基于路径序列相似度判别的程序克隆检测方法

吕博然,吴军华
LV Boran, WU Junhua

南京工业大学 计算机科学与技术学院,南京 211800
School of Computer Science and Technology, Nanjing University of Technology, Nanjing 211800, China

LV Boran, WU Junhua. Clone detection method based on path sequence similarity determination. Computer Engineering and Applications, 2018, 54(2):55-61.

Abstract: Code clone is a common phenomenon in the software system. The program code is converted to path execution sequence constituted by program nodes, through static analysis, by attribute definitions of nodes in this paper, and the calculation for the similarity is solved by discrete sequence similarity detection distance, line model and sequence correlation coefficient, and similarity between different programs. The experiments and data analysis verify the feasibility of this approach.

Key words: clone detection; control flow graph; program node; execution sequence; path similarity

摘 要: 代码克隆是软件系统中常见现象。将程序代码通过静态分析,转换为由程序结点构成的路径执行序列,通过结点属性的定义,将程序代码相似度检测转化为离散序列距离,折线模型和序列相关度问题,针对上述三种模型计算不同代码执行路径间相似度,最终得出程序间克隆相似度。经过实验和数据分析,验证该方法的可行性。

关键词: 克隆检测;控制流图;程序结点;执行序列;路径相似
文献标志码: A **中图分类号:** TP311 doi:10.3778/j.issn.1002-8331.1608-0344

1 引言

克隆代码在软件开发过程及软件系统中的运用十分普遍。据相关研究,克隆代码在软件系统中的运用约占了近5%~20%。目前,不加改动地复制代码片段在开发软件中是一种常见的现象。被复制的代码就是原始代码,一般来说,在软件系统中很难辨别哪段是原始的、哪段是克隆的。所以,在软件系统中有一定程度上相似的代码,就被认为是克隆代码。克隆代码的存在给软件维护增加了困难,即使是在高质量的软件系统中,代码克隆也难以避免。当在软件维护过程中出现错误时,就必须快速地把分散在各处与其他相似代码找出来,以避免其所有发生错误的可能,对一些软件功能做调整时,也需要找出相似代码。因此,在软件系统中尽可能高质量地检测克隆代码就显得十分必要。

目前已有很多种克隆检测方法,主要包括基于文本的检测、基于token序列的检测[1]、基于AST语法树的检测[2-4],基于度量的检测、基于图的检测[5-6]等。

基于文本的检测,是从文本结构和语法的角度去分析克隆,易扩展到多种语言。但是,这种检测忽略不同语言、语法差异的行为,会使一些克隆无法检测。有部分研究,是从程序文本出发,将代码中的变量语句等内容进行规范化处理,让代码的书写风格统一化,进而在针对文本的相似度进行克隆分析[7]。

基于AST语法树的检测,是将待检测代码转化为抽象语法树。它是通过分析语言的全部语法结构,之结果对代码的语法结构进行静态分析。这种方法的代价过大,检测时间相对较长[8-9]。

基于图的检测,是将代码转换为程序依赖图,对图中结点进行分类获取匹配结点,分析结点的数据依赖和控制依赖矩阵,查找图中的同构子图并对其进行泛化处理,计算程序的差异向量距离实现克隆检测。这种检测方法对于断层克隆极为有效,但它对更深层次的功能克隆效果不明显[10]。

基于Token序列的检测,通过将源代码中的标志符、关键字、运算符等词法标识符转换成统一的内部符号,并去掉标识符的实际名称和实际值,对整个token序

作者简介: 吕博然(1992—),通讯作者,男,硕士生,主研方向为软件测试,E-mail:luboray@163.com;吴军华(1965—),女,博士,副教授。
收稿日期: 2016-08-17 **修回日期:** 2016-10-09 **文章编号:** 1002-8331(2018)02-0055-07
CNKI **网络优先出版:** 2017-02-27, http://kns.cnki.net/kcms/detail/11.2127.TP.20170227.1058.016.html

列进行切分,再把所有的token子序列构造成一棵后缀树,再对源程序文件与目标程序文件转换后的Token序列后缀树进行比对匹配分析,以实现代码克隆检测[11]。这是目前使用最广泛的代码克隆检测方法。

基于度量的检测,是通过预先设计多项属性,如环路复杂度、参数表、继承树等属性对程序进行度量。比较程序属性向量的欧式距离得到相似度。这种方法在属性向量各分量属性的设定复杂,且度量值不能代表比较单元的全部特征,只能进行固定粒度的检测,会遗漏很多克隆现象,存在属性指定少造成检测结果不准确的情况[12-13]。

