Constructing Universal Version History

Hung-Fu Chang
University of Southern California
University Park Campus,
University of Southern California
Los Angeles, CA 90089
+1 213 740-9621
hungfuch@usc.edu

Audris Mockus
Avaya Labs Research
233 Mt. Airy Rd.
Basking Ridge, NJ, USA 07920
+1 908 696-5608
audris@avaya.com

ABSTRACT
Developers often copy code for parts or entire products to start a new product or a new release. In order to understand the software change history and to determine the code authorship, we propose to construct a universal version history from multiple version control repositories. To that end we create two practical code copy detection methods at the level of the source code file: prefix-postfix algorithm and prefix algorithm. The full pathname of a file and its version history are used to construct the universal version history of a file by linking together change histories of files that had the same code at any point in the past. The assumption of both algorithms is that developers often duplicate files by copying entire directories. Once the copying is identified we propose an algorithm to link version histories from multiple repositories in order to construct universal version history. The results show that about 41.32% of source files (in the repository involving more than 6M versions of around 2M files) were duplicated among the Avaya's source code repositories for more than ten different projects. The prefix-postfix algorithm is more suitable than prefix algorithm due to the reasonable error rates after validation of the known copying behaviors.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement – Restructuring, reverse engineering, and reengineering

General Terms: Algorithms, Measurement

Keywords
Cloning, Version Control, Change History, Code copying, Code Authorship

1. INTRODUCTION
Software reuse is always a key factor to increase the speed of the software development. Studies done by Baker (1995) and Lague (1997) suggested that duplicated codes were 5% to 10% of the source codes of large programs. Most projects are implemented based on an existing piece of codes. Therefore, it is generally believe that code cloning or code coping is often used in developing industrial systems. Companies may use different version control systems to maintain different projects due to the evolution of version control systems and cross-organization work. Hence, cloned codes are often distributed among different version control systems.

A previous study [3] proposed that cloned sources were more reliable than non-clone code. In that particular study the maintenance cost of a duplicated module was less than for a non-duplicated module. Furthermore, knowing clones may help fix bugs in all cloned copies if a bug is detected in any one of them. Understanding how code is cloned allows us to identify the original authors of the program as well as determine the code sharing relationships and interdependencies between people and projects over time.

There are various kinds of cloned code detection methods that have been proposed. Baxter (1998) used abstract syntax trees to find out the cloned codes. Ducasse (1999) proposed a pattern matching technique that divided programs into strings and then the duplications were caught by comparing those strings against each other. Kamiya (2002) developed a language independent tool - CCFinder to find clone codes by matching token sequences that were transformed from source files by lexical analysis. In addition, in order to overcome the navigation difficulties of a huge set of results generated by clone detection tool, Kapser (2005) implemented a supportive tool for visualizing the clone codes detected from CCFinder. However, these methods might not be suitable to detect clones for source code version history repositories because they could take enormous computation efforts even on a single version of the code. With more than six million versions in the repository we have analyzed assuming average file size of 500 lines the CCFinder would require 13 days if computer memory is not exhausted. We base this estimate on the published 3 minute execution time for 500K lines of code assuming linear complexity for CCFinder. Furthermore, the clone size those techniques are targeting is fairly small in comparison to a large software project. Moreover, the copying behaviors of various versions of one project or projects to projects have not been discussed in the existing methods. Therefore, we look at algorithms that are appropriate for large code repositories involving version histories.

The purpose of this paper is to introduce two algorithms that were applied to detect the duplicated files in Avaya's version control repositories. A universal version history over different version control systems was built according to the identified copies of the files.

MSR'06, May 22–23, 2006, Shanghai, China.

2. METHOD

2.1 Methodology

It is computationally infeasible to detect cloned codes at the function or syntax level for a very large population of files. Therefore, our approach was to extract the possible clone code files by examining the full pathnames of source code files in the initial stage of clone detection and then find the duplicate code from the candidate files. This allowed us to avoid computationally intensive step of comparing the content of the files.

