计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(7):57−64 [doi: 10.15888/j.cnki.csa.009152]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于克隆检测技术的性能 Bugs 查找①

邓子含[1,2], 徐　云[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 230027)
[2](安徽省高性能计算重点实验室, 合肥 230026)
通信作者: 徐　云, E-mail: xuyun@ustc.edu.cn

摘　要: 性能 bug 是指代码中降低程序运行效率的缺陷. 现有的检测工具只能查找特定类型的性能 bug 并且需要复杂的程序分析过程, 因而缺乏通用性并且时空开销巨大. 同时, 有许多经典的克隆检测技术被用于一般性相似代码检测, 但是它们只能检测高度相似的代码或者需要依赖训练集, 使得它们难以用于在真实数据集中查找性能 bug. 基于此, 通过构建带有标记 token 的代码模板, 本文提出一种使用克隆检测技术来查找多种类型的性能 bug 的方法. 通过对不同类型和频度的 token 标记不同的权重, 本文提出的方法可以区分其重要性并因此提取出代码中的关键信息. 在真实项目构成的数据集上的实验表明, 本方法可以发现更多类型的性能 bug 同时比现有工具耗时更少. 另一项实验也证明了本方法显著提升了基于 token 的克隆检测技术的检测能力, 相比于现有的克隆检测方法更适合用于性能 bug 查找.

关键词: 性能 bug 检测; 代码克隆检测; 代码模板; 带标记的 token

引用格式: 邓子含,徐云.基于克隆检测技术的性能 Bugs 查找.计算机系统应用,2023,32(7):57–64. http://www.c-s-a.org.cn/1003-3254/9152.html

Finding Performance Bugs Based on Clone Detection Technique

DENG Zi-Han[1,2], XU Yun[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Key Laboratory of High Performance Computing of Anhui Province, Hefei 230026, China)

Abstract: Performance bugs are defects in codes that slow down program execution. Existing detection tools can only find certain types of performance bugs and require complex program analysis processes. Therefore, they lack generality and need high costs in space and time. Meanwhile, many classical clone detection techniques have been used for general similar code detection, but they can only detect highly similar codes or rely on training datasets, which makes them inapplicable for detecting performance bugs in real-world datasets. To this end, this study proposes a method of using clone detection techniques to find multiple types of performance bugs by constructing code templates with labeled tokens. By labeling tokens with different weights according to their types and frequencies, this method can distinguish tokens' importance and thus extract key information from codes. Experimental results on real-world projects show that this method can find more types of performance bugs and consume less time than existing tools. Another experiment also proves that this method significantly improves the detection capability of token-based clone detection techniques and is more suitable for finding performance bugs than existing clone detection techniques.

Key words: performance bugs detection; code clone detection; code template; labeled token

① 基金项目: 国家自然科学基金 (61672480); 国家外专局 111 引智计划 (BP0719016)
收稿时间: 2022-12-21; 修改时间: 2023-01-19; 采用时间: 2023-02-08; csa 在线出版时间: 2023-04-07
CNKI 网络首发时间: 2023-04-10

在代码中, 影响程序运行速率和性能的缺陷被称为性能 bug[1]. 已有的研究表明, 这些性能 bug 会显著影响用户使用体验甚至造成社会性事件[2–4]. 对于性能 bug 的定义最早是由 Jin 等人[5] 提出, 在此之后, 又有许多对性能 bug 现状以及成因的综合性研究[6,7]. 这些研究表明, 性能 bug 在现实的项目中是普遍存在的, 同时对它们的修复能带来软件性能的显著提升. 因此, 对性能 bug 进行查找是一项非常有意义的工作.

已经有许多工具被应用于特定类型的性能 bug 检测. 根据检测方法的不同, 可以分为以下两类: 基于动态检测的, 主要通过测试案例来获取运行时的相关信息[8–10]. 基于静态检测的, 使用程序流程图或其他类似结构来获取并分析代码的整体架构[11,12]. 近年来, 又出现了使用自然语言处理技术[13] 和机器学习模型[14] 来查找性能 bug 的, 同样基于动态分析的工具. 这些方法往往将某种类型的性能 bug 转化为特定的模式, 再通过复杂的分析过程寻找与这种模型相似的代码片段 (例如: 低效循环[8,9,11], 冗余的载入操作[10], 多余的集合遍历操作[11], 配置文件相关[13] 以及微处理器中[14] 的性能 bug). 因此, 这些方法不仅时空开销巨大, 而且其检测流程只对一种类型的性能 bug 有效, 缺乏通用性.