本文所提出的基于路径相似度判别程序克隆检测方法,通过静态分析,将程序转化为控制流图后,得到其执行路径结点序列,通过对各序列的相似度判别,最终得出程序间的相似度值。与传统克隆检测方法相比,检测准度与之处于同一水平。在检测过程所耗费的时间有所改善,减少程序运行的时间,提高结构克隆检测效率。为验证本文算法正确性,同现有基于token序列的自由粒度匹配检测算法检测结果比较,该算法中不需要使用重量级的语法分析器,容易扩展到多种编程语言以及纯文本文档的检测,自由粒度匹配算法相比基于树的匹配算法具有更高的执行效率,对大规模软件系统有较好的检测效果。实验选用以此算法设计的CCFinder工具,它适用于多种编程语言的克隆检测[13]。

2 程序结点和路径执行序列

本文从程序代码的结构分析出发,采用路径比较进行研究分析克隆代码的问题。其主要思想为:将执行代码转换成为控制流图,并将图的连通性得到各条路径执行进行相似度比较,进而得到的一种代码相似度检测方法。该方法将程序映射成有带信息的结点构成的控制流图的模型,通过图的连通的路径和路径上的结点信息分析,从程序的结构和编程逻辑角度出发,简化程序在语法检测过程,一定程度上减少了算法复杂度。

控制流图(Control Flow Graph,CFG)是一个有向图 $G=<V,E>$,其中 V 是控制流结点的集合,E 是有向边的集合[7]。控制流图的结点表示程序的语句,边表示语句间的执行关系。令 $<x,y>\in E$,表示控制流图中的边 $x\rightarrow y$,称 x 是 y 的前驱,y 是 x 的后继[9]。

块是可以被看作为一个单元的一组连续的程序语句,总是在一起执行的最大的程序语句序列。块有两类:结点(Node)和段(Segment)。结点包括:判断、连接、程序单元的入口点和出口点。段是一段顺序执行的单入口且单出口的程序语句序列[14-15]。

在本文研究中,将块的结点和段记作为程序结点 $node(node\in V)$。控制流图(CFG)中,任意两个结点之间都是单连通的,即 $\forall x,y\in V$,若有 $<x,y>\in E$,则 $<y,x>$ 不存在[16]。

2.1 程序结点

程序结点,是指代码中由程序中各个块的首尾约束位置抽象结果。结点属性由以下几个方面构成,包括结点位置、序号、对结点序号、关键词类型和块位置。结点位置指的是,该结点处于代码文本中的位置;序号,是标记的序号值,同一模块下的结点序号唯一;对结点序号是指,块的首尾分别映射的一对结点中除本结点外的序号;对关键词类型,是块的语法关键词,将其映射为一个属性值;块位置是用来标记结点位于块首或尾。

定义1 程序结点为一个五元组,记作 $node=(p,n,bn,k,e)$。其中 p 表示结点位置;n 表示序号,其中 $n\in N$;bn 表示块对结点序号,$bn\in N$ 且 $n\neq bn$;k 表示对关键词类型,其中 $k\in\{\text{for, while, do, else, switch, case, com}\}$;$e$ 表示该结点的位于作用块的首尾位置标定,$e\in\{\text{head, end, null}\}$。

以上文中定义的结点类型为单位,将代码程序抽象为有一系列程序结点集合,记作 $nd=\{node_n,n\in N\}$,其中 $node_k$ 是程序结点类型元素。程序示例如图1所示,其中带圈数字是各个位置表示结点的标号。

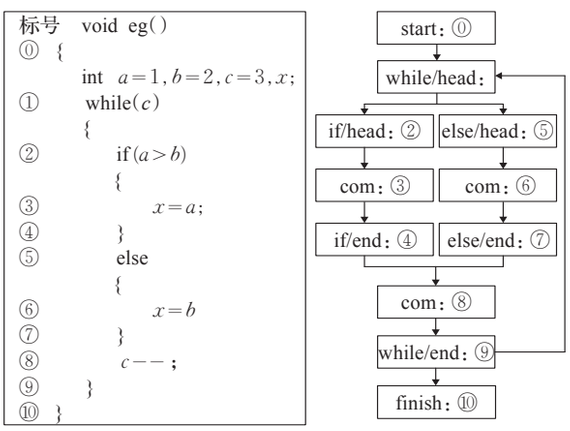

图1 程序转换结点示例

2.2 路径执行序列

路径执行序列,是指由代码映射得到的程序结点集合构成的控制流图,根据深语法规则在深度优先算法(DFS)依次遍历的程序结点元素构成的序列。

定义2(路径执行序列) 是有序集合 $P\{node_{x_r},r\in[0,len-1],r\in Z,x_{r-1}<x_r|_{r>0}\}$,其中 len 是序列长度,$node_{x_r}\in nd$,且对任意输入 r,其执行结点 $node_{x_r}\in P$。