Base on the observations of product development we identified a number of scenarios that could cause identical files with different pathnames.

(1) A substantial amount of code copying takes place when a software project copies the code base for a new release or for a similar but distinct product.
(2) While such copies can be realized as branches in a version control system, some projects create separate instances of the version controlled files by copying them into a separate directory.
(3) A different version control repository is usually used if another organization is developing a similar product and wants to have their own version control repository for the code.
(4) In some cases, projects change the version control tools, forcing the change in repositories.

Our definition of duplicated source files is that files have at least one nonempty version between them (excluding spaces). In other words, let a_1, …, a_n be versions of file a and b_1, ..., b_m be versions of file b. Then a and b are duplicate if exists 1<=i<=n and 1<=j<=m, such that a_i is identical to b_j (excluding white spaces) and a_i is non-empty.

The methodology of this paper (see Figure 1) contains five primary steps. Firstly, a list of filenames was analyzed and then two algorithms were used to detect the common files. According to known copies, the algorithms were calibrated or validated. To trace authorship and version history of cloned files, a universal version history was finally created by connecting the histories of individual cloned source files.

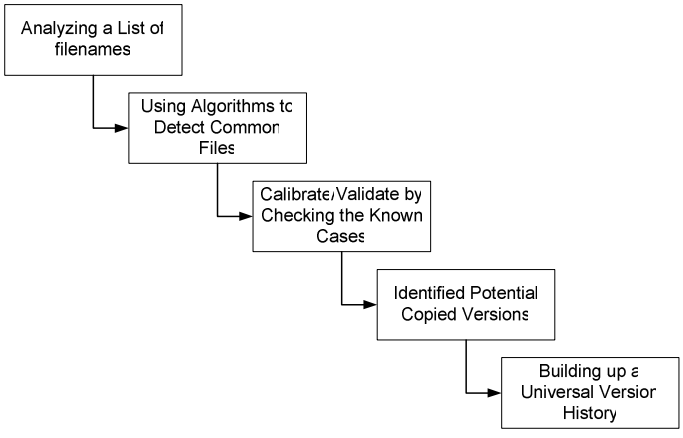

Figure 1. Approach procedure

2.2 Algorithms

Based on the observed developers' copying behaviors, in order to duplicate files, developers often copy the entire directory. Hence, the files sharing a name in any two folders may be identical. Therefore, the directories that share a large fraction of filenames should be identified first and then cloned files can be obtained by comparing the files with a common name in such directories. As a result, the goal of the following two algorithms is to retrieve the potentially copied directories.

2.2.1 Terminology

To simplify presentation we introduce some terminology first.

(1) Division level: the full file path can be broken into levels according to the '/' symbol (the repositories we investigated are mostly in UNIX file system. We have converted Windows '\' directory separator into UNIX '/' for the few version repositories kept in Windows file systems).
(2) Prefix and postfix: in the whole directory path of a file, the part of the path preceeding the division level is prefix and the remaining part of the path is postfix. For example, for a path D_i/D_j/D_k. If D_i is the prefix, D_j/D_k is the postfix if the division level is one.
(3) Common directory pair: if two directories are thought to be the copied, these two directories are called the common directory pair
(4) Common prefix pair: if two prefixes are thought to be the copied, these two prefixed are called the common prefix pair.

2.2.2 Prefix algorithm

The assumption of prefix algorithm is that directories i and j are considered as candidates for common directory pairs as long as there are at least certain portion of identical filenames in i and j. If Ni and Nj are the number of files in i and j as well as n_{ij} is the number of identical files in i and j, we use the following criteria in order to identify common directory pairs among different version repositories. The n_{ij} divided by N_{min} should be larger than a cutoff coefficient, where N_{min}= min(Ni, Nj). In other words, the fraction of files with identical names has to exceed a cutoff for a smaller directory in order to consider two directories to be potentially copied. We use this criterion to make sure that common filenames used in many unrelated projects such as, system.h, config.h, and main.c, do not affect our algorithm.

