]

[61] Karampatsis RM, Sutton CA. How often do single-statement bugs occur?: The ManySStuBs4J dataset. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul: ACM, 2020. 573–577. [doi: 10.1145/3379597.3387491]

[62] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 169–180. [doi: 10.1145/3293882.3330574]

[63] 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. Santa Barbara: ACM, 2017. 261–272. [doi: 10.1145/3092703.3092731]

[64] Liu K, Kim D, Koyuncu A, Li L, Bissyandé TF, Le Traon Y. A closer look at real-world patches. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 275–286. [doi: 10.1109/ICSME.2018.00037]

[65] Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: ACM, 2011. 1–10. [doi: 10.1145/1985793.1985795]

[66] Madeiral F, Urli S, Maia M, Monperrus M. BEARS: An extensible Java bug benchmark for automatic program repair studies. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019. 468–478. [doi: 10.1109/SANER.2019.8667991]

[67] Lin D, Koppel J, Chen AGL, Solar-Lezama A. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proc. of the 2017 Companion of the ACM SIGPLAN Int'l Conf. on Systems, Programming, Languages, and Applications: Software for Humanity. Vancouver: ACM, 2017. 55–56. [doi: 10.1145/3135932.3135941]

[68] Saha R, Lyu YJ, Lam W, Yoshida H, Prasad MR. Bugs. jar: A large-scale, diverse dataset of real-world Java bugs. In: Proc. of the 15th IEEE/ACM Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 10–13.

[69] Wang SW, Wen M, Chen LQ, Yi X, Mao XG. How different is it between machine-generated and developer-provided patches? An empirical study on the correct patches generated by automated program repair techniques. In: Proc. of the 2019 ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Porto de Galinhas: IEEE, 2019. 1–12. [doi: 10.1109/ESEM.2019.8870172]

[70] 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. Urbana: IEEE, 2017. 637–647. [doi: 10.1109/ASE.2017.8115674]

[71] Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Le Traon Y. FixMiner: Mining relevant fix patterns for automated program repair. Empirical Software Engineering, 2020, 25(3): 1980–2024. [doi: 10.1007/s10664-019-09780-z]

[72] Li Y, Wang SH, Nguyen TN. DLFix: Context-based code transformation learning for automated program repair. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 602–614.

[73] Durieux T, Monperrus M. DynaMoth: Dynamic code synthesis for automatic program repair. In: Proc. of the 11th Int'l Workshop on Automation of Software Test. Austin: ACM, 2016. 85–91. [doi: 10.1145/2896921.2896931]

[74] Xuan JF, Martinez M, Demarco F, Clément M, Marcote SL, Durieux T, Berre DL, Monperrus M. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. on Software Engineering, 2017, 43(1): 34–55. [doi: 10.1109/TSE.2016.2560811]

[75] Yuan Y, Banzhaf W. ARJA: Automated repair of Java programs via multi-objective genetic programming. IEEE Trans. on Software Engineering, 2020, 46(10): 1040–1067. [doi: 10.1109/TSE.2018.2874648]

[76] Wang SW, Mao XG, Niu N, Yi X, Guo AB. Multi-location program repair strategies learned from successful experience. In: Proc. of the 31st Int'l Conf. on Software Engineering and Knowledge Engineering. Lisbon: KSI, 2019. 713–777. [doi: 10.18293/SEKE2019-007]

王尚文(1994－), 男, 博士生, CCF 学生会员, 主要研究领域为智能化软件工程, 软件维护与演化.

刘逵(1988－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化软件工程.

林博(1996－), 男, 博士生, 主要研究领域为智能化软件工程, 软件维护与演化.

黎立(1989－), 男, 博士, 高级讲师, 博士生导师, CCF 专业会员, 主要研究领域为智能化软件工程, 软件安全.

KLEINJacques(1979－), 男, 博士, 教授, 博士生导师, 主要研究领域为可信软件工程.

BISSYANDÉ Tegawendé François(1985－), 男, 博士, 副教授, 博士生导师, 主要研究领域为可信软件工程.

毛晓光(1970－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为高可信软件技术.