由于路径执行序列仅仅表示一段抽象为结点代码,静态分析下的结点执行顺序,路径中并不包含整条路径的前后逻辑和代码结构的信息,因此引入前驱关系序列。

2.3 前驱关系序列

前驱关系序列,是路径执行序列中各结点与其前驱

结点的关键词类型和块位置属性组合映射值构成的序列。该序列的每一个值与原路径执行序列中元素顺序上是一一对应的关系。路径执行序列 P 中，$\forall node_{x_r} \in P$，若 $\exists node_{x_{r-1}} \in P$，那么 $node_{x_{r-1}}$ 是 $node_{x_r}$ 的前驱结点。

定义 3（前驱关系序列）是根据路径执行序列中当前结点与其前驱结点属性的映射值构成的有序集合，记作 $S\{v_r, r \in [0, len-1], r \in Z\}$。

对于路径执行序列的有序集合 $P\{node_{x_r}, r \in [0, len-1], r \in Z\}$，对应有前驱关系序列 $v_r = f\left(node_{x_{r-1}}(k,e), node_{x_r}(k,e)\right)$，$node_{x_r}(k_{x_r}, e_{x_r})$ 是结点关键词类型和块位置属性构成的二元组，映射规则 f 建立如表1，表1中值自行定义。

表1　前驱关系映射值示例

$node_{xr}(k,e)$	$node_{xr-1}(k,e)$	v_r
null	（if，head）	t_0
（if，head）	（com，null）	t_1
…	…	…

如图2所示，得到为有序集合的前驱关系序列，记为 $S'\{t_0, t_1, \cdots, t_5\}$。

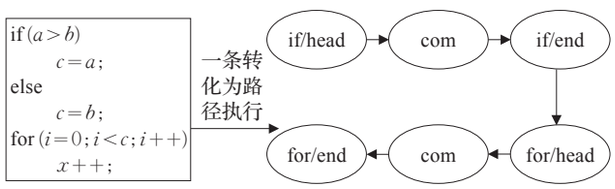

图2　程序转换路径执行序列示例

前驱关系序列，不仅具有路径执行上各个结点的属性，同时还含有结点间的逻辑关系信息以及程序路径的逻辑结构信息。前驱关系序列的元素值也反映了路径序列上不同类型结点变化带来的运算复杂度。之后采用的相似度度量计算，都是基于前驱关系序列进行。

3　相似度度量方法

相似度度量是根据路径执行序列映射得到的前驱关系序列计算。通过计算序列编辑相似度和序列折线相似度来实现。执行序列相似度，是执行序列映射的前驱关系序列间序列编辑相似度和序列折现相似度平均值。

定义 4（执行序列相似度）

$$Similary(P_1, P_2) = w_m \cdot m(S_1, S_2) + w_r \cdot r(S_1, S_2) \quad (1)$$

其中 P_1、P_2 是执行序列，S_1、S_2 是 P_1、P_2 对应的前驱关系序列。$m(S_1, S_2)$ 是序列距离相似度，$r(S_1, S_2)$ 是序列折线相关系数，w_m、w_r 是权重，取 $w_m = \frac{1}{3}$，$w_r = \frac{1}{3}$。

3.1　序列距离相似度

序列距离相似度是两组序列的编辑相似度与折线相似度的均值。记作：

$$m(S_1, S_2) = \frac{1}{2}\left[edit(S_1, S_2) + dist(S_1, S_2)\right] \quad (2)$$

其中 $edit(S_1, S_2)$、$dist(S_1, S_2)$ 分别是序列 S_1, S_2 的编辑相似度和折线相似度值。编辑相似度是指两组序列在编辑操作上差异的度量，折线相似度是序列转换为折线在几何上相似的度量。

