基于程序控制流图源代码相似程度分析系统①

陈 新

(广东省培英职业技术学校, 广州 510630)

摘 要: 源代码相似程度分析在软件工程和计算机教学等领域都有重要的应用. 软件工程领域的源代码盗窃和著作权纠纷仲裁, 计算机教学领域的学生作业作弊分析都需要源代码相似程度的分析. 良好的源代码相似程序分析软件还可以应用于相似代码聚类和搜索引擎的源代码搜索领域. 尽管源代码相似程度分析问题由来已久, 但是这个问题并没有令人十分满意和惊喜的研究结果. 源代码有其特殊结构, 使用传统的纯文本相似度分析显然是不合适的. 将首先介绍这个问题的研究历史和进展, 简单分析这个问题的难点所在, 继而介绍一个新的基于程序控制流图分析的源代码相似程度分析系统, 并给出其算法和实现细节. 文章最后将分析这个方法的优劣所在, 讨论这个方法的进一步改进方向.

关键词: 作弊检测; 源代码相似度; 控制流图; 哈希函数; GCC

Program Control Flow Graph Based on the Similarity of Source Code Analysis System

CHEN Xin

(Puiying Occupation Technical School of Guangdong Province, Guangzhou 510630, China)

Abstract: It is very important to detect source code similarity in the field of both software engineering and computer science education. Source code stealing and copyright dispute in software engineering, as well as plagiarism detection of student assignment in programming course, call for automation of source code similarity detection. Reliable software to detect source code similarity also helps in the field of source code clustering and source code searching in wide range. Though such problem rises along with the invention of program language, there is not satisfying research result up till now. Source code has its particular structure, making it improper to use traditional pure text similarity detection method over it. This paper first introduce the history and current progress of this problem, analysis the difficulty, then presents a new system for source code similarity detect base on control flow graph analysis, along with the algorithm it uses and the implementation details. In this paper we also discuss the advantage and disadvantage of the method and ways of improve it.

Key words: plagiarism detection; source code similarity; control flow graph; hash function; GCC

源代码相似程度分析在多种场合都与相当重要的应用. 在软件工程领域, 软件源代码窃取和著作权纠纷层出不穷. 代码扰乱技术的出现使得这类纠纷难以分析和仲裁. 源代码盗窃者可以使用代码扰乱器轻易使得所窃取代码面目全非, 从而令人难以判断它是否源自被窃取的源代码. 在计算机教学领域, 学生作弊抄袭问题长期困扰着教师们. 通过简单的变量名替换, 改换程序编程风格, 打乱无关的代码顺序或者利用一些特殊的语法技巧, 作弊者能够轻易使得抄袭程序难以被识别[1]. 教师需要在几十乃至数百份作业中人工分拣出作弊程序所需要的工作量相当大. 同样的问题也发生在各类程序设计竞赛中, 例如 ICPC(国际大学生程序设计竞赛). 源代码相似程度分析系统可以帮助解决以上问题. 一个良好的源代码相似程度分析系统还有助于教师把多份学生作业按照相似程度进行分类, 从而发现对于一道题目学生有多少种不同的解法.

① 收稿时间:2012-08-07;收到修改稿时间:2012-09-10

类似的系统也可以应用于以搜索源代码为目标的搜索引擎, 利用代码相似程度分析来改善搜索结果. 该类系统还可能作为编译器的代码优化的辅助手段[23].

本文将介绍一个新的源代码相似程度分析系统, 系统所使用的算法也基于源代码结构, 特别地, 这个算法基于程序控制流图(Control Flow Graph, CFG)的同构分析, 在局部上使用最长公共子串(Longest Common Subsequence, LCS)等方法来进一步细化统计程序相似程度.

1 现有的源代码相似程度分析

(1) Diff 工具和 LCS 算法

Diff 工具是一个文本文件比较工具, 它能显示两个文件每行之间的改动. Diff 工具的输出被称为 patch, 因为它可以作为 patch 工具的输入而对文件打补丁. Diff 工具最早在 1970 年代在 Unix 系统上进行开发并作为 Unix 系统的一个附带的工具. Diff 的作者在找到正确算法之前做了很多尝试, 最终他们认为 LCS(Longest Common Subsequence, 最长公共子序列)算法是最适合的[4,5].

