基于改进 WL 图核的代码克隆检测方法*

班必奂[1,2],徐 云[2,3]
(1.中国科学技术大学 大数据学院,安徽 合肥 230026;
2.安徽省高性能计算重点实验室,安徽 合肥 230026;
3.中国科学技术大学 计算机科学与技术学院,安徽 合肥 230026)

摘 要:基于程序依赖图(Program Dependency Graph,PDG)的代码克隆检测方法是检测代码克隆的重要方法之一,近年来提出的基于 Weisfeiler-Lehman(WL)图核迭代的近似图匹配方法在克隆检测中取得了较好的效果,但 PDG 中少量顶点的差异会随着图核迭代传播到越来越多的顶点,从而导致算法召回率的下降。为此,针对 WL 图核在克隆检测应用中存在的问题,提出了一种基于改进 WL 图核的代码克隆检测方法,将 WL 图核迭代过程中采用的普通哈希算法替换为局部敏感哈希,同时引入向量的相似性度量方法,进一步提升了对 PDG 近似子结构的识别能力。实验结果表明,改进后的方法不仅可以检测出更多的差异克隆对,同时还保持了良好的精度和时间性能。

关键词:代码克隆检测;程序依赖图;Weisfeiler-Lehman 图核
中图分类号:TP311.5 **文献标识码**:A DOI:10.20044/j.csdg.2097-1788.2022.03.010
引用格式:班必奂,徐云. 基于改进 WL 图核的代码克隆检测方法[J].网络安全与数据治理,2022,41(3):60-66.

An improved WL graph kernel-based code clone detection method

Ban Bihuan[1,2], Xu Yun[2,3]
(1.School of Data Science, University of Science and Technology of China, Hefei 230026, China;
2.Key Laboratory of High Performance Computing of Anhui Province, Hefei 230026, China;
3.School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

Abstract: Code clone detection based on Program Dependency Graph(PDG) is one of the important methods in code clone detection domain. In recent years, a code clone detector that based on Weisfeiler-Lehman (WL) graph kernel has achieved success in detecting code clones. However, several different vertices in the PDG will "spread" to the whole PDG with iterations and finally results in a decline in recall rate. Therefore, an improved WL kernel-based clone detector is proposed to address the problem of the WL kernel when applied in clone detections. The ordinary hash algorithm used in the iterations of WL kernel is replaced with the local sensitive hash algorithm, and the vector similarity measure approach is employed, which enhances the ability to identify the similar PDG sub-structures. The experimental results show that compared to the original WL graph kernel-based detector, the improved method can not only detect more clones but also keep good accuracy and a low computing time.

Key words: code clone detection; program dependency graph; Weisfeiler-Lehman kernel

0 引言

在实际的软件系统开发过程中,开发人员出于提高开发效率或者降低软件错误风险的原因,通常会将其他模块中功能相同或者相近的代码片段通过复制—修改(或不修改)—粘贴的模式引入到当前正在开发的模块中,这些粘贴而来的功能相同或相似的代码片段在学术界通常被称为克隆代码[1]。大量的研究表明,过多的克隆代码会给软件系统带来维护成本提高[2]和 Bug 蔓延[3]等问题。例如,假设某一段代码中存在 Bug 并且这段代码被复制粘贴在

*基金项目:国家自然科学基金项目(61672480);教育部和外专局高等学校学科创新引智计划项目(B0703308)

不同的位置,那么这个 Bug 将会随着这些克隆代码在整个软件中传播,进而增加了维护的难度。因此,检测出软件系统中的克隆代码并对其进行统一管理是非常有必要的。

随着代码克隆检测技术的不断发展,高级别代码克隆(即 Type-3 和 Type-4 克隆)的检测成为当前的研究热点之一。由于高级别克隆代码往往在复制粘贴之后还进行了一些修改,因此也更有可能存在 Bug[4]。另外,修改所导致的差异性也使得这一类型的克隆更加难以被检测。