3.2　序列编辑相似度

序列的构成是由一组元素按照一定顺序规则排列得到的数组，而字符串序列同样是由单个字符排列组成数组。字符串间的编辑距离计算过程，是针对序列进行插入删除移动单个字符的操作进行，序列也具有同样的性质和操作方法。

定义 5（编辑相似度）是指两段序列间的序列编辑距离与序列中较长长度的比值。其中编辑距离，是指两个序列中，一序列变换为另一序列，进行插入、移动和替换结点操作的次数，记作 et。

两组前驱关系序列 S_1、S_2 的编辑相似度：

$$edit(S_1, S_2) = \frac{et}{\max len(S_1, S_2)} \quad (3)$$

其中 $\max len$ 是求解两组序列中最长长度的方法。

3.3　序列折线相似度

序列折线，是将前驱关系序列中各元素的值，映射到二维图标上离散点连线后得到的折线。序列折线的相似度就是两组前驱关系序列的相似度。

定义 6（序列折线）是有前驱关系序列通过一定的映射规则，得到的离散点集构成的连线。映射折线记作 $L\{(n, value_n), n \in [0, len-1], n \in Z\}$，$len$ 是原前驱关系序列 S 长度，$value_n = \sum_{i=0}^{n} S[i]$。

在计算路径执行的相似度时，需要将路径执行序列转换为前驱关系序列，前驱关系序列转换为序列折线，针对不同路径执行转换后的折线进行相似度计算。通过代码程序转换成几何上的比较，如图3所示。序列折线计算相似度，各序列的长度相等。

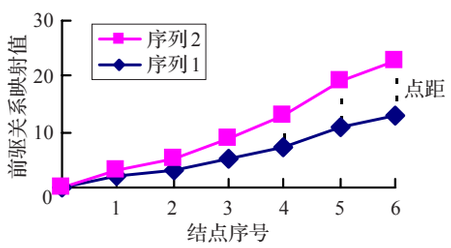

图3　序列折线示意图

序列折线的相似度，是折线上各点距值（如图3中所示，各离散点的高低点连线）的标准差，与最大点距值的比值。点距值，是两组序列对应折线上，同横坐标的离散点对应纵坐标差值的绝对值，所构成的全部各点距

值有序集合记作：$d\left\{d_n=\left|value_x\right|_{L_1,x=n}-value_x\right|_{L_2,x=n}\Big|,n\in[0,len-1]\right\}$，其中 L_1、L_2 是两组比较折线。

定义 7(折线相似度)

$$simline(L_1,L_2)=1-\frac{S(d)}{d_{\max}} \tag{4}$$

其中 $S(d)$ 是点距集合 d 的标准差，$d_{\max}$ 是集合 d 中的元素最大值。L_1、L_2 是由前驱关系序列 S_1、S_2 映射在二维坐标图的折线。因此有下式成立：

$$dist(S_1,S_2)=simline(L_1,L_2)$$

两段等长序列映射后的折线，对应结点之间前驱关系值间的差值，表示两段路径映射成的折线间距。如果间距值都接近，表明折线在变化趋势上接近，那么在路径执行序列认为相似度较高。

3.4　序列折线相关系数

两组等长的路径执行序列转换为前驱关系序列间，计算相关系数，有 $r(S_1,S_2)=r(L_1,L_2)$ 成立。相关系数反映了两组路径执行序列存在相互联系的程度，相关系数越大，路径执行序列间对应的折线间相关度越高，路径间的相似度也越高。

4　算法改善

为了消除进行比较的路径长度可能存在一定的差异性，两种预处理的方式都为在不破坏原序列结构的前提下，得到两段长度相等的序列，便于计算。滑动窗口的处理用来两段序列长度差别较大；序列插值在序列长度较为接近。

4.1　滑动窗口

滑动窗口是在两段序列长度差异较大的情况下使用的预处理方法。目的是从较长的序列中截取出与短序列等长的序列片段后，再比较的方法。这是一种局部比较的方法。

两段序列中取较长的作为轨道序列，较短的作为窗口序列，长度记为窗口长度。在轨道序列中从序列初始元素开始，截取所有的连续且长度为窗口长度的序列片段，与窗口序列进行相似度计算。序列长度值较小与较大的比值作为长度系数。图4所示，为滑动窗口的实现方法。其中单个的正方形表示序列中的结点。

两段序列的相似度值，取在滑动计算过程中得到最大值与长度系数的乘积作为结果。两组序列采用滑动窗口比较时，由于轨道序列在不同窗口下参与比较项不同。因此在全部的结果中取其中最大值作为滑动窗口的最终相似度度量结果。