另一方面, 代码克隆技术可以被视为一种通用的性能 bug 检测手段: 通过将特定的模式转化为代码模板, 进而查找与其相似的代码片段, 从而就能查找到相应类型的性能 bug. 根据源代码在克隆检测的分析过程中表征方式的不同, 代码克隆技术可以分为以下几类: 基于 token 的[15–18], 基于抽象语法树 (AST) 的[19,20], 基于程序依赖图 (PDG) 的[21], 基于深度学习的[22–24]. 基于 token 的方法检测速度快, 开销小, 但难以检测相似度较低的克隆. 基于 AST 和 PDG 的可以检测这类克隆但是运行时间较长. 基于深度学习的方法具有最好的检测能力, 但是其效果严重依赖于训练数据集. 基于以上考虑, 本文提出通过提高基于 token 方法的检测能力, 并将其应用到性能 bug 的检测中, 实现一种快速且通用的性能 bug 查找方法.

根据检测过程基本检测单元的粒度的不同, 基于 token 的检测方法可以分为行粒度和词粒度两类. 基于行粒度的方法中, CCAligner[17] 通过建立 e-误配索引表检测存在多行插入或删除的 large-gap 克隆, 并在后续的工作 LVMapper[18] 得到了进一步发展. 但是, 这些方法在面对多个代码行都存在细微修改的情况时, 会出现检测效果不足的问题. SourcererCC[16] 采用基于词粒度的方法, 可以较好地处理这类问题, 但是无法检测 large-gap 类的克隆. 因此, 本文提出基于词粒度和行粒度综合进行考量的检测方案.

首先, 本文提出对 token 按照其重要程度进行分级并打上标记. 在之前的工作, SourcererCC 中, 提出了根据频度对 token 进行二元划分 (低频, 高频) 并只保留低频的 token 用作后续进一步的分析. 本文提出基于 TF-IDF 算法对不同类型的 token 进行多级别划分并赋予不同的权重. 通过这种方式, 本方法可以更有效地提取能反映代码结构特征的关键信息.

接着, 本方法提出通过构建能够反映性能 bug 特征的源代码模板, 在代码仓中搜索相似的代码片段, 从而将性能 bug 检测转化为相似性代码查找的问题. 本方法首先以带标记的 token 为基本单位 (词粒度) 来衡量两行代码之间的相似度, 再以行为基本单位衡量两段代码之间的相似度. 计算它们的最长公共子序列 (longest common sub-sequences, LCS), 如果序列长度达到阈值则判定为相似. 从而克服现有方法在检测时遇到的挑战, 提高基于 token 方法的检测能力.

1 基于克隆检测技术的性能 bug 查找

本方法可以分成以下两个主要步骤: 词法分析和克隆检测. 整体框架如图 1 所示.

词法分析阶段的主要目的是将模板以及源代码转化成标准化的带标记 token 序列. 首先将代码仓中的源码提取成以函数为基本单位的代码块 (模板也是以函数为基本单位构建的). 之后进行 token 化, 将各类基本词法单元转化为类型有限的 token 并对其标记上不同的权重.

在克隆检测阶段, 仿照 CCAligner[17] 的方式, 将每一个代码块按行哈希后构建全局索引, 使用模板查询索引得到候选的代码块, 之后计算每个候选代码块与模板之间的相似性, 从而得到查找到的性能 bug 代码段.

1.1 词法分析

首先, 本方法用 SourcererCC[16] 和 CCAligner[17] 等方法中使用的代码提取工具 TXL[25] 从代码仓中代码块. 这些代码块以函数为粒度按 XML 文件的格式存储, 并且标记有其所属的文件路径以及起止行.

之后, 使用由 Flex[26] 生成的扫描器对代码块进行 token 化. 代码块中的每个基本词法单元都会被映射成

一种类型的 token (比如关键字, 操作符等). 对于变量名, 会统一映射为 id, 这种归一化操作可以有效地处理因为换名造成的差异, 大幅提高检测能力. 代码 1 给出了一个 token 化过程示例.

