基于后缀数组检测函数克隆*

侯 敏，张丽萍，史庆庆，刘东升
（内蒙古师范大学 计算机与信息工程学院，呼和浩特 010022）

摘 要：为了提高检测效率，提出了一种新的函数克隆检测方法。该方法对传统后缀数组进行了改进，优化了基于后缀数组的算法。利用该算法可高效查找重复函数子串，进而检测出 Type-1 和 Type-2 类型的函数克隆。同时开发出相应的函数克隆检测工具 FCD 以实现该方法，并检测了 24 款 C 语言的开源软件。实验结果的分析验证了 FCD 能高效检测软件中的函数克隆。

关键词：函数克隆；克隆检测；token 串；后缀数组；公共函数前缀
中图分类号：TP311 **文献标志码**：A **文章编号**：1001-3695(2014)04-1082-04
doi:10.3969/j.issn.1001-3695.2014.04.030

Detecting function clone based on suffix array

HOU Min, ZHANG Li-ping, SHI Qing-qing, LIU Dong-sheng
(*College of Computer & Information Engineering*, *Inner Mongolia Normal University*, *Hohhot* 010022, *China*)

Abstract: In order to improve the detection efficiency, this paper presented a new function clone detection method which improved the traditional suffix array, optimized algorithm based on the suffix array. Using the algorithm could efficiently search repeated token substrings and detect function clones of Type-1 and Type-2. And also it developed a function clone detection tool FCD which implemented the method and applied it to detect clones in 24 open source software systems of C language. Analysis of experimental results show that the FCD can accurately and efficiently detect function clones in the huge software systems.

Key words: function clone; clone detection; token string; suffix array; public function prefix

在软件开发过程中，程序员为了提高编程效率经常用拷贝、粘贴源代码的方法快速实现一些相似或相同的功能。被拷贝的代码段有的被略作修改，有的甚至被原样引用，所以这些代码段间存在着相同或相似的语法或语义特征；而像设计模式、框架、相似 API 等的使用，本身也会导致一些类似或相同代码段的产生。这些代码段都被叫做克隆代码[1]。研究表明，一个软件系统中会有 9% ~17% 的克隆代码，有时甚至高达 50% 以上[2]。由于克隆代码的普遍性，以及对代码质量的重要影响，近年来克隆的相关研究成为代码分析领域中一个十分活跃的分支。从不同的角度可以把克隆代码分成不同类型，现有研究中主要采用两种方法进行分类。一种方法是根据代码相似度，将其分成四类[3]：Type-1 克隆指代码段中除了空格与注释外都相同；Type-2 克隆指代码段中除了标志符、类型、空格和注释外句法上都相同；Type-3 克隆指复制代码段后，除了改变标志符、数据类型、空格和注释外，对语句也作了增/删/改；Type-4 克隆指功能上相同但句法上不同的代码段。其中，Type-1 被称为完全克隆，Type-2 和 Type-3 克隆被称为近似克隆[4]，Type-4 被称为语义克隆[5]。另一种方法根据检测粒度，将其分成文件克隆、类克隆、函数克隆、块克隆以及语句克隆五种类型[5]。

代码的克隆在软件开发过程中具有积极的作用，如复制一段无缺陷的源代码，不仅可以降低编写新代码的潜在风险，而且可以提高开发效率，这些有益的克隆代码称为无害克隆或有益克隆。但情况并不总是这样，如果复制一段含有潜在 bugs 的代码，可能会导致这些 bugs 的大量繁殖。有时维护者对一段克隆代码修改后，必须要对与此相关的所有克隆作一致修改，否则就有可能引入新的 bugs[6,7]。因此，当一款规模较大的软件系统由于克隆而出现问题时，快速准确定位克隆代码将对后续维护起到关键而积极的作用。

1 克隆检测技术描述

克隆检测就是定位源代码中的克隆代码，并以克隆对或克隆群的形式反馈。其中，克隆对（clone pair）是指相互之间存在克隆关系的代码段；克隆群（clone group）是一些克隆代码段的集合，其中任意两个代码段都是克隆对。

克隆检测在软件开发和维护过程中承担了非常重要的角色，是克隆研究领域的基础工作，为后续的深入研究提供基础数据。业内越来越多的人在关注克隆检测，许多工具和技术已被提出。检测完全克隆和近似克隆主要采用基于 token 串[8]、基于解析树（parse tree）[9] 和基于 Java 字节码[10] 等方法。其中，基于解析树的方法时空复杂度高，可以有效检测 Type-3 克隆；而基于 token 的方法能有效地检测 Type-1 和 Type-2 克隆，时空复杂度较低且与语言无关，但在处理 Type-3 克隆时会有许多误报；基于 Java 字节码的方法一般用于检测类克隆。在基于 token 串的方法中，常使用后缀树算法[1] 查找重复子串，