4.2　序列插值

序列插值是针对两段长度相近的序列进行的预处理。对两段序列中较短的，在不破坏原序列结构的前提下，将特定元素插入原序列的指定位置，得到两段等长序列的预处理方法。如图5所示序列长度值较小与较大的比值作为长度系数。

两段序列的相似度值取差值计算结果与长度系数乘积。

4.3　处理方法选择

定义 8　长度系数是序列长度值较小与较大的比值，记作 K：

$$K=\frac{\min len(W_1,W_2)}{\max len(W_1,W_2)} \tag{5}$$

其中 W_1、W_2 是两组序列，$\min len$ 、$\max len$ 分别是求出序列对中较短与较长长度的方法。

当 $K\leqslant 0.7$，采用滑动窗口的处理方式，计算距离相似度值和相关系数。取滑动比较过程中的最大值为比较结果 re。

当 $K>0.7$，未对短序列插值时，先采用滑动窗口得出的数值，与直接计算距离相似度值取较大值作为结果，记作 re_{sld}。再对短序列插值后，只计算距折线相似

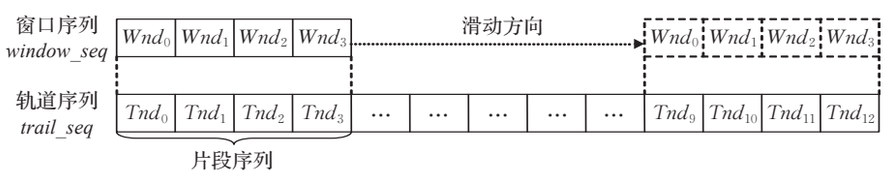

图4　滑动窗口示意图

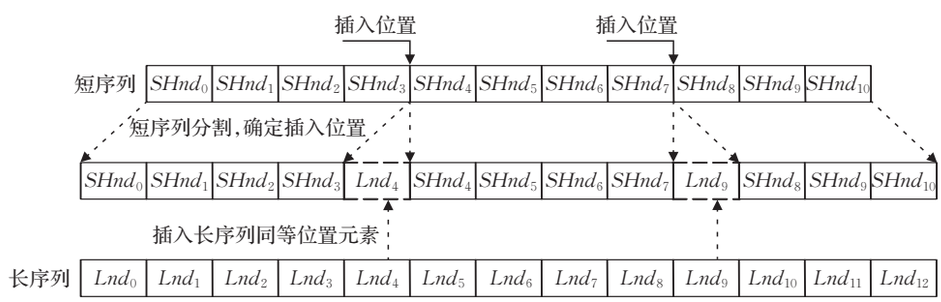

图5　插值过程示意图

度作为距离相似度值，与相关系数作为结果，记作 re_{add}。取插值前后相似度结果的较大值作为比较结果为 $re=\max(re_{sld},re_{add})$。

具体方法如表2所示。

表2　处理方案

K	$\frac{m(S_1,S_2)}{edit(S_1,S_2)}$	$\frac{m(S_1,S_2)}{dist(S_1,S_2)}$	$r(S_1,S_2)$	计算结果	比较结果
$\leqslant 0.7$	√	滑动窗口 √	√	re	re
>0.7	√	未插值，滑动窗口 √	√	re_{sld}	re_{sld}，re_{add} 的均值
	×	插值 √	√	re_{add}	

两组序列相似度是比较结果与长度系数 K 的乘积。

5　代码相似度量

两个代码文件的相似度度量，是先将代码拆分成可单独执行的程序模块。从程序不同模块中的路径间开始计算相似度，再得出模块的相似度，最后综合结果得到代码文件之间的相似度值。

假设有两个模块，一个设为目标模块 $dest\_mod$，其中包含有 M 条路径所构成集合 $dest\_path=\{dest\_path_i, i\in[0,M-1], i\in Z\}$；另一个为比较模块 cmp_mod，其中包含有 N 条比较路径集合 $cmp\_path=\{cmp\_path_j, j\in[0,N-1], j\in Z\}$。

5.1　单目标—多比较路径相似度量

计算 $dest\_path$ 的相似度 $path\_result$。

步骤1　取 $dest\_path_i$ 与 cmp_path_j 分别求相似度 $Similarity(dest_path_i,cmp_path_j)$的结果，记为 cmp_result_{ij}。