(2) 序列铺嵌的贪心算法

所谓序列铺嵌的贪心算法(Greedy String Tiling Algorithm), 是贪心地寻找一个方案, 把要匹配的两个序列分拆成片段, 然后尽可能多地互相铺嵌匹配[4,7].

(3) 线性化代码的方法

Jeong-Hoon Ji, Gyun Woo 和 Hwan-Gue Cho 提出了一种把代码线性化(Linearization, 这里可能可以翻译成“平坦化”)的源代码相似程度分析方法, 并把这种方法应用于 ICPC(国际大学生程序设计竞赛)的作弊检测中.

这种线性化代码的方法, 如同所有的基于源代码结构的方法一样, 首先对需要分析的源代码进行词法分析并把源代码转换为符号(Token)序列. 之后它使用所谓“程序静态跟踪方法”(Program Static Tracing Method)把代码线性化(平坦化)[7].

2 源代码相似程度分析算法设计

2.1 算法框架

本文所述的系统采用一个新的方法对源代码进行相似程度分析, 该方法揉合了多种算法. 一次分析大致分为以下五个步骤:

S1. 把需要比较的源代码转换为控制流图;

S2. 对所得到的控制流图进行必要的简化;

S3. 对两个控制流图进行同构判定;

S4. 如果两个控制流图同构, 则进一步比较相对应的基本块的语句序列;

S5. 输出总的语句匹配数量和匹配语句占整个源代码的比例, 作为相似程度分析的依据.

2.2 程序控制流图的表示

在本文所述的系统中, 每一个源代码由一个程序控制流图来表达. 每一个结点依然表示一个基本块. 在控制流图上有两种边: 跳转边和函数调用边. 跳转边表示直接跳转和条件跳转动作. 显然, 根据基本块的定义, 每个结点(基本块)只能有一个或两个跳转边出边, 但可能有多个跳转边入边. 每个结点(基本块)可能有多个函数调用, 每个对应一个函数调用边. 函数返回不用任何边表示. 每个节点各自保存其语句流信息. 程序控制流图能够清晰表达程序的逻辑结构, 并且去除函数书写顺序等语法信息的干扰. 按照前面的方法得到的控制流图, 还可以进行进一步简化, 例如, 对程序结果无影响的多余代码和不可达的代码可以从图中移除; 只包含一个跳转的基本块可以被化简之后移除; 如果一个基本块的出口和另一个基本块的入口唯一对应, 那么这两个块可以合并; 比较简单的函数可以被展开, 特别地只被调用过一次的函数应该被展开.

简化控制流图的目的是消除一些代码抄袭/扰乱手段的的影响, 如函数合并展开, 或增加多余代码. 通过简化的控制流图适合进行相似程度判定. 忽略结点的内部信息(基本块内的语句序列), 控制流图就是一个一般的有向图(注意它并不是有向无环图). 一般有向图的同构判定已经被证明是 NP 完全问题, 没有多项式复杂度的确定性算法. 但是可以使用计算 hash 函数的方法大幅减小计算量. 算法由下面的伪代码描述:

输入: 有向图 $G_1(V_1,E_1)$,$G_2(V_2,E_2)$

输出: G_1 和 G_2 是否同构

if ($|V_1|<>|V_2|$) *return false*;

对于每个 $vi \in V_1$, 根据 *vi* 邻近区域信息计算 *hash* 函数 $h(vi)$;

对于每个 $ui \in V_2$, 根据 *ui* 邻近区域信息计算 *hash* 函数 $h(ui)$;

for(每个 V_1 到 V_2 的一一对应 *f*, 使得对于每个 $f(ui)=vj$ 都满足 $h(ui)=g(vj)$){

flag←*true*;

for(每个 $e(xi,yi) \in E_1$){

if(*e*(*f*(*xi*),*f*(*yi*))不属于 E_2) *flag*←*flase*;

}

for(每个 *e*(*xi*,*yi*) ∈E_2){

if(*e*(f^{-1}(*xi*),f^{-1}(*yi*))不属于 E_1) flag←flase;

}

if(*flag*=*true*) *return true*;

}

return false;

上述算法其实是加入剪枝的盲目搜索算法. 剪枝的关键是需要设计一个良好的hash函数综合描述有向图中以一个指定的结点为中心的领域信息. 好的 hash 函数能戏剧性地降低搜索算法的复杂度.