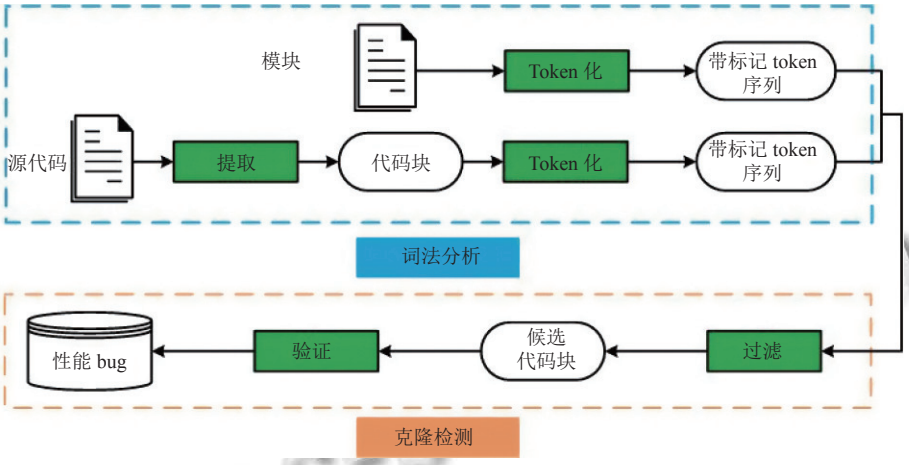

图 1 整体框架

代码 1. 代码块 token 化后的结果示例

① 源代码

```
public int gcd(int a,int b)
{
  if (b==0) return a;
  return gcd(b, a%b);
}
```

② Token 序列

```
PUBLIC IND ID(INT ID, INT ID)
{
IF (ID == CONSTANT) RETURN ID;
RETURN ID(ID, ID MOD ID);
}
```

下一步, 本方法基于频度来区分不同类型 token 的重要性. 以来自 BigCloneBench[27] 的代码作为数据集, 基于 TF-IDF 算法计算不同 token 的权重: 本方法以函数作为基本的统计单位, 计算每一类 token 在每个函数中平均出现的次数取对数后作为 *tf*, 计算每类 token 在所有函数中出现的比例, 取倒数后再取对数作为 *idf*. 相乘得到 *tf-idf* 后, 把得到的结果正则化至 1–10 的整数作为其权值, 权值为 10 的 token 会被标记为是关键 token, 在后续的相似性判断中做特殊处理.

表 1 给出了部分 token 类型对应的权重作为示例. 其中: “int”“else”“for”是不同语言中的关键字; “id”如上文所指, 对应变量名函数名; “constant”对应代码中出现的具体数值 (比如−10, 2.5, 1E7 等); “MOD”表示操作符“%”; “{”以及“,”及源码中的标志符, 在 token 化过程中直接保留. 这些计算得到的权重结果会以字典的形式存储下来, 方便后续流程直接查找. 在之后的检测阶段, 所使用的都是基于 BigCloneBench 的同一套权重标记方案.

表 1 不同类型 token 权重示例

Token 类型	权重
{	1
,	1
int	3
id	4
mod (%)	4
constant	5
else	9
for	10

1.2 克隆检测

在克隆检测阶段, 首先我们要根据待检测的性能 bug 类型构建相应的代码模板. 模板只需要保留能反映待检测性能 bug 模式最基本的结构即可, 后续的查找算法可以有效地处理待检测代码段有其他无关代码的情况. 同时, 根据代码提取工具 TXL[25] 的需要, 模板要以函数粒度来构建, 不要求能够编译, 但必须符合语法规范. 代码 2 给出了一个用 Java 语言编写的, 用于检测冒泡排序的模板示例. 对于任何其他类型的性能 bug, 通过构建类似于此的模板就能进行检测.