得益于程序依赖图(Program Dependency Graph,PDG)中所包含的大量语法结构信息和语义信息,基于 PDG 的克隆检测方法在检测高级别克隆上更具优势,能够检测出更多的结构相似但文本差异较大的高级别克隆代码。但在已有的方法中,采用精确图匹配算法的 CCSharp[5]、GPLAG[6]等克隆检测方法存在时间性能差和召回率低的缺点,而采用 Weisfeiler-Lehman(WL)图核进行近似图匹配计算的 CCGraph[7],受限于节点初始标签的单一性和 WL 图核迭代过程中对近似子结构识别能力的缺失,召回率也相对较低。为此,本文针对 WL 图核算法在 PDG 克隆检测应用中存在的缺陷,提出了一种改进思路:首先利用语法分析技术将 PDG 节点所包含的源代码信息解析为特征向量的形式来作为该节点的初始标签,使其能够保留更多的语义信息;然后通过引入局部敏感哈希技术提升 PDG 中近似子结构的识别能力,进而提升高级别克隆的召回率。

1 相关研究

1.1 基于文本的方法

基于文本的代码克隆检测方法通常将代码片段视为普通的字符序列,通过计算这些代码字符序列的相似度来寻找克隆。例如在经过一系列如删除注释、删除多余空行和对代码格式重新排布之后再使用字符串匹配算法来检测克隆。典型的研究成果为 Baker 所提出的工具 Dup[8]。此类方法优点在于检测的速度非常快,但往往只能检测出 Type-1 克隆以及部分 Type-2 克隆,更高级别的 Type-3 和 Type-4 克隆则几乎无法检测。

1.2 基于 Token 的方法

基于 Token 的代码克隆检测方法主要思路依然是采用字符串匹配相关的算法来检测克隆,其与基于文本的方法最大的区别就是在进行比对之前增加了词法分析过程,例如,将变量名统一解析为“id”。因此,这种方法能够忽略变量重命名带来的影响,除了能够快速并且较为精确地识别出 Type-1 克隆以外,还能够识别出 Type-2 克隆和少量的相似度较高的 Type-3 克隆,但对于文本相似度较低的 Type-3 克隆则几乎无法检测。其中,SourcererCC[9]和 CCAligner[10]是这类方法中比较典型的工具。

1.3 基于抽象语法树的方法

基于抽象语法树(Abstract Syntax Tree,AST)的代码克隆检测方法主要考虑源代码的语法规则,将源代码转换为其对应的 AST 形式,然后通过寻找 AST 之间的相似性来寻找克隆。Deckard[11]是其中较为典型的工具,其主要思想是通过将 AST 转换为特征向量的形式来快算寻找 AST 之间的相似性。但由于 Type-3 克隆的 AST 结构之间往往差异较大,并且子树的定位较为复杂,因此其召回率并不高。

1.4 基于程序依赖图的方法

基于 PDG 的代码克隆检测方法首先利用分析工具来生成源代码的程序依赖图,再利用图匹配算法来寻找相似的图结构,相似或者同构的 PDG 对将被视为克隆。比较典型的方法有 CCSharp 和 CCGraph,其中 CCSharp 采用了精确子图同构判定算法 VF2[12];而 CCGraph 则采用了基于 WL 图核的近似图匹配算法。然而这两种方法都没有考虑到高级别克隆对 PDG 之间的特点,因此召回率偏低。

2 方法设计与实现

本文所提出的检测方法主要由两个部分组成,即预处理阶段和克隆检测阶段。如图 1 所示,预处理阶段首先生成了与源代码相关的函数级别的 PDG,然后利用解析工具将每个顶点所代表的语义解析为特征向量的形式,最后利用滑动窗口算法对所有的 PDG 建立一个索引表,具有相同 Key 值的

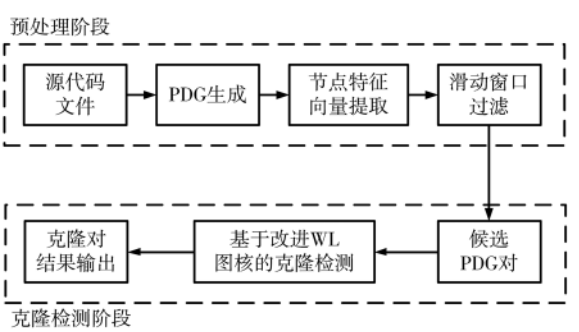


图 1 本文代码克隆检测方法流程图

PDG 将被视为潜在的克隆对进入到下一个阶段。在克隆检测阶段主要应用改进的 WL 图核算法进行图相似性的比较,判定其是否为克隆对。

2.1 节点特征向量提取

由于解析工具生成的原始 PDG 中,图节点的标签被定义为与之相对应的源代码,不利于直接进行图核计算,而如果只将其简单地按照语句类型(如赋值语句、条件判断语句等)进行处理则会损失掉大量有用信息,因此需要对其进行解析,如提取出不同变量个数、赋值运算符个数、加减乘除运算符个数、比较运算符个数、数组访问运算符个数和不同函数调用个数等语义信息(如表 1 所示),最终以特征向量的形式作为该节点的初始标签。