As long as those common directory pairs were extracted, a group of identical directories can be produced by using transitive law; that is, if we found a common directory pair (i,j) and a common directory pair (j,k), directories i, j, and k are identical. After a group of identical directories was gathered, those potential identical files can be matched under those identical directories.

2.2.3 Prefix - Postfix algorithm

The prefix – postfix algorithm is very similar to the prefix algorithm with one additional procedure. The new assumption is that prefixes i and j are considered as candidates for the common prefix pair if the result of the number of common postfixes following prefixes i and j divided by the minimum of the number of postfixes after prefixes i and j is larger than a cutoff coefficient of the criterion, the prefixes are considered to be common. Instead

of counting the fraction of common files in two directories as in the prefix algorithm, we are counting the fraction of postfixes.

Like gathering identical directories in prefix algorithm, those identical prefixes can be grouped and then the identical postfixes under the identical prefixes can be found.

2.3 Creating the Universal Version History

The files that are identified as copied are ordered according to the initial creation time of each file. The oldest one of them is assumed to be the main trunk of the universal version, while the other files are connected to the main trunk. If two files do not share the content for any version, they are considered not copied.

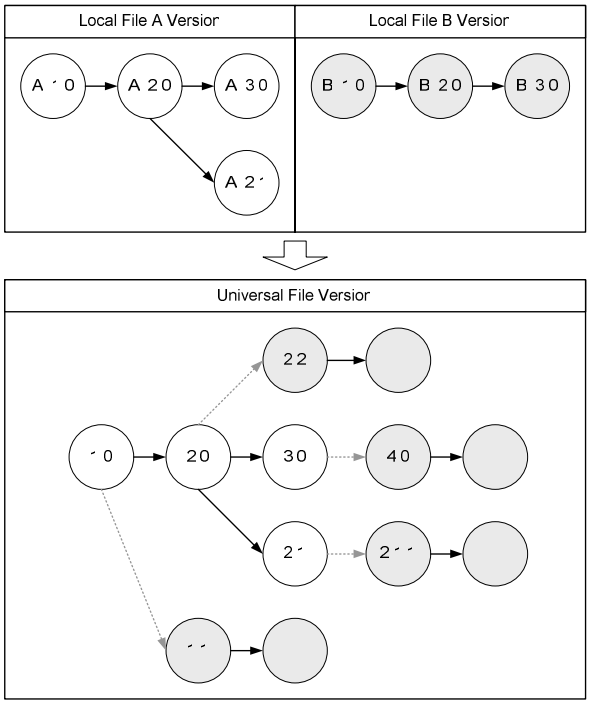

Figure 2. Universal version building procedure

For example, if two files, A and B are copied (see Figure 2), four possible merges of version histories may be appropriate depending of the versions of file A and File B that are identical. The assumption of this rebuilding process is that the first version of the later file should be copied from some version of the earlier file and then the later file is changed. Therefore, this approach has been simplified. We only connected the first version of the branch file to the main trunk file even though a later version of branch file was the same as any versions of the main trunk file. For instance, we do not connect 2.0 and 3.0 of B to any versions of A (see Figure 2).

3. RESULTS

The Avaya's version repositories we have analyzed contained more than 6M versions of more than two million source code files. From our previous work with various projects, we have identified the known cloned cases. We used the Type I & II error method to validate both algorithms. Type I error identifies the fraction of file pairs (i, j) not known to be copied that were identified by our algorithms as copied. Similarly, Type II error is the fraction of the pairs (i, j) known to represent copied files that our algorithm did not identify as copied. The population size of directory pairs is n(n-1)/2, where n is the number of the total directories we explored. Obviously, the number of d_i = d_j represents only a small part of the population; therefore, the Type I error for both algorithms were small. However, the Type II error of the prefix algorithm was fairly large (see Table 2, 3). Therefore we used prefix–postfix algorithm to identify the copied files.