为了能够高效快速地查找性能 bug, 需要通过过滤

阶段筛选得到候选的代码块. 本方法对与待检测的代码块的每一行取哈希值并以此构建索引. 之后用模板中每一行对应的哈希值去查找该索引对应的代码块, 查找到则说明该代码块与模板存在公共行, 当公共行数达到设定的阈值后, 该代码块就会被判定为候选代码块并进行下一步的检测. 通过这种方式, 可以将至少 90% 的无关代码块筛选掉, 从而使得本方法能对大规模的代码仓进行检测.

代码 2. 模板示例

```
public static void bubbleSort(int[] a){
    for(int i=0;i<a.length-1;i++){
        for(int j=0;j<a.length-1;j++){
            if(a[j] > a[j+1]){
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }
    }
  }
}
```

对于得到的候选代码块, 本文逐一检测其与模板相似程度. 要衡量两个代码块之间的相似度, 其基础是衡量两行代码之间的相似度. 对于来自模板中的一行代码 L_t 和来自待检测代码块中的一行代码 L_c, 本方法计算其最长公共子序列) L_l, 并从以下 3 个方面来衡量其相似性: 模板行 L_t 和待检测行 L_c 的 token 个数; 模板行 L_t 和得到的最长公共子序列 L_l 的 token 的权值和; 以及两者所包含的关键 token 的个数. 具体的算法流程如下方的算法 1 所示.

算法以两行代码 L_t L_c 作为输入, 依次判断是否满足 3 个方面分别对应的阈值. 首先在算法的 1–4 行本算法计算两行代码 token 个数的差异, 如果其小于阈值 T_1, 说明两行代码长度差异过大, 判定为不是相似的, 直接返回 False. 无需再进行后续的计算过程. 若大于阈值 T_1, 之后则计算其最长公共子序列 L_l, 在算法的 6–10 行, 本方法计算 L_l 和 L_t 的 token 的权值和, 如果小于阈值 T_2, 则返回 False. 在算法 11–16 行, 计算 L_l 和 L_t 所包含的关键 token 的个数, 如果小于阈值 T_3, 则返回 False. 由于权值已经区分了不同 token 的重要程度, 只有当 L_l 包含了模板行 L_t 所拥有的重要度较高的 token, 这两个阈值才有可能达到. 因此, 当算法返回为 True 时, 说明待检测的这一行包含有我们在模板构建过程中所包含的关键信息. 整个算法的时间复杂度是 $O(n^2)$ 的, 其中 n 表示两行所包含的 token 个数. 通过这种检测方式, 即使两行之间有一定程度的差异, 也能够识别其相似.

算法 1. 判定两行代码是否相似

输入: 两行 token 化后的代码序列 L_t, L_c, 其中 L_t 为模板中代码行, L_c 为待检测的代码行.
输出: 是否相似 (布尔值).

1. N_t=L_t.tokenNumbers, N_c=L_c.tokenNumbers
2. **if** min(N_t, N_c)/max(N_t, N_c)< T_1 **then**
3. **return** False
4. **end if**
5. L_l=LCS(L_t, L_c)
6. W_l=sum(L_l.tokenWeight)
7. W_t=sum(L_t.tokenWeight)
8. **if** $W_l/W_t < T_2$ **then**
9. **return** False
10. **end if**
11. K_l=sum(L_l.keytokenNumbers)
12. K_t=sum(L_t.keytokenNumbers)
13. **if** $K_l/K_t < T_3$ **then**
14. **return** False
15. **end if**
16. **return** True

在有了两行之间的相似度判定标准之后, 我们以行为基本单位, 衡量模板和待检测代码块之间的相似性. 仍然是采用求最长公共子序列的算法, 被判定为相似的两行即为模板和待检测代码块的公共行. 统计得到的子序列长度与模板行数的比值, 如果达到设定的阈值, 则判定待检测代码块是要找的性能 bug.

在以上的过程中, 除了阈值 T_1 之外, 其他的计算都是在所得到的最长公子序列和模板之间进行的. 这是因为, 来自实际项目中代码仓可能包含许多处理其他业务的无关代码, 因而通过计算最长公共子序列将这些代码过滤掉. 只要待检测代码块包含模板中所构建的关键信息, 那么该代码块就包含本方法要查找的性能 bug. 通过这种方式, 可以查找到同一种类型的性能 bug 在实际项目中的各种变式.