表 1 语义信息类型

数据类型	含义
NumOfID	不同变量的个数
AssOp	赋值运算符
LogOP	逻辑运算符
BitOP	位运算符
CmpOP	比较运算符
ShfOP	移位运算符
AsOP	加减运算符
MdOP	乘除运算符
AAOP	数组访问
FunCall	函数调用

例如,假设某个 PDG 节点所包含源码信息为“OutputStream zOut = new OutputStream(out)”,则该节点的初始标签为[2,1,0,0,0,0,0,0,0,1],表示在该节点中共有 2 个变量(分别为“zOut”和“out”)且该节点是一个赋值节点,同时还进行了一次函数调用。

2.2 滑动窗口过滤

如算法 1 所示,在获得所有节点的特征向量之后,将这些特征向量按照其对应顶点在程序流程中出现的先后顺序排序,再利用窗口大小为 w,步长为 1 的滑动窗口在这些排好序的节点上滑动。假设节点的个数为 N,则将获得 $N-w+1$ 个标签(向量)序列。然后,对这 $N-w+1$ 个序列做哈希,将哈希值作为索引,对应的 PDG 序号和相关信息作为值插入到索引表中。例如,假设 PDG 的序号为 P_1,相应的排好序的标签为{a,b,c,d},窗口大小为 3,则索引表的内容为{hash(a,b,c):[P_1],hash(b,c,d):[P_1]}。最后通过遍历索引表来获取所有的候选克隆对。

算法 1:滑动窗口过滤算法

输入:所有的待检测 PDG 集合 G,窗口大小 w;

输出:候选克隆对 C。

```
Begin
1  hashSet=Ø
2  //建立索引表
3  for each program dependency graph g in G
4      sorted(g.L);  //g.L 表示图 g 的标签集合
5      n=sizeof(g.L);
6      for i=1 to n-w+1 do //w 表示窗口大小
7          currWindow=[l_i, l_{i+1}, ⋯, l_{i+w-1}];
8                          //其中 l_i 表示第 i 个标签
9          h=hash(currWindow);
10         if hashSet.find(h) then
11             hashSet[h].value ∪ g;
12         else
13             hashSet.insert(h, g);
14         end if
15     end for
16 end for
17//获取候选克隆对
18 for each hash bucket B in hashSet do
19     if sizeof(B)≥2 then
20         for each combination pair P in B.value do
21             C=C ∪ P;
22         end for
23     end if
24 end for
25 return C
End
```

2.3 基于改进 WL 图核的克隆检测

原始的 WL 图核迭代过程主要分为 3 步:

(1)对于每个节点,获取其所有的直接邻居节点的标签从而得到一个多重标签;

(2)对每一个多重标签集合进行排序,再利用哈希算法对排序后的多重标签集合进行哈希;

(3)将得到的哈希值作为该节点的新标签进行下一次迭代。

从 WL 图核的迭代过程中不难看出,两个标签集合即使只相差一个元素,在迭代步骤(2)进行哈希后得到的标签也会完全不同,而随着迭代次数的

增加，这些差异将会随着迭代影响到越来越多的节点，最终导致原本相似的两个 PDG 被判定为不相似。

针对该问题，本文提出改进的 WL 图核，利用局部敏感哈希的特点，使得迭代过程中差异较小的子结构能够被识别出来。同时，为了降低精度的损失，本文还在子结构匹配过程中引入了向量相似度量方法。

以图 2 中的两个源代码片段为例，其生成的 PDG 如图 3 所示。其中，虚线表示控制流，实线表示数据流。在预处理阶段，已经将节点对应的源码解析为特征向量的形式，如图 4(a)和图 4(b)所示。改进WL 图核算法每次迭代的过程同样也分为 3 个步骤：