2.3 对基本块内部语句序列进行匹配

如果两个源代码的控制流图是同构的, 那么就可以认为这两个源代码很相似, 可以进行进一步的比较. 2.2 节中使用的算法不仅对两个控制流图进行了同构判定, 还给出了两个控制流图中的结点(基本块)的一一对应关系. 对于两个基本块, 可以使用 LCS 算法计算出它们的语句序列的最长匹配序列. 两个语句是匹配的当且仅当他们的类型相同并且额外信息相匹配. 对于不同类型的语句, 匹配的含义是不一样的. 程序控制流图没有保留任何源代码的格式信息, 没有保留表示符名称信息, 几乎没有保留程序所使用的数据结构和数据类型信息(仅在基本块内部的语句序列中保留了操作数的类型信息). 因此, 修改注释和空白、重命名表示符和简单的代码段位置移动都不会影响该算法对程序相似程度的判断. 通过适当的化简控制流图(算法第二步), 算法同样能检测经过函数拼合或者拆分的相似代码. 和前面列出的集中基于程序结构的分析方法一样, 这个算法并不能原生地检测等价表达式变换. 但等价表达式变换不能改变程序的控制结构, 对算法的影响仅仅是局部的; 况且可以通过改进算法的第四步来获得对等价表达式检测的更好结果. 如果算法的第二部化简控制流图能够考虑更多的情况的话, 算法也能顺利检测部分改变程序控制结构的代码扰乱.

由于在得到控制流图的过程中损失了很多源代码的信息, 算法可能会把两个很不相似的代码误判为相似. 但是由于需要作相似程度分析的代码一般在功能上是相同的, 那么若果它们的结构也相同, 则判为相似也是合适的.

3 源代码相似程度分析系统实现

3.1 利用 GCC 的调试输出获得程序控制流图

前面所描述的算法最难以实现的部分是把源代码转化为控制流图表示. 传统的方法是使用 Bison,Yacc, Antlr 或者 Coco 等“编译器的编译器”制作出相应语言的语法分析工具. 但对于 C++等比较复杂的高级语言, 使用这类工具需要相当大的工作量. 所幸的是, GCC(GNU Compiler Collection)提供了一个调试特性, 为这类的语法分析提供了方便.

在 GCC 提供了大量的编译选项, 值得注意的是 -fdump-tree-* 选项族. -fdump-tree-* 选项族可以在 GIMPLE TREE 上的每一步优化进行后输出相应的结果. 例如在编译源代码的时候加上-fdump-tree-cfg 选项即可得到算法所需的程序控制流图. 使用 -fdump-tree-cfg 选项将得到 C 语言风格的 GCC 调试输出, 格式清晰明了. 使用-fdump-tree-final_cleanup 选项还能得到经过所有树优化之后得到的最终的语法树输出. 这个输出非常符合人类阅读的方式, 但是却不利于机器自动处理(需要作类似 C 语言的语法分析器). 所幸的是-fdump-tree-*选项族还提供 raw 子选项. 使用 -fdump-tree-cfg-raw 得到的调试输出看上去就像下面的样子.

```
;; Function int main() (main)
int main() ()
@1 function_decl name: @2 type: @3 srcp: x.cpp:3
lang: C    link: extern      body: @4
@2    identifier_node    strg: main lngt: 4
@3    function_type    size: @5 algn: 8    retn: @6prms: @7
@4    statement_list    0    : @8 1    : @9 2 : @10
3    : @11 4 : @12    5    : @13 6    : @14
@5    integer_cst    type: @15    low : 8
@6 nteger_type    name: @16    size: @17 algn: 32
prec: 32    sign: signed    min : @18 max : @19
@7    tree_list                valu: @20
@8 call_expr    type: @6 fn : @21    args: @22
@9    modify_expr    type: @6 op 0: @23 op 1: @24
@10 oto_expr    type: @20            labl: @25
@11 modify_expr type: @6 op 0: @23 op 1: @24
@12    goto_expr    type: @20    labl: @25
@13    label_expr    type: @20    name: @25
```

@14 return_expr type: @20 expr: @26

(以下略)

容易看出 raw 格式输出的是一颗树, 并且这棵树以广搜顺序列出结点. 每个结点有若干属性, 这些属性可能是数值、字符串或者指向另一个节点的指针. GCC 文档指出这个格式和 GCC 的树的内部表示一样是语言无关的, 但结点有什么属性由特定的语言确定, 所以又是语言相关的. GCC 文档也指出使用 -fdump-tree-*选项族得到的输出格式并无规范, 可能不同版本的 GCC 会在格式上有戏剧性的变动.