收稿日期：2013-06-18；**修回日期**：2013-07-30 **基金项目**：内蒙古自然科学基金资助项目(2011MS0906)；国家自然科学基金资助项目(61363017)

作者简介：侯敏(1973-)，女，讲师，硕士，主要研究方向为软件方法学、计算机教育应用(houmin920@163.com)；张丽萍(1974-)，女，副教授，硕士，主要研究方向为软件工程、计算机教育应用；史庆庆，男，硕士研究生，主要研究方向为软件方法学；刘东升(1956-)，男，教授，主要研究方向为软件工程、软件方法学、计算机教育应用.

从而定位克隆代码。

本文提出了一种新的更高效准确的函数克隆检测方法,使用基于优化了的后缀数组的算法代替基于后缀树算法查找重复函数子串,进而检测 Type-1 和 Type-2 类型的函数克隆。本方法分三个步骤:a)token 化源代码,使用词法分析工具将对克隆代码无意义的内容删掉,把源代码转换成 token 串;b)使用基于后缀数组算法在词法分析输出的 token 串里查找重复函数子串;c)根据 b)查找的重复子串定位克隆代码。根据上述方法实现了一款克隆检测工具 FCD,利用该工具能准确检测大规模软件的函数克隆。进一步,用该工具检测了 24 款 C 语言的开源软件。

2　基于后缀数组的函数克隆检测

由于 Type-1、Type-2 克隆代码的特征较为明显,而且稳定性较低[11],值得投入更多的关注。本文的研究目标就是使用基于优化了的后缀数组的算法代替基于后缀树算法查找重复函数子串,以高效检测 Type-1 和 Type-2 类型的函数克隆,并根据此原理实现了 FCD 工具。在深入研究现有检测方法或技术的优缺点后发现,尽管基于解析树和基于 PDG[12] 等方法也能检测这两类克隆,但它们的时空复杂度和误检率都较高,而 token作为源代码的中间表示形式,能更有效地检测这两类克隆代码。通过在 token 串中查找重复子串也可以简化函数克隆的检测。FCD 具体处理步骤如图 1 所示。

图 1　FCD 的处理步骤

2.1　Token 化源代码

这部分是把源代码作为输入,经过词法分析抽取出软件源代码中所有的函数并格式化为 token 串。本文只检测 C 语言软件的 Type-1、Type-2 函数克隆,所以输入文件是 C 源程序文件。Type-1 克隆代码相互间要么完全相同,要么只存在注释、空白符及格式上的差异。所以只需将代码内的注释及空白全部删除,经过此预处理后,字符串完全一样的两段代码就是 Type-1 克隆。然而,Type-2 克隆代码相互间除了存在空白符及注释上的不同外,还有标志符名称、常量及数据类型的差异。所以在对源代码作词法分析时,还需对这些标志符、类型等进行标准化处理,以消除它们间的差异。由于比较代码段在 Type-1 及 Type-2 层上的相似性时不涉及语法验证及语义信息,所以不需要像常规的词法分析那样要产生一个与 token 相应的符号表。处理过程虽然并不十分困难,但对检测技术的查全率和准确度具有重要影响。

本文选择 Flex 作为词法分析器的生成工具。格式化过程包括三个主要步骤:a)去除掉代码中的空格和注释部分;b)因为只检测函数克隆,所以去除掉不是函数的代码部分;c)把经过前两步处理后的代码按照本文设计的规则标准化后转换成 token 串。其中主要是步骤 c)可能会造成误检率和漏检率。针对 C 语言的软件系统,本研究设计的主要标准化规则如表 1 所示。

本文只研究函数克隆,所以源代码中函数的识别及标记是词法分析部分的重要内容。当源代码被转换为 token 串时,需在每个函数对应串的开始处添加一个特殊符号‘$’,该字符在转换后的 token 串中的 ASCII 码值最小。这样不仅可以标记函数而且能将所有函数转换后的 token 串在后缀数组里按字典序连续地排在最前面,在查找公共函数前缀时只要对排在前面的后缀进行即可,对降低查找的时空复杂度非常关键,而且可以简化后续工作中的查找克隆群算法的实现。