(1)将当前节点的邻居节点特征向量信息聚集到当前节点中作为新的特征向量；

(2)根据局部敏感哈希函数以及该节点的类型和上一次迭代的分类情况，对得到的新特征向量进行进一步划分，即：具有相同或相近哈希值并且上一次迭代中被划分为同一类的节点将被视为潜在的相似节点；

(3)计算两张图中相似节点对（即具有相同哈希值并且特征向量的余弦相似度超过指定阈值的节点对）的个数作为本次迭代的图核值。

```
public void write1() {
    byte[] data = new byte[10000];
    int inByte = in.read(data);
    while (inByte != -1) {
        zOut.write(data, 0, inByte);
        inByte = in.read(data);
    }
}
```

(a)示例代码 1

```
public void write2() {
    byte[] data = new byte[1024];
    int buff = 0;
    while (0 <(buff = fis.read(buffer))) {
        out.write(buffer, 0, buff);
    }
}
```

(b)示例代码 2

图 2 示例代码片段

仍然以图 4 为例，假设只进行一次迭代。迭代之前，假设节点“<3>”“<12>”和“<13>”被归为同一类，节点“<4>”和“<7>”为一类，节点“<6>”和“<15>”为一类。经过迭代步骤(2)后，由于节点“<13>”与节点“<3>”和“<12>”的特征向量相差较大，因此有很大的概率被判定为不相似。而其余节点对之间因为特征向量依旧非常接近，因此将有非常大的可能性仍然被判定为相似(即“<3>”和“<12>”相似，“<4>”和“<7>”相似，“<6>”和“<15>”相似)，此时本次迭代的图核值为 $k=2$。而如果采用原始的 WL 图核，迭代后图中的所有节点将会拥有不同的标签，即本次迭代的图核值为 0。

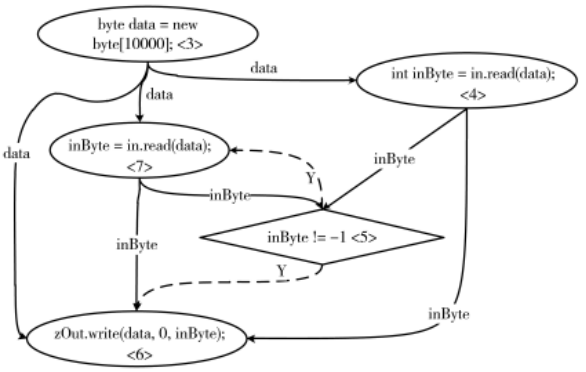
最终，总的改进 WL 图核值定义如下：

$$k^{(h)}=\alpha_1 k^{(1)}+\alpha_1 k^{(2)}+\cdots+\alpha_h k^{(h)} \tag{1}$$

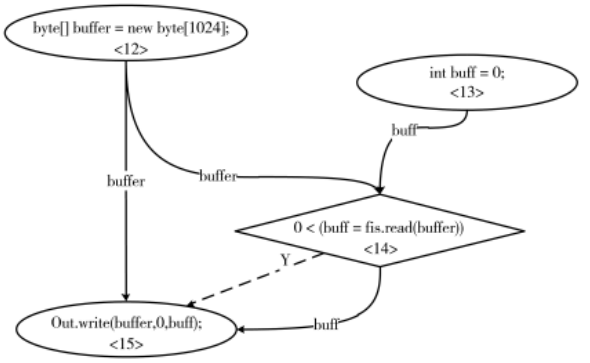
其中 $k^{(i)}$表示第 i 次迭代的图核值，h 表示迭代的总次数，α_i 为第 i 次迭代的权重因子。

在实际应用过程中，由于改进 WL 图核每次迭代都可以看做是将不同距离的节点标签信息汇聚到当前节点中，例如第 i 次迭代获取了与当前节点的最短路径距离为 i 的节点的标签信息，因此对所有的 PDG 设置一个太大的迭代次数反而会影响到时间性能，除此之外，相距越远的节点对当前节点的影响也应该越小，为此，本文在计算两个 PDG 之间的图核值时动态地设置迭代次数为 $(|V_1|+|V_2|)/2$（其中$|V|$表示 PDG 中节点的总数），并在每次迭代中加入权重因子 $\alpha_i=(h-i+1)/h$ 来给予距离较近的邻居节点信息更高的权重。