2 实验结果与分析

为了验证本方法的有效性, 本文设计了两组实验: 在经典的开源项目中查找不同类型的性能 bug 并与现有工具作比较, 验证本方法的通用性; 与现有的一些克隆检测工具作比较, 验证说明本方法是最适于用于进

行性能 bug 查找的克隆检测技术, 同时证明本方法显著提高了基于 token 的克隆检测方法的检测能力.

2.1 检测方法通用性的实验结果与分析

为了与检测方法的通用性, 本实验构建了 3 类模板并且在 5 个开源的 Java 项目 (Ant, Lucence, PDFBox, Solr, Tomcat) 上进行了检测. 这 5 个项目都是被广泛使用的各类 Java 基本框架, 前两类模板为冒泡排序和顺序查找. 这两种类型的性能 bug 都是会显著影响程序性能的. 结果如表 2 所示.

表 2 两类模式的查找结果

项目	算法模式	个数
Ant	冒泡排序	0
	顺序查找	3
Lucence	冒泡排序	1
	顺序查找	4
PDFBox	冒泡排序	0
	顺序查找	3
Solr	冒泡排序	1
	顺序查找	7
Tomcat	冒泡排序	0
	顺序查找	4

通过表 2 的结果可以看到, 这两类明显的低效算法模式, 尤其是顺序查找, 在真实的项目里也是普遍存在的, 而且只需要做少许的修改, 就能带来软件质量的显著提升. 这两种类型的性能 bug 都是本方法首次发现的, 同时在这些项目中一些较老的版本中也能检查到同样的代码片段. 其中, Lucence 项目中所包含的冒泡排序代码段至少存在了 10 年之久.

为了与现有的性能 bug 检测工具作比较 , 本次实验又构建了第 3 类模板并与 Carmel[11] 进行比较. Carmel 是一个基于分析程序流程图的静态分析工具, 用于检测程序中可以包含可以提前跳出的低效循环语句的代码段. 本文将 Carmel 所查找的这类性能 bug 也构建模板, 在这 5 个 Carmel 也使用过的项目上进行对比实验.

表 3 可以看到, 在这 5 个项目上, 本方法能查找到 Carmel 所查找到的特定类型的性能 bug, 并且查找正确结果的数量基本相同, 还有保持更高的准确率. 这说明我们的方法在保持基本相同检测能力的同时有更高的准确率. 此外, 相比于 Carmel, 本方法的速度也要快很多. 前者需要 1 243 min, 而本方法只需要 57 min.

以上两组表 2 和表 3 的对比实验说明, 本方法能检测出多种类型的性能 bug, 与现有的方法比也更快速, 具有很好的通用性. 只要使用者构建出一种类型的模板, 本方法就能够有效地检测出其对应的性能 bug.

表 3 与 Carmel 比较

项目	Carmel	本文方法
Ant	1	1
Lucence	14	2
PDFBox	10	15
Solr	2	2
Tomcat	4	4
总计	31	24
正确的结果	23	21
精确度 (precision) (%)	74	91

2.2 与现有克隆工具的比较

本方法是基于克隆检测技术来查找性能 bug 的, 因此还需要将本方法与其他克隆检测工具作比较, 以说明本文提出的方法是最适宜于查找性能 bug 的克隆检测技术, 并且还需要验证本方法显著地提升了基于 token 的方法的检测能力.

首先, 将本方法与现有的 3 个基于 token 的方法 (SourcererCC[16], CCAligner[17], LVMapper[18]) 进行比对, 来检测其他基于 token 的方法是否适用于查找性能 bug. 本文选择来自 BigCloneBench[27] 的源文件构建数据集, 这个数据集是被学术界广泛使用的用于验证代码克隆检测能力的数据集这个数据集相比于真实项目构建的数据集规模更大, 因而能检测出更多的结果便于比较. 在本次实验中, 我们构建了 3 类模板, 前两类是前一个实验中使用到的冒泡排序和顺序查找, 第 3 个是算法复杂度为 $O(n^3)$, 可以进行优化的朴素矩阵相乘算法. 为了衡量这 3 种工具的检测能力, 本次实验将模板与测试数据集一并作为 3 个克隆检测工具的输入. 再从输出的克隆对查找与模板匹配为克隆对的函数, 即为查找到的结果数目. 因为不知道实验集中每一类性能 bug 具体有多少个, 因此本实验从查找的结果数目, 其中正确的结果数目以及正确率 3 个维度来衡量各方法的查找能力. 实验结果如表 4 所示.