表 1　C 语言的主要标准化规则

规则	规则描述
规则 1	基本数据类型由 D 替换,如 int、float、void、double、long 等
规则 2	关键字的转换,for→F、while→W、if→I、else→E、case→C 等
规则 3	结构指针由 P 替换,普通变量和函数名由 V 替换,数组由 r 替换
规则 4	整数、浮点数、字符常量由 0 替换,比较字符由 c 替换,字符串常量由 S 替换
规则 5	数据结构的初始化由 1 替换,如“={5,5,7;8,9,10}”→1。节省存储空间,减少比较次数
规则 6	‘,’和‘;’直接去除,能很好地降低存储空间,并减少比较操作

对于每个函数的末尾结束符号‘}’,需用另一个特殊符号‘~’替换,以标志函数的结束。这样在 token 串中函数对应的子串都是从字符‘$’到‘~’的子串,这将有助于以函数为单位的相似性比较,不会使相似性比较操作跨过函数的结束符。借助重复单元为函数这一特征,进一步优化了基于后缀数组的算法。图 2 是源代码标准化前后的对比图,为了直观地对比源代码被 token 化的前后效果,图中的源代码还未应用转换规则 6 去掉‘,’和‘;’,也未删除空白符。

```
int connect_to_many(struct address_list *al,unsigned short port,int silent)
{
  int i,start,end;
  address_list_get_bounds(al,&start,end);
  for(i=start;i<end;i++)
  {
  ip_address addr;
  int sock;
  address_list_copy_one(al,i,&addr);
  sock=connent_to_one(&addr,port,silent);
  if(sock>=0)
  /*success*/
    return sock;
  address_list_set_faulty(al,i);
  /*The attempt to connect failed. Continue
    with loop and try next address.*/
  }
  return -1;
}
```

```
$D V(T V * V,D V,D V)
{
  D V,V,V;
  V(V,&V,&V);
  F(V=V;V c V;V++)
  {
  V V;
  D V;
  V(V,V,&V);
  V=V(&V,V,V);
  1(V c 0)
  R V;
  V(V,V);


  }
  R 0;
~
```

图 2　C 代码 token 化的示例

最后将软件中的所有 token 串连接起来存放在一个一维数组 token 中。在 token 化过程中,还必须要记录每个 token 串在源代码中的位置,此位置信息要被应用到函数克隆的反馈报告中。本文实现的 FCD 工具现在仅能检测 C 源文件的函数克隆,但是只要制定出相应的标准化规则,上述基于 token 化的处理方法可以检测任何语言的源代码。

2.2　查找公共函数前缀

函数克隆检测的关键问题就是查找公共函数前缀即重复函数子串。重复子串的查找也是克隆检测过程中最耗时的一步,目的是在词法分析器产生的 token 序列中查找不重叠的重复子串。现有的基于 token 的克隆检测工具总是选用基于后缀树的算法来查找重复子序列,如 CCFinderx[13] 和 iClones[3]。考虑到后缀数组是一种更简单的、易实现的数据结构,并且在查找重复子串时比后缀树具有更优的时空复杂度,本文对其进一步优化后替换后缀树,并设计相应的算法来查找不可重叠重复函数子串。

1993 年,Manber 等人[14] 引入后缀数组,它是把一个字符串的所有后缀按字典序排序后形成的数组。字符串后缀是指从字符串的某个位置 i 开始到串尾的子串。后缀数组可以按

照任何给定的模式高效地搜索文本,它包含了许多和后缀树相同的信息,但在很多应用中更简单高效。在查找字符串的重复子串时,后缀数组需和名次数组 rank、最长公共前缀(LCP)数组 height 结合使用。名次数组 rank 保存了所有后缀的排名,是后缀数组的逆运算。后缀数组中相邻元素的最长公共前缀的长度保存在 height 数组中。图 3 举例阐述了这三个数组及相互关系。

字符串 S

a	b	a	h	g	e	a	b

排好序的后缀	后缀数组	名次数组	LCP 数组
ab	SA[1] = 7	rank[1] = 2	height[1] = 0
abahgeab	SA[2] = 1	rank[2] = 5	height[2] = 2
ahgeab	SA[3] = 3	rank[3] = 3	height[3] = 1
b	SA[4] = 8	rank[4] = 8	height[4] = 0
bahgeab	SA[5] = 2	rank[5] = 7	height[5] = 1
eab	SA[6] = 6	rank[6] = 6	height[6] = 0
geab	SA[7] = 5	rank[7] = 1	height[7] = 0
hgeab	SA[8] = 4	rank[8] = 4	height[8] = 0