最后，考虑到差异较大的 Type-3 克隆的 PDG 中通常只存在一定的局部相似性，因此本文采用了非

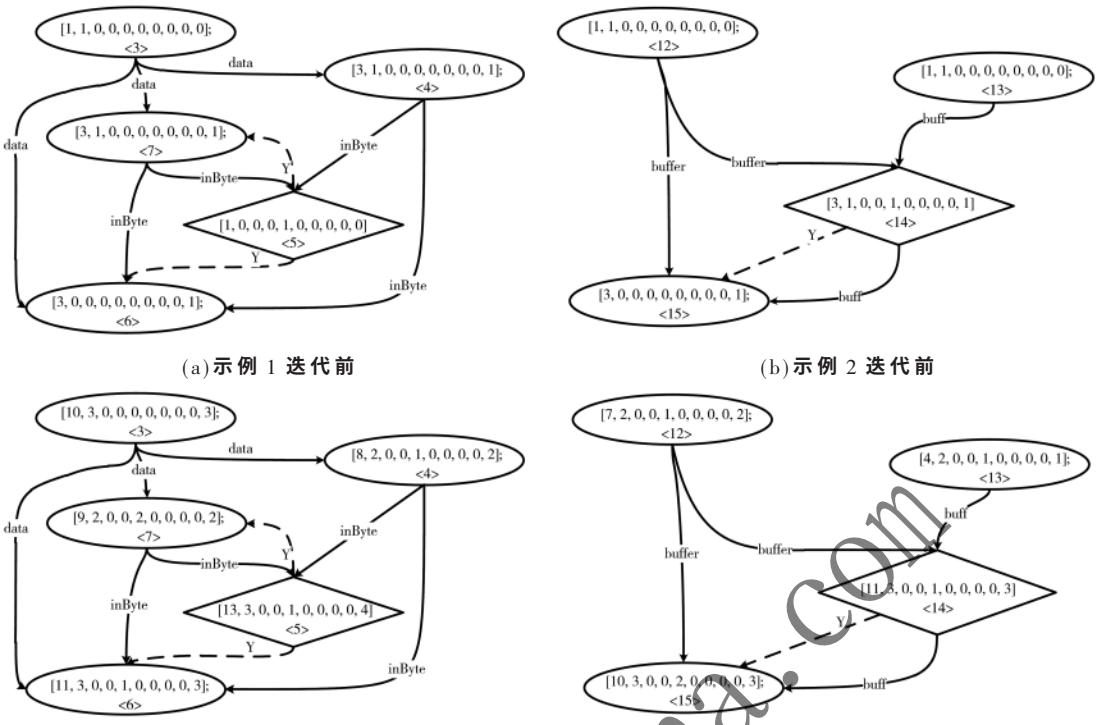


(a)代码片段 1 的程序依赖图



(b)代码片段 2 的程序依赖图

图 3 示例代码片段的程序依赖图



(a)示例 1 迭代前　(b)示例 2 迭代前

(c)示例 1 迭代后　(d)示例 2 迭代后

图 4　改进 WL 图核的一次迭代过程示例

对称相似性系数来衡量两个 PDG 之间的相似程度,其计算方式如下:

$$\mathrm{sim}(G_1, G_2)=\frac{k^{(h)}(G_1, G_2)}{\min(|V_1|, |V_2|)} \tag{2}$$

其中 G_1, G_2 表示 PDG;$k^{(h)}(G_1, G_2)$表示这两个 PDG 的改进 WL 图核值;$|V_1|, |V_2|$表示 PDG 的顶点个数。最终,相似度超过阈值的 PDG 对将被视为克隆。

3 实验结果与分析

3.1 数据集与实验配置

在验证工具的有效性方面,本文选用了公开的克隆检测数据集 BigCloneBench[13]进行实验。该数据集由加拿大的 Svajlenko 团队构造,从 IJaDataset-2.0 中选取了总共 55 499 个 Java 文件,并标记出其中的 8 584 153 个真实克隆对和 279 032 个假克隆对,是学术界用于评估检测工具的常用数据集之一。

为了评估工具的可扩展性,在从 IJaDataset-2.0 数据集中随机选取出来的 1k、10k、1M 和 10M LoC (Lines of Code)的规模数据集分别进行了实验。