Table 1. Type I and II errors of prefix – postfix algorithm

		Experimental results	
		d_i <> d_j	d_i = d_j
Known cases:	D_i <> D_j	2154385410	Type I error: 28804 (Rate: 1.34E-03%)
	D_i = D_j	Type II error: 822 (Rate: 1.5%)	53867

Table 2. Type I and II errors of prefix algorithm

		Experimental results	
		d_i <> d_j	d_i = d_j
Known cases:	D_i <> D_j	2153498853	Type I error: 915361 (Rate: 0.042%)
	D_i = D_j	Type II error: 20871 (Rate: 38.1%)	33818

An observation of the program file naming behavior indicates some common filenames that are often used in many unrelated software projects; for example, programmers often used “main.c” for the entry program of C or C++, “help.c” for the instruction file or “main.h” for include files. Hence, we excluded these high frequency filenames when counting the number of identical filenames (n_{ij}) between two directories. The experiment showed that about 41.32% of the total files were identical. These common files came from different projects and programming languages.

The algorithms are implemented in the Perl programming language and were run on a Sun V40Z machine that contains 16G RAM and dual Opteron CPUs, running SunOS 5.10. for processing more than 6M versions of more than 2M files. (Each file had its version history.) The approximate computational time of these two algorithms and the rebuilding procedure is shown in Table 3.

Table 3. Computation time comparison

		Time
Clones Extraction	**Prefix**	2.1 hours
	Prefix – postfix	5.1 hours
Universal Version Construction		10.2 hours

The whole universal building process includes two steps – (1) matching the identical files from the common directories that we

acquired via prefix-postfix algorithm; (2) integrating versions among the identical files. The total time of this process is about 10.2 hours.

4. DISCUSSION

We presented two practical algorithms for detecting the copied files in multiple large version control repositories. Although the prefix – postfix algorithm could provide more accurate results, it needed much more computation time than the prefix algorithm. These two detection methods identify copies only if filename does not change. Therefore, we cannot detect the copies if the programmers modified the filename. Besides, these methods may not be effective for other copying strategies that are not considered in our assumption, like programmers randomly copy the file without duplicating the entire directory. However, there are some advantages of this approach. It can identify clones over multiple versions, for various programming languages, and for multiple version control repositories. The processing speed of this method is also acceptable for identifying clones in multiple very large version repositories. The main value of constructing universal version history is to identifying code authorship and copying patterns over multiple large repositories.

5. FUTURE WORK

Although prefix – postfix algorithm provides some advantages for finding the clones of different programming languages in distributed huge repositories, the method can only identify identical source files; that means, similar but not identical files or copied source code segments cannot be found. This implies that those more detail level clone identifying methods are complimentary to our approach and could be combined in future studies.

Since we did not consider more complicated possible version history construction situations, a more complete universal version history building method should be further investigated.

The scripts used to implement the described algorithms are available upon request from authors.

6. ACKNOWLEDGMENTS

Our thanks to Avaya Labs Research for providing us all the resources.

7. REFERENCES

[1] Brenda Baker. On finding duplication and near duplication in large software system, IEEE Working Conference on Reverse Engineering 1995.

[2] B. Lague, D. Proulx, E. Merlo, J. Maryland, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, IEEE International Conference on Software Maintenance 1997.

[3] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software, Proceedings of the 8th International Symposium on Software Metrics 2002.

[4] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo SantAnna and Lorraine Bier. Clone detection using abstract syntax trees. In Proceedings of the 8th International Symposium on Software Metrics 1998.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. International Conference on Software Maintenance 1999.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering, Vol. 28, No.7, 2002.

[7] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. International Conference on Software Maintenance 2005.