图 3 几个数组及关系

本文选用 DC3 算法[15]构造目标串的后缀数组 SA,时空复杂度均为 $O(n)$,其中 n 为 token 序列的长度。为了避免查找过程中数组越界问题,需在 token 串的末尾添加一个特殊字符'\t'。至于名次数组 rank,遍历一次后缀数组 SA 就可得出;height 数组可根据 GetHeight 算法[16]求得,时间复杂度为 $O(n)$。但是由于本文是查找 token 序列中相同的函数前缀,而不是求两后缀的最长公共前缀,所以需对上述的 GetHeight 算法进行改进:以函数为单位查找重复子串,而且如果在相邻的后缀中存在公共函数前缀,那么其长度值就赋值给 height 数组对应元素,否则其值为 0。改进的 GetHeight 算法的 C 语言实现如下所示:

```
void calheight( int *SA,int n)
{
  int i,j,rk;
  short int k=0;
  for(i=0;i<n;i++) rank[SA[i]]=i;//get rank array
  for(i=0;i<n-1;i++)
  {
    if(rank[i]>=1)
    {
      j=SA[rank[i]-1];//suffix j is front of suffix i
      while(token[i+k]==token[j+k]&& token[i+k]!='~')
      k++;
      if(token[i+k]!=token[j+k])
        rk=0;
      else//two suffix have common prefix function
        rk=k+1;
    }
  height[rank[i]]=rk;
  if(k>0)//use height[rank[i]]>=height[rank[i-1]]-1
    --k;
  }
  return;
}
```

2.3 定位克隆代码

通过上述改进的 GetHeight 算法计算出 height 数组元素的值,一组连续的非 0 值表示一个克隆群内的一组相同的函数序列,不同克隆群是由 0 值隔开。根据后缀数组、名次数组及公共前缀数组,公共函数前缀很容易获得。在后缀数组里,所有的函数后缀都连续地排在前面,所以相应的 height 值也位于 height 数组的前面。只需遍历数组 height 前面单元就可查找 token 序列中的重复函数子序列,因此时间复杂度是个常量。找到公共函数前缀,结合之前记录的每个 token 串在源代码中的位置,就可以报告源码中的函数克隆检测结果。

3 实验结果分析

根据上述原理设计了函数克隆检测工具 FCD,并利用该工具检测了 24 个 C 语言的开源软件,其中包括操作系统、数据库、游戏软件和 IM 软件等。这些目标软件的选择是基于软件规模、类型及功能的多样性。本文也引入了一些度量单位以便从不同角度分析实验结果进而对此工具作出评价。实验环境配置为:i3 CPU 2.2 GHz,内存 2 GB;操作系统 Ubuntu 11.10,词法工具 Flex,编译器 GCC-4.6.1。FCD 的输入是 C 源文件。

3.1 度量单位

为了表示软件系统内函数克隆的程度,定义了函数克隆密度(function clone percentage, FCP),指软件内被克隆的函数个数占软件内函数总个数的百分比;也定义了行数克隆密度(lines cloned percentage,LCP),指软件内被克隆的行数占软件所有源代码总行数的百分比。这两个度量单位提供了一个软件系统总的克隆程度数据,但它们不能显示这些克隆是来源于一个特定的文件还是分散在整个系统的所有文件之中。所以,本文又定义了包含函数克隆的文件密度(file associated with clones percentage,FAWCP),它表示包含函数克隆的文件占系统中所有源文件的百分比。表 2 中,大小表示所有 C 源文件的总代码行数,检测时间表示 FCD 从开始读源文件到输出克隆结果的总时间。