实验环境为 Ubuntu 18.04 系统,Intel® Xeon® Gold 5120 2.20 GHz CPU,503 GB 内存空间。

3.2 评估标准与结果分析

本文主要用精确率、召回率和时间性能三个指标来作为评估标准。在评估召回率方面主要采用了 BigCloneEval[14]评估框架。BigCloneEval 是基于 BigCloneBench 数据集的自动化评估工具,能够根据检测工具所报告的克隆对自动计算出相应的召回率。同时,BigCloneEval 还将 Type-3 克隆按照源代码的相似程度分为了 Very Strongly Type-3(即源代码相似度在 90%~100%之间)、Strongly Type-3(即源代码相似度在 70%~90%之间)、Moderately Type-3(即源代码相似度在 50%~70%之间)和 Weakly Type-3/Type-4(即源代码相似度在 0%~50%之间)。

在对比工具方面,分别选择了各种类型的代码克隆检测工具中比较具有代表性的方法,包括基于 Token 的检测方法 CCAligner、SourcererCC,基于 AST 的检测方法 Deckard,基于深度学习的方法 Oreo[15]以及基于 PDG 的方法 CCGraph。由于 CCSharp 不支持 Java 语言的检测,因此未进行对比。最终的实验结果如表 2 所示。

在召回率方面,本文方法在除了 ST3(Strongly Type-3)克隆以外的所有克隆类型上的召回率明显优于采用原始 WL 图核的 CCGraph。在源代码相似程度最低的 MT3(Moderately Type-3)上召回率取得了最好的结果,并且在 T1(Type-1)、T2(Type-2)和

表 2 不同工具在 BigCloneBench 上的检测结果

检测工具	召回率/%					精确率/%
	T1	T2	VST3	ST3	MT3	
本文方法	99	99	95	75	44	80
CCGraph	96	82	75	77	35	79
SourcererCC	100	98	93	61	5	99
Deckard	60	58	62	31	12	35
CCAligner	100	99	97	70	10	80
Oreo	100	99	100	89	30	89

VST3(Very Strongly Type-3)克隆的召回率与基于 Token 方法的 CCAligner 和 SourcererCC 非常接近,均达到了接近 100%的召回率。Type-1 和 Type-2 克隆召回率未达到 100%的主要原因是由于 PDG 生成工具自身存在的问题,致使某些程序未能正确生成相应的 PDG。

在精确率方面,由于 BigCloneEval 评估工具只报告方法的召回率,而检测得到的克隆对数又较大,因此本文从检测结果中随机选取了 400 对克隆进行人工确认,结果如表 2 中所示。从表 2 的结果可以看到,精度最高的工具为 SourcererCC,但其 MT3 克隆的召回率只有 5%,而 MT3 克隆往往是假阳性克隆对的主要来源之一。本文方法略高于 CCGraph,这是因为预处理阶段将 PDG 节点中所包含的源代码信息解析为特征向量的形式保留了更多的语义信息。

在对结果的进一步分析中发现,由于局部敏感哈希函数依然存在一个较小的可能性将特征向量相差较大的节点归为同一类,从而导致了假阳性的产生。实际应用中可以根据用户需求通过调整相似度阈值以及多次哈希取并集的方式来平衡召回率和精确率。

在评估时间性能方面,本文主要对比了同样基于 PDG 的克隆检测工具。由于 CCSharp 不支持 Java 数据集的检测,因此为了体现采用图核方法与采用精确子图同构检测方法的性能差距,本文加入了将 WL 图核换成 CCShrap 中所采用的 VF2 算法进行比对。通过表 3 的实验结果可以看出,与采用精确子图同构检测的 VF2 算法相比,采用近似图匹配的本文方法和 CCGraph 在时间性能方面得到了巨大的提升。但由于本文方法在预处理阶段增加了节点特征向量提取的步骤,因此时间性能上比 CCGraph 略差,但依然在可接受范围内。

表 3 算法运行时间比较

检测方法	有效代码行数			
	1k	10k	1M	10M
本文方法	2 s	6 s	18 m 25 s	2 h 59 m 2 s
本文方法-VF2	4 s	14 s	1 h 24 m 16 s	13 h 2 m 33 s
CCGraph	2 s	4 s	15 m 54 s	2 h 21 m 10 s