对于 SourcererCC, 我们设置了两种不同的相似度阈值, 原因可以很明确地从表 4 中的结果看到. 最开始本实验采用的是默认的阈值 0.8, 结果无法查找到任何结果, 之后将阈值降低到 0.7, 查找到的结果均为正确结果但总数目依然很少, 之后又进一步将阈值下降到了 0.6, 结果准确率有严重下降, 是每一类模式中最低的. 类似的, 对于 LVMapper, CCAligner. 本实验也选取

了最佳阈值, 如果再降低一些阈值, 几乎不会找到更多的结果, 如果调高, 会有大量正确的结果被过滤掉. 通过表 4 的结果可以看到, 相比于现有的基于 token 的检测工具, 本文提出的方法查找的正确结果有明显的提升.

表 4 与基于 token 的工具比较

模式	工具	结果	正确数	准确率 (%)
冒泡排序	SourcererCC-0.7	10	10	100
	SourcererCC-0.6	38	25	66
	LVMapper	40	39	98
	CCAligner	79	76	96
	本文方法	182	132	73
顺序查找	SourcererCC-0.7	21	21	100
	SourcererCC-0.6	211	185	88
	LVMapper	108	106	98
	CCAligner	149	132	89
	本文方法	206	197	96
矩阵相乘	SourcererCC-0.7	2	2	100
	SourcererCC-0.6	23	2	9
	LVMapper	3	2	67
	CCAligner	17	12	71
	本文方法	104	87	84

同时, 本文有通过另一组在 BigCloneBench[26] 的实验来验证本方在代码克隆检测上的能力. 该数据集通过统计不同克隆检测工具检测到的科隆对数目, 即召回率来衡量克隆检测工具性能. 根据相似度从高到低, BigCloneBench 将克隆对分为以下几级: T1 (Type-1), T2 (Type-2), VST3 (Very Strong Type-3), ST3 (Strong Type-3), MT3 (Moderately Type-3), WT3&4 (Weakly Type3&Type-4). 检测结果如表 5 所示, 召回率对应每种克隆工具在对应类型上检测到的克隆对占该类克隆对的占比, 准确率即为每种检测工具检测的结果是否为真. 通过表中结果可以看到, 相比于其他几种基于 token 的方法, 本方法对于 ST3 以及相似度更低的克隆对的检测能力有显著的提升. 从而进一步说明本方法提高了基于 token 的方法的检测能力.

表 5 克隆检测能力比较 (%)

方法	召回率						准确率
	T1	T2	VST3	ST3	MT3	WT3&4	
本文方法	100	99	98	96	76	25.7	94
SourcererCC	100	98	93	61	5	0	98
CCAligner	100	99	97	70	10	0.2	77
LVMapper	100	99	98	82	19	0.3	87

近年来, 基于深度学习的工具在克隆检测中取得了良好的效果, 因此本文设置了另一组对比实验来验证基于机器学习的工具是否适用于查找性能 bug. 本实验选择了 ASTNN[22], 能找到的唯一的可适用于 Java 语言检测的开源工具. 基于深度学习的检测工具首先要构建训练集进行训练, 之后在测试集上进行验证. 这些数据集都是以带标签的克隆对构建的, 最终通过准确率, 召回率, $F1$ 值来衡量检测工具性能. 实验结果如表 6 所示.

表 6 与基于机器学习的工具比较 (%)

模式	工具	准确率	召回率	$F1$
冒泡排序	ASTNN-1	48	69	57
	ASTNN-2	50	100	67
	ASTNN-3	96	74	83
	本文方法	73	96	82
顺序查找	ASTNN-1	73	55	63
	ASTNN-2	64	100	78
	ASTNN-3	83	95	88
	本文方法	96	61	74
矩阵相乘	ASTNN-1	76	99	86
	ASTNN-2	76	100	86
	ASTNN-3	76	100	86
	本文方法	84	96	89

本实验收集了表 4 所示的与 token 的对比实验中的所有克隆对来构建测试数据集, 这些结果包含中有的正确有的错误. 本实验共收集了 277 对冒泡排序的克隆对, 其中 138 对为真. 504 对顺序查找的数据集, 其中 325 对为真. 120 对矩阵相乘的数据集, 其中 91 对为真.