表 2 FCD 检测 24 款软件的结果

软件名称	大小(KLOC)	函数个数	LCP/%	FCP/%	FAWCP/%	检测时间/s
wget-1.13	65	675	0.4	4.1	10.4	0.13
dovecot-2.1.8	272	3 650	1.6	11.1	20.3	1.34
clamav-0.97.5	203	1 739	1.4	11.9	14.9	0.85
Advor-0.3.0	113	2 760	1.4	6.1	45.2	0.46
gcc-4.7.0	2 865	12 924	1.2	18.6	5.7	21.0
Geany-1.22	91	2 124	2.5	9.4	52.6	0.2
cvs-1.11.17	104	2 297	4.8	24.9	56.7	0.17
bluefish-2.2.1	64	1 912	2.8	9.2	46.1	0.22
Linux-3.4.7	11 473	215 988	4.1	16.8	48.9	76.7
Nginx-1.3.2	119	1 556	4.3	14.6	50.9	0.28
pidgin-2.10.4	392	10 449	6.5	18.3	59.6	2.02
ffmpeg-0.11.1	524	10 005	3.6	10.6	34.9	4.08
xemacs-21.5.29	372	9 238	6.8	23.9	67.0	1.42
zabbix-2.0.0	108	1 685	6.4	18.5	35.5	0.32
gthumb2	164	5 875	12.6	29.9	82.1	0.50
gedit-3.4.1	82	2 433	9.3	24.9	88.0	0.19
smalltalk3.2	120	1 803	4.3	18.3	29.1	0.27
emacs-21.4	318	4 165	4.9	30.1	46.2	0.91
reactos-0.3.14	3 519	67 748	8.9	33.9	44.6	25.77
postgresql9.2be	1 009	1 520	0.7	21.1	5.6	5.39
freeciv-2.3.2	398	8 003	9.8	26.2	39.7	2.16
fdisk-2.0	14	525	34.1	47.1	75.9	0.05
nagios-3.4.1	103	1 431	8.1	22.8	45.4	0.38
ext2fsd	98	677	1.5	14.3	61.6	0.16

通过观察表 2 发现,FCD 能快速检测出 Type-1 和 Type-2 类型的函数克隆,而且能够检测大规模软件。表中也显示出大多软件的 LCP 低于 FCP,说明大部分的克隆函数都非常小;但也有例外,如 fdisk 中的 LCP 与 FCP 几乎相同。FAWCP 有利于评价软件的维护成本,因为它反映了软件中克隆代码的分散情况。函数克隆在软件中分散得越厉害,则越难控制或维护。表 2 显示 FAWCP 在绝大多数情况下比 CLP 及 FCP 都高,并且有 14 款软件的 FAWCP 高于 45%。

3.2 准确性分析

到目前为止,克隆检测研究领域里还没有函数克隆的标准

测试集,检测结果的准确性分析主要是参照一些类似系统或通过手工进行。但是手工分析所有软件的克隆结果是不可能的,本文抽查三款软件分析 FCD 误检率,结果如表 3 所示。

表 3　FCD 检测三款软件的误报率

软件项目	克隆函数数目	误报的数目	误报率
gedit	605	25	4.13
gthumb	1 755	97	5.53
Nginx	227	6	2.64

通过人工查看,发现这些被误报的克隆函数含有的执行语句不超过 4 条,虽然都满足 Type-2 克隆的特征,但它们在语义或功能上完全不同,所以不能将这种相似或相同看成克隆。

至于漏报率,由于软件的规模都较大,无法手工找出所有的函数克隆,本文通过两种方法来分析 FCD 的漏报率。一种方法参照 NICAD[9] 的检测结果,NICAD 是一款成熟的克隆检测工具,它能比较准确地检测 Type-1、Tpye-2 类型克隆代码。分析比较结果发现,NICAD 能检测到的 Type-1、Tpye-2 函数克隆 FCD 都能检测到,而且 FCD 检测到的更多。原因是 FCD 定义的 Tpye-2 类型克隆范围更大,不仅标准化了标志符、数据类型、空格和注释,而且还包括了常量和复杂数据结构的初始化,除了此六种元素外句法上相同的代码都属于克隆代码。两工具对四款不同软件检测的结果如表 4 所示。表中 NOF 是克隆函数的个数(number of function clones)。

表 4　NICAD 与 FCD 检测结果比较

工具	bluefish(NOF)	geany (NOF)	Pidgin (NOF)	gthumb(NOF)
NiCad	299	310	1 277	829
FCD	382	409	1 914	1 755

分析 FCD 漏报率的另一种方法是在 Roy 等人[3] 设计的一个简单测试集 Benchmark 上进行验证。该测试集包括四种类型的克隆代码,本文只选前两种。验证结果如表 5 所示。

表 5　FCD 在 Benchmark 上的验证结果

工具	Type-1	Type-2
Benchmark	3	4
FCD	3	3

表中显示 FCD 漏报了一个 Tpye-2 克隆代码,此代码段里用两个变量替换了原来的一个变量,也就是说在代码段里增加了一些标志符,这种情况是 FCD 的 Tpye-2 克隆不包括的。综上实验结果证明,FCD 能精确检测出 Tpye-1、Tpye-2 克隆,有很低的漏检率和误检率。