步骤2　计算 cmp_result_j 的路径复杂度，依次记为 $pathcplx_j$，作为权重。

定义9（复杂度）路径执行序列是每个结点 k 属性赋予的值 c_k 之积，为此记作 $cplx(path)=\prod_{k=0}^{len}c_k$，$len$ 是当前 cmp_path_j 序列长度。

步骤3　$dest\_path_i$ 的路径相似度算式为：

$$cmp\_path\_sim_i=\frac{\sum_{j=0}^{N-1}cmp\_result_{ij}\times cplx(cmp\_path_j)}{\sum_{j=0}^{N-1}cplx(cmp\_path_j)} \tag{6}$$

步骤4　依次计算 $dest\_path_i$ 的路径相似度，记为 $dest\_path\_sim_i$。

以上得出目标模块中的单条路径执行的相似度值。

5.2　单目标—单比较模块相似度量

计算模块的相似度 $dest\_mod\_sim$。

步骤5　目标模块的相似度通过本模块中各条路径的相似度综合得出结果，算式如下：

$$dest\_mod\_sim=\frac{\sum_{i=0}^{M-1}dest\_path\_sim_i\times cplx(dest\_path_i)}{\sum_{i=0}^{M-1}cplx(dest\_path_i)} \tag{7}$$

以上得出两个模块之间的相似度值。

5.3　单目标—多比较模块相似度量

假设两个代码文件中，设一个为目标文件 $dest\_file$，其中由 P 个模块构成集合 $dest\_mod=\{dest\_mod_n, u\in[0,P-1], u\in Z\}$，另一个设为比较文件 $dest\_file$，其中有 Q 个模块构成集合 $cmp\_mod=\{cmp\_mod_v, v\in[0,Q-1], v\in Z\}$。

步骤6　通过上文所述方法，取目标文件中模块集合中元素 $dest\_mod_u$，依次与 cmp_mod 集合元素 cmp_mod_v 相似度计算，得到结果依次记为 $cmp\_mod\_sim_{uv}$。

步骤7　以各个比较模块的路径数目为权重，记为 $cmp\_mod\_pathnum_v$。

步骤8　计算单个目标模块相似度：

$$dest\_mod\_result_u=\frac{\sum_{v=0}^{Q-1}cmp\_mod\_result_{uv}\times cmp\_pathnum_v}{\sum_{v=0}^{Q-1}cmp\_pathnum_v} \tag{8}$$

以此类推计算 $dest\_mod$ 中各目标模块相似度值，记为 $dest\_mod\_result_u$。

5.4　代码文件相似度

步骤9　计算目标代码文件的相似度，以 $dest\_mod$ 中各模块的路径数目为权重，记为 $dest\_mod\_pathnum_u$。目标代码文件与比较代码文件的综合最终相似度的算式为：

$$dest\_file\_result=\frac{\sum_{u=0}^{P-1}dest\_mod\_result_u\times dest\_pathnum_u}{\sum_{u=0}^{P-1}dest\_pathnum_u} \tag{9}$$

通过上述的算法最后得到两个代码文件之间的克隆相似度值，记作 R。

6　实验结果与分析

6.1　实验设计与数据

实验采用20组代码文件作为测试用例。验证组选取15个JDK1.5中常用类库，包括了常用输入输出、文本字符、网络接口、图形接口和数据库等功能实现，如表3所示，余下5组选取C/C++程序库中与验证组实现功能

相近类库,包括基本输入输出、字符串操作等库作为对照组,如表4所示。以上20组库都为Java和C/C++开发进程中广泛使用的基础类库,有着良好的复用性和扩展性,库中代码总量在15万行以上,共约1 100个文件,代码数量上保证了足够的测试样本用例。并且其中包含了多数常用的基础算法、数据结构、句柄和常用类定义及相关方法的实现代码,在词法结构、逻辑流程和控制结构上具有代表性。

表3 验证组选用类库

No.	类库	No.	类库	No.	类库
1	java.io	6	java.applet	11	java.security
2	java.naming	7	java.awt	12	java.sql
3	java.lang	8	java.awt.image	13	java.rmi
4	java.math	9	javax.swing	14	java.util
5	java.text	10	java.net	15	java.corba

表4 对照组选用类库

No.	类库
16	iostream.h
17	string.h
18	math.h
19	sqlca.h
20	socket.h

先对验证组中15组类库检测相似度值,再依次取对照组中代码样本与验证组全体代码进行测试,依次得到对照组中每个库与全体验证组的相似度,验证在不同编程语言条件下,实现相同功能的程序之间本文设计算法对其中代码和结构克隆的检测效果。