对于训练数据集, 本实验首先采用 ASTNN 开源代码中默认的训练数据集, 这个数据集也是由来自 BigCloneBench[27] 的源文件构建的. 对应的结果为表 6 中的 ASTNN-1. 但在后续的实验分析中发现该数据集的标签似乎与 BigCloneBench 中的并不一致, 所以本实验重新提取了 BigCloneBench 的数据集与标签构建了新的训练数据集, 对应结果为 ASTNN-2. 相比于前者, ASTNN-2 的 $F1$ 值有了提升, 但结果表明其并没有做到对真假结果的区分.

基于以上考虑, 本实验将收集到的所有克隆对作为其训练和测试数据集. 按照 ASTNN 中相同的方式, 将每种模式的数据集随机分成 3 份: 60%, 20%, 20%, 分别用于训练, 验证和测试. 得到的结果对应表中的 ASTNN-3. 该结果证明了基于机器学习的克隆检测工具确实具有较好的学习能力, 不过本方法也与其保持在同等水平. 更为重要的是, 在现实的检测中, 难以对

每一类性能 bug 都构建像 ASTNN-3 这样的数据集. 除此之外, 基于深度学习的方法还有很高的时空开销.

通过以上的两组对比实验可以说明, 本方法相比于现有的克隆检测工具, 是最适于进行性能 bug 查找的. 本方法显著地提高了基于 token 方法的检测能力, 比起现有方法能查找结果数量有显著提升. 与基于深度学习方法的对比实验说明, 这类方法的效果高度依赖于数据集, 而且本方法依然能到达相同的水平.

3 结论与展望

本文创造性地提出将克隆检测技术应用于性能 bug 查找中, 从而实现了一种通用且快速的性能 bug 检测方法. 通过构建模板搜索相似代码, 做到使用一种检测技术查找任意类型的性能 bug. 在检测环境, 本文提出基于 TF-IDF 的算法来区分不同类型 token 的重要程度比依据次给 token 打上标签. 对比实验的结果表明了本方法相比于想要性能 bug 查找方法的通用性和低开销. 同时, 与现有的克隆检测工具的对比说明, 本方法显著提升了基于 token 的克隆检测技术的检测能力, 是最适于作为性能 bug 查找的克隆检测技术. 在后续的研究中, 可以进一步为 token 打上信息量更加丰富的标签, 进一步提升检测能力.

参考文献

1 Bugzilla@Mozilla. Bugzilla keyword descriptions. https://bugzilla.mozilla.org/describekeywords.cgi. [2023-01-20].

2 Bryant RE, O'hallaron DR. Computer Systems—A Programmers Perspective. Upper Saddle River: Pearson Education, 2003.

3 Richardson T. 1901 Census site still down after six months. https://www.theregister.com/2002/07/03/1901_census_site_still_down/. (2002-07-03).

4 Kallender P. Trend micro will pay for PC repair costs. http://www.pcworld.com/article/120612/article.html. [2023-01-20].

5 Jin GL, Song LH, Shi XM, *et al*. Understanding and detecting real-world performance bugs. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. Beijing: ACM, 2012. 77–88. [doi: 10.1145/2254064.2254075]

6 Liu Y, Xu C, Cheung S. Characterizing and detecting performance bugs for smartphone applications. Proceedings of the 36th International Conference on Software Engineering. Hyderabad: ACM, 2014. 1013–1024. [doi: 10.1145/2568225.2568229]

7 Zaman S, Adams B, Hassan AE. A qualitative study on performance bugs. Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. Zurich: IEEE, 2012. 199–208. [doi: 10.1109/MSR.2012.6224281]

8 Nistor A, Song LH, Marinov D, *et al*. Toddler: Detecting performance problems via similar memory-access patterns. Proceedings of the 35th International Conference on Software Engineering. San Francisco: IEEE, 2013. 562–571. [doi: 10.1109/ICSE.2013.6606602]

