社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

A Clone Detection Approach for a Collection of Similar Large-Scale Software Products

Eunjong CHOI†, Norihiro YOSHIDA††, Yoshiki HIGO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-shi, Osaka, 565–0871 Japan
†† Graduate School of Information Science, Nara Institute of Science and Technology
Takayama-cho 8916–5, Ikoma-shi, Nara, 630–0192 Japan
E-mail: †{ejchoi,higo,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

Abstract Reusing existing software with or without modifications is frequently occurred to develop new large software at low cost with high quality. So far, many techniques and tools have been proposed for detecting reused pieces in source code. However, existing tools have low scalability; they spend lots of memory and time to detect reused pieces on large-scale software. In this paper, we proposed an approach for detecting reused files as well as reused code fragments, code clones for a collection of similar large-scale software products. For the case study, we applied our approach to three OSS projects and compared code clone detection time between only using CCFinder and our approach. We found that our approach takes shorter time to detect code clones.

Key words hasing, code clone, open source system

1. Introduction

Software reuse, using existing software with or without modifications during the construction of a new software system, is frequently occurred to develop large-scale software at low cost with high quality. However, in some ways, these reused pieces occur annoying problems because if a defect is found in a reused piece, all of its same pieces should be checked for the same defect. This task takes a lot of time and effort, especially in large-scale software.

Detecting and managing reused pieces in software is important for not only detecting bugs but also understanding code quality, plagiarism detection, copyright infringement investigation. Many tools have been proposed for detecting reused pieces in software at code fragment level, so called code clones [5], [6], [7]. However, existing clone detection tools have a low scalability, these tools take numerous times and memory to detect code clones on large-scale of software. To overcome this limitation, several tools have been proposed and implemented but their scalability is still not high.

In case of detecting reused files, our previous study suggested an approach for detecting reused file without any (or just slight) modifications in comments or headers on large-scale software [12]. We used hash computation of the tokenized files and implemented a tool named FCFinder (File Clone Finder). In case study, we detected reused files in the FreeBSD Ports Collection in 17.16 hours with a single workstation and investigated characteristics of them. However, the goal of this study is not detecting code clones or reused files but investigating characteristics of sets of reused files.

This paper suggests an approach for detecting code clones for a collection of similar large-scale software products. Our approach is similar to previous study; We also compute hash values of files to detect reused files. However, this study is different from the previous study on following two aspects (1) the outputs of our approach are not only reused code fragments, code clones but also reused file. (2) we only detected sets of files that are reused in other collections without modifications.

To detect code clones, we use CCFinder which detects code clones on token-base [7]. As case study, we compare clone detection time between using our proposed approaches and only using CCFinder in three Open Source Software (OSS) projects. As a result of the case study, we found that our approach is much faster than only using CCFinder to detect code clones on large-scale software.

The rest of this paper is organized as follows. Section 2. explains the background of this study, definition of code clone and its related terms, and CCFinder, a token-based code clone detection tool. Section 3. explains our approach. Section 4. describes the case study and its results. Section 5. explains related work. Section 6. summarizes this paper and discusses future work.

2. Background

This section explains definition of code clone and its related terms, CCFinder, a token-based code clone detection tool, to give a clear understanding of this study.

2.1 Code Clone

A code clone is a code fragment that has lexically, syntactically, or semantically similar code fragments in source code. Code clones are created because a programmer sometimes intentionally reuses existing code when she develops a new hardware drivers, or new platforms to improve the maintainability of source code. Meanwhile, she unintentionally creates code clones due to programming idioms or algorithmic idioms [8]. Many code clones are contained in software systems. In large-scale software, 13 - 20 percent of code clones are contained [1].

Code clones are categorized following types base on the similarity of text [2]:

Type 1: Identical code fragments except for variations in whitespace, layout and comments.

Type 2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type 3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

We call a pair of code cones as a *Clone Pair* and a set of code clones that are identical or similar to each other as a *Clone Set*.

2.2 CCFinder

Many studies suggested code clone detection technics and its implemented tool [4], [5], [6], [7], [10]. Between them, token-based techniques have high the recall and scalability [2], [9]. One of the famous token-based code clone detection tools is CCFinder which is developed by Kamiya et al [7]. It takes a source files as an input and outputs clone pairs information. It can report Type 1 and Type 2 clones in several languages (e.g. C, C++, Java, COBOL).

The process of CCFinder consists of following four steps:

(1) **Lexical Analysis.** Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis. At this step, the white spaces (including "\n" and "\t" and comments) between tokens are removed from the token sequence, but those characters are sent to the formatting step to reconstruct the original source file.

(2) **Transformation.