方法对比实验设计为与CCFinder检测工具的运行结果作比较,CCFinder是一种基于token序列的自由粒度匹配算法[11],原理上与本文方法有本质上区别,所以这里主要是为了验证本文方法的有效性。本文算法检测实验结果如表5所示。

表5 本文方法实验数据

验证组						对照组	
No.	相似度	No.	相似度	No.	相似度	No.	相似度
1	0.279	6	0.139	11	0.230	16	0.481
2	0.247	7	0.183	12	0.327	17	0.769
3	0.361	8	0.316	13	0.228	18	0.863
4	0.325	9	0.104	14	0.114	19	0.781
5	0.202	10	0.166	15	0.092	20	0.615

采用CCFinder检测工具对同一组测试用例进行检测,所得结果在表6中所示。

表6 CCFinder对比数据

验证组						对照组	
No.	相似度	No.	相似度	No.	相似度	No.	相似度
1	0.315	6	0.152	11	0.266	16	0.531
2	0.273	7	0.225	12	0.351	17	0.778
3	0.391	8	0.351	13	0.241	18	0.885
4	0.374	9	0.124	14	0.135	19	0.791
5	0.239	10	0.219	15	0.116	20	0.632

为检测对本文设计算法的运行效率,另从JDK1.5和C++标准库(SL)中各取7组类作为实验样本,每组中类库数量随机,并将每组中类库作为整体分别使用本文方法与CCFinder工具对同一组内部检测克隆执行时间上比较,耗时数据如表7所示。

6.2 结果分析

通过对表5中数据分析,在验证组中,虽然用的是同种编程语言,但在类库实现不同功能时采用的结构逻辑定义等方面各不相同,总体相似度偏低。度量中出现重合的部分大多是类库间某些类或者某些相似方法重复定义,或是其中某些方法实现在对应控制流图结构和其中变量定义上的类似,同时库中包含如set/get组成的构造析构函数,或是数据库连接过程中相同的代码实现流程结构和句柄代码,这些造成了程序结构和代码上的克隆。对照组类库的实现功能与验证组中的一些类似,例如java.math和math.h、sqlca.h和java.sql以及java.net和socket.h的相似度值,每对虽然采用不同编程语言,但都是对数学运算函数、数据库连接操作和网络套接字连接的功能实现,在流程、类定义、算法等功能的底层实现上非常接近,所以在与验证组执行克隆检测时,两者出现了较高的相似度值。本文设计的算法针对结点控制流图模型中各条执行路径结点序列的对比计算相似度,如math.h和java.math、string.h和java.text中相同同功能方法的算法相同,且其对应的控制流图和执行逻辑接近,适合从控制流图和执行路径分析的角度整体上检验代码算法结构相似性,因此对上述类型的代码分析效果明显,反映在实验结果中,这一类库检测相似度值与CCFinder结果差距最小。对表5、6中同一用例检测结果的对比,在总体准确性上基本与CCFinder保持同一水平,验证本文提出方法可行性。

对表7分析,本文提出的方法在检测时间上缩短。由于对执行结点路径序列的比较计算相似度,程序结点化、遍历路径序列是对程序控制流图的操作,相比于需要对每行代码转换成对应的token序列的过程复杂度不高,本文用于计算代码间相似度是通过程序对应的控制

表7 运行时间对比

s

方法	JDK1.5							C++SL						
	No.1	No.2	No.3	No.4	No.5	No.6	No.7	No.8	No.9	No.10	No.11	No.12	No.13	No.14
本文方法	235	832	58	199	528	24	67	401	231	195	124	759	251	56
CCFinder	372	1 307	114	423	875	57	155	602	372	339	304	1 101	444	84

流图中的每一条路径序列，再依次比对属于不同图上的两组执行路径序列间编辑距离，以及映射成序列折线间点距和折线相关度，分别从序列文本编辑距离、折线图几何关系以及统计相关度三方面最后按照一定权重比例得出程序间的相似度，计算复杂度上接近自由粒度匹配算法，所以在综合运行效率上有所提升。

综上分析，总体测试结果上本文方法与基于token序列的自由粒度匹配检测算法结果相比偏低，但是运行时间较之缩短。对检测精确度不高、注重时效性的需求具有可行性和应用价值。

7 结束语

通过实验结果数据，验证了上述方法的可行性。文中所述的算法，有复杂度低、耗时短的特点，能够快速检测出程序代码在结构上的相似度，可用在克隆检测的初始阶段分析，结果作为程序相似度度量的影响因子。同时，在针对代码结构检查和后续的重构问题上，具有一定的应用意义。

参考文献：

[1] Keivanloo I，Roy C K，Rilling J.SeByte：scalable clone and similarity search for bytecode[J].Science of Computer Programming，2013.