3.2 应用实例

图 1 给出两个源代码, 图 1(A)是学生提交的作业, 该程序输出形如 ax≡b(mod n)的模线性方程的解数. 图 1(B)中的源程序经过了手工的扰乱: 增加了注释; 包含了多余的库文件; 标识符进行了替换; 两个函数的位置进行了掉换; 两个 if…else…条件跳转语句被替换成了三值语句; 求公约数的子函数计算了多余的变量. 本文所述的系统准确地发现这两个源代码是雷同的.

```
#include<iostream>
using namespace std;
int mGCD(int a,int k) {
if(b==0)return a;
else return mGCD(b,a%b);
}
int main(){
int counter;
int a,b,n;
while(scanf("%lc %ld %lc",&a,&b,&n)!=EOF){
      counter=mGCD(a,n);
      if(b%counter==0){
          printf("%ld\n",counter);
      }
      else printf("0\n");
}
 return 0
}
```

(A)

```
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>
#include<stdlib.h>
#include<math.h>
using namespace std;
int gcd(int,int,int&,int&);
int i,j,k;
 int main()
 { //freopen("in.txt ","r",stdin);
   int a,b,n;
  //loop for every cases
while(scanf("%d%d%d",&i,&j,&k)!=EOF)
  {
   int x,y;
   j%gcd(i,k,x,y)==0?printf("%d\n",g)
                    :printf("0\n");
  }
   return 0;
  }
 int gcd(int a,int b,int& x,int& y)i
   {
      retrun (b==0?x=1,y=0,a
      :int t=x-a/b*y,x=y,y=t,gcd(b,a%b));
    }
```

(B)

图 1 一个例子

表 1 展示了本文所述的系统对一次程序设计竞赛的选手提交的分析结果. 表中平均语句条数以 GCC 的中间代码表示为标准来衡量. 题目的难度可以通过正确提交数和这些提交的平均语句条数(编码复杂度)来衡量. 从表中可以看出, 题目简单并且代码简单的题目, 不同的选手的做法近似, 所以容易产生雷同; 而题目难度越大, 代码越复杂, 则越不容易发生雷同. 本文所述的系统所得到的分析数据是合乎常理的.

4 结语

本文简要分析了几种程序相似程度分析方法, 并提出了一种根据程序控制流图对源代码进行相似程度分析的算法, 并讨论了相应系统的实现方法. 尽管本文中描述的算法和实现仍然略显粗糙, 但方法本身尤其优点. 通过进一步的改进, 相信该方法会产生令人满意的结果.

表 1 实测数据

题目	正确提交数	平均语句条数	雷同数(对)
Zero Puzzling	76	58	35
Gene Reprogram	17	151	0
Travel	11	188	0
Abbreviation	71	105	1
String Sequence	44	57	0
DNA Recombination	11	95	1
Xiaoshi's Problem	86	35	347

参考文献

1 王春晖,程金宏,孟繁军,刘东升.程序代码相似性检测技术在教学中的应用.计算机教育,2007,12,137−139.

2 Faidhi JAW, Robinson SK. An empirical approach for detecting program similarity and plagiarism within a university programming environment, Computers & Education, 1987,11(1).

3 Belkhouche B, Nix A, Hassell J. Plagiarism detection in software designs. Proc. of the 42nd Annual Southeast Regional Conference.

4 Ji JH, Woo PG, Cho PG. A source code linearization technique for detecting plagiarized programs. Proc. of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. 2007.

5 Gitchell D, Tran N. Sim: a utility for detecting similarity in computer programs. SIGCSE '99: Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education. New York: ACM, 1999: 266–270.

6 Whale G. Identification of program similarity in large populations. Comput. J, 1990,33(2):140−146.

7 Villano AU. Compiler hacking for source code analysis. Software Quality Journal, 2004,12:383−406.