9 Song LH, Lu S. Performance diagnosis for inefficient loops. Proceedings of the 39th IEEE/ACM International Conference on Software Engineering. Buenos Aires: IEEE, 2017. 370–380. [doi: 10.1109/ICSE.2017.41]

10 Su PF, Wen SS, Yang HL, *et al*. Redundant loads: A software inefficiency indicator. Proceedings of the 41st IEEE/ACM International Conference on Software Engineering. Montreal: IEEE, 2019. 982–993. [doi: 10.1109/ICSE.2019.00103]

11 Nistor A, Chang PC, Radoi C, *et al*. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. Florence: IEEE, 2015. 902–912. [doi: 10.1109/ICSE.2015.100]

12 Olivo O, Dillig I, Lin C. Static detection of asymptotic performance bugs in collection traversals. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. Portland: ACM, 2015. 369–378. [doi: 10.1145/2737924.2737966]

13 He HC, Jia ZY, Li SS, et *al*. CP-detector: Using configuration-related performance properties to expose performance bugs. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne: ACM, 2020. 623–634. [doi: 10.1145/3324884.3416531]

14 Barboza EC, Jacob S, Ketkar M, *et al*. Automatic microprocessor performance bug detection. Proceedings of the 2021 IEEE International Symposium on High-performance Computer Architecture. Seoul: IEEE, 2021. 545–556. [doi: 10.1109/HPCA51647.2021.00053]

15 Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. Proceedings of the 16th IEEE International Conference on Program Comprehension. Amsterdam: IEEE, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

16 Sajnani H, Saini V, Svajlenko J, *et al*. SourcererCC: Scaling code clone detection to big-code. Proceedings of the 38th International Conference on Software Engineering. Austin: IEEE, 2016. 1157–1168. [doi: 10.1145/2884781.2884877]

17 Wang PC, Svajlenko J, Wu YZ, *et al*. CCAligner: A token based large-gap clone detector. Proceedings of the 40th International Conference on Software Engineering. Gothenburg: ACM, 2018. 1066–1077. [doi: 10.1145/3180155.3180179]

18 Wu M, Wang PC, Yin KQ, *et al*. LVMapper: A large-variance clone detector using sequencing alignment approach. IEEE Access, 2020, 8: 27986–27997. [doi: 10.1109/ACCESS.2020.2971545]

19 Baxter ID, Yahin A, Moura L, *et al*. Clone detection using abstract syntax trees. Proceedings of the 1998 International Conference on Software Maintenance. Bethesda: IEEE, 1998. 368–377. [doi: 10.1109/ICSM.1998.738528]

20 Jiang LX, Misherghi G, Su ZD, *et al*. DECKARD: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th International Conference on Software Engineering. Minneapolis: IEEE, 2007. 96–105. [doi: 10.1109/ICSE.2007.30]

21 Zou Y, Ban BH, Xue YX, *et al*. CCGraph: A PDG-based code clone detector with approximate graph matching. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne: IEEE, 2020. 931–942. [doi: 10.1145/3324884.3416541]

22 Zhang J, Wang X, Zhang HY, *et al*. A novel neural source code representation based on abstract syntax tree. Proceedings of the 41st IEEE/ACM International Conference on Software Engineering. Montreal: IEEE, 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]

23 Yu H, Lam W, Chen L, *et al*. Neural detection of semantic code clones via tree-based convolution. Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension. Montreal: IEEE, 2019. 70–80. [doi: 10.1109/ICPC.2019.00021]

24 Fang CR, Liu ZX, Shi YY, *et al*. Functional code clone detection with syntax and semantics fusion learning. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2020. 516–527. [doi: 10.1145/3395363.3397362.]

25 Cordy JR. The TxL programming language. https://www.txl.ca/. [2023-01-20].

26 Paxson V. This is Flex, the fast lexical analyzer generator. https://github.com/westes/flex/. [2023-01-20].

27 Svajlenko J, Islam JF, Keivanloo I, *et al*. Towards a big data curated benchmark of inter-project code clones. Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution. Victoria: IEEE, 2014. 476–480. [doi: 10.1109/ICSME.2014.77]

(校对责编: 牛欣悦)