4　研究的不足

本研究主要有以下三个方面的不足:

a)函数最小规模阈值的选取。经过手工检查发现,包含执行语句太少的重复函数只有极少部分有意义即能作为克隆函数,尽管这些函数除了数据类型、变量名和函数名不同外句法上都相同,但它们不能被认定为克隆。而 FCD 的误检率主要是由这些小函数引起的,提高函数规模阈值又会引起漏检率的升高。

b)目标系统受限。本研究只是选取了一些开源软件进行测试,没有涉及到商业软件,所以还不能把研究结果一般化。

c)实验结果主要通过手工验证。在论证实验结果的准确性时,由于没有标准测试集,所以主要是通过手工检查分析,难免对结果造成影响。

5　结束语

本文提出一种新的函数克隆检测方法,即利用基于后缀数组的算法查找重复函数以检测函数的完全克隆和 Type-2 类近似克隆。利用此方法实现了一款函数克隆检测工具 FCD,并对 24 款 C 语言开源软件进行测试。实验结果显示,FCD 可以准确而快速地检测软件中的克隆函数。对商业软件的测试和对 Java 等语言源代码的检测是笔者下一步研究工作的重点。

参考文献:

[1] KAMIYA T, KUSUMOTO S, INOUE K. CCFinder:a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Trans on Software Engineering,2002,28(7):654-670.

[2] ZIBRAN M F,ROY C K. IDE-based real-time focused search for near-miss clone[C]//Proc of the 27th Annual ACM Symposium on Applied Computing. New York:ACM Press,2012:1235-1242.

[3] ROY C K,CORDY J R,KOSCHKE R. Comparison and evaluation of code clone detection techniques and tools:a qualitative approach[J]. Science of Computer Programming,2009,74(7):470-495.

[4] ZIBRAN F M,SAHA R,ASADUZZAMAN M,*et al*. Analyzing and forecasting near-miss clones in evolving software:an empirical study[C]//Proc of the 16th IEEE International Conference on Engineering of Complex Computer Systems. Washington DC:IEEE Computer Society,2011:295-304.

[5] ZIBRAN M F,ROY K C. The road to software clone management:a survey,TR 2012-03[R]. Saskatoon:University of Saskatchewan,2012:1-66.

[6] ROY K C. Detection and analysis of near-miss software clones[D]. Kingston:Queen's University,2009.

[7] 党映农,S·卡恩,D·张,等. 代码克隆通知以及体系结构改变可视化:CN,ZL201110427723.6[P]. 2012-09-19.

[8] LI Zhen-min,LU Shan,MYAGMAR S,*et al*. CP-Miner:finding copy-paste and related bugs in large scale software code[J]. IEEE Trans on Software Engineering,2006,32(3):176-192.

[9] ROY C K,CORDY J R. NICAD:accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization[C]//Proc of the 16th IEEE International Conference on Program Comprehension. Washington DC:IEEE Computer Society,2008:172-181.

[10] CUOMO A,SANTONE A,VILLANO U. A novel approach based on formal methods for clone detection[C]//Proc of the 6th International Workshop on Software Clones. Washington DC:IEEE Computer Society,2012:8-14.

[11] MONDAL M,ROY C K,RAHMAN M S, *et al*. Comparative stability of cloned and non-cloned code:an empirical study[C]//Proc of the 27th Annual ACM Symposium on Applied Computing. New York:ACM Press,2012:1227-1234.

[12] KRINKE J. Identifying similar code with program dependence graphs[C]//Proc of the 8th Working Conference on Reverse Engineering. Washington DC:IEEE Computer Society,2001:301-309.

[13] BAKER B S. A program for identifying duplicated code[J]. Computing Science and Statistics,1992,24(1):49-57.

[14] MANBER U,MYERS G. Suffix arrays:a new method for on-line string searches[J]. SIAM Journal on Computing,1993,22(5):935-948.

[15] KARKAINEN J,SANDERS P,BURKHARDT S. Linear work suffix array construction[J]. Journal of the ACM,2006,53(6):918-936.

[16] KASAI T, LEE G, ARIMURA H,*et al*. Linear-time longest-common-prefix computation in suffix arrays and its applications[C]//Proc of the 12th Annual Symposium, Combinatorial Pattern Matching. Berlin:Springer-Verlag,2001:181-192.