4 结论

本文针对 WL 图核在 PDG 近似匹配过程中存在的问题,提出了一种改进思路:在迭代过程中将普通哈希算法替换为局部敏感哈希算法,同时参考上一次迭代的结果来计算图核值。此外,本文还根据改进后的 WL 图核方法设计并实现了相应的克隆检测工具:首先利用静态解析工具将源代码解析为特征向量的形式,然后根据程序流程的先后顺序对这些特征向量排序,利用滑动窗口算法建立索引表并过滤掉那些不可能为克隆的 PDG 对;最后利用改进的 WL 图核方法对候选 PDG 对进行图相似性检测并输出最后的结果。实验结果表明,本文方法在 Type-3 克隆上的召回率得到了较大提升,同时在检测精度和时间性能上得到了良好的保持。

参考文献

[1] 乐乔艺,刘建勋,孙晓平,等.代码克隆检测研究进展综述[J].计算机科学,2021,48(S2):509-522.

[2] MONDAL M,RAHMAN M S,SAHA R K,et al.An empirical study of the impacts of clones in software maintenance[C]//2011 IEEE 19th International Conference on Program Comprehension.IEEE,2011:242-245.

[3] ISLAM J F,MONDAL M,ROY C K.Bug replication in code clones:an empirical study[C]//2016 IEEE 23rd International Conference on Software Analysis,Evolution,and Reengineering(SANER).IEEE,2016,1:68-78.

[4] MONDAL M,ROY C K,SCHNEIDER K A.A comparative study on the bug-proneness of different types of code clones[C]//2015 IEEE International Conference on Software Maintenance and Evolution(ICSME).IEEE,2015:91-100.

[5] WANG M,WANG P C,XU Y.CCSharp:an efficient three-phase code clone detector using modified PDGs[C]//2017 24th Asia-Pacific Software Engineering Conference(APSEC).IEEE,2017:100-109.

[6] LIU C, CHEN C, HAN J W, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C]//Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006: 872-881.

[7] ZOU Y, BAN B, XUE Y, et al. CCGraph: a PDG-based code clone detector with approximate graph matching[C]// 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020: 931-942.

[8] BAKER B S. On finding duplication and near-duplication in large software systems[C]//Proceedings of 2nd Working Conference on Reverse Engineering. IEEE, 1995: 86-95.

[9] SAJNANI H, SAINI V, SVAJLENKO J, et al. SourcererCC: scaling code clone detection to big-code[C]//Proceedings of the 38th International Conference on Software Engineering. IEEE, 2016: 1157-1168.

[10] WANG P C, SVAJLENKO J, WU Y Z, et al. CCAligner: a token based large-gap clone detector[C]//Proceedings of the 40th International Conference on Software Engineering, 2018: 1066-1077.

[11] JIANG L X, MISHERGHI G, SU Z D, et al. Deckard: scalable and accurate tree-based detection of code clones[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 96-105.

[12] CORDELLA L P, FOGGIA P, SANSONE C, et al. A (sub) graph isomorphism algorithm for matching large graphs[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004, 26(10): 1367-1372.

[13] SVAJLENKO J, ROY C K. Evaluating clone detection tools with bigclonebench[C]//2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2015: 131-140.

[14] SVAJLENKO J, ROY C K. Bigcloneeval: a clone detection tool evaluation framework with bigclonebench[C]//2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2016: 596-600.

[15] SAINI V, FARMAHINIFARAHANI F, LU Y, et al. Oreo: detection of clones in the twilight zone[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018: 354-365.

(收稿日期:2022-04-23)

作者简介:

班必奂(1993-),男,硕士,主要研究方向:代码克隆检测。

徐云(1960-),通信作者,男,博士,教授,主要研究方向:并行计算、生物信息学、区块链技术、代码克隆检测。E-mail: xuyun@ustc.edu.cn。

版权声明

凡《网络安全与数据治理》录用的文章，如作者没有关于汇编权、翻译权、印刷权及电子版的复制权、信息网络传播权与发行权等版权的特殊声明，即视作该文章署名作者同意将该文章的汇编权、翻译权、印刷权及电子版的复制权、信息网络传播权与发行权授予本刊，本刊有权授权本刊合作数据库、合作媒体等合作伙伴使用。同时，本刊支付的稿酬已包含上述使用的费用，特此声明。

《网络安全与数据治理》编辑部