[2] 于冬琦，彭鑫，赵文耘.使用抽象语法树和静态分析的克隆代码自动重构方法[J].小型微型计算机系统，2009(9)：1752-1760.

[3] 田冰川，孙珂，巢汉青.简化GCC抽象语法树的新型算法[J].计算机科学，2015(S1)：516-518.

[4] 杨昌坤，许庆国.C程序控制流程模型的提取技术与实现[J].计算机科学，2014(5)：208-214.

[5] Komoddoor R，Horwitz S.Using slicing to identify duplication in source code[C]//Proceedings of the 8th International Symposium on Static Analysis，2001：40-56.

[6] Liu Chao，Chen Chen，Han Jiawei，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] 吴冲.基于抽象语法树的重复代码检测[D].上海：上海师范大学，2015.

[8] 叶俊民，王珍，戴跃庭，等.一种源程序级软件验证方法研究[J].小型微型计算机系统，2014(3)：543-548.

[9] 李亚军，徐宝文，周晓宇.基于AST的克隆序列与克隆类识别[J].东南大学学报：自然科学版，2008(2)：228-232.

[10] 郭婧，吴军华.一种新的检测结构克隆的方法[J].计算机工程与科学，2011(12)：78-83.

[11] 刘云龙.基于Token的结构化匹配同源性代码检测技术研究[J].计算机应用研究，2014(6)：1841-1845.

[12] 杨晓杏.基于度量元的软件缺陷预测技术[D].合肥：中国科学技术大学，2014.

[13] 辛天卿.基于序列匹配的代码克隆分析系统设计与实现[D].辽宁大连：大连理工大学，2009.

[14] Jeh G，Widom J.A measure of structural-context similarity[C]//Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining，Edmonton，2002.

[15] Zager L A.Graph similarity and matching[D].Cambridge，MA：Dept of Electrical Engineering and Computer Science，Massachusetts Institute of Technology，2005.

[16] 单永明.一种源程序到控制流图的自动生成方法[J].小型微型计算机系统，1996(10)：45-49.

(上接54页)

[6] Srisangngam P，Mongkudvisut P，Yimman S，et al.Symetric IIR notch filter design using pole position displacement[C]//International Symposium on Intelligent Signal Processing and Communications Systems，2009：1-4.

[7] Yimman S，Praesomboon S，Soonthuk P，et al.IIR multiple notch filters design with optimum pole position[C]//International Symposium on Communications and Information Technologies，2006：281-286.

[8] Pei S C，Tseng C C.IIR multiple notch filter design based on allpass filter[J].IEEE Transactions on Circuits & Systems II Analog & Digital Signal Processing，1997，44(2)：133-136.

[9] 王秋生，杨浩，袁海文.基于粒子群优化的数字多频陷波滤波器设计[J].仪器仪表学报，2012，33(7)：1661-1667.

[10] 任伟，曾以成，陈莉，等.基于自由搜索算法的数字多频陷波滤波器设计[J].计算机工程，2014，40(12)：209-213.

[11] Fan Chaodong，Ouyang Honglin，Zhang Yingjie，et al.Optimization algorithm based on kinetic-molecular theory[J].Journal of Central South University，2013，20(12)：3504-3512.

[12] 范朝冬，张英杰，欧阳红林，等.基于改进斜分Otsu法的回转窑火焰图像分割[J].自动化学报，2014，40(11)：2480-2489.

[13] Van Waterschoot T，Moonen M.A pole-zero placement technique for designing second-order IIR parametric equalizer filters[J].IEEE Transactions on Audio Speech & Language Processing，2007，15(8)：2561-2565.

[14] Wang Qiusheng，Song Jialing，Yuan Haiwen.Digital multiple notchfilter design based on genetic algorithm[C]//International Conference on Instrumentation & Measurement，2014：180-183.

[15] 欧阳波，程栋，王玲.改进小波阈值算法在心电信号去噪中的应用[J].计算机工程与应用，2015，51(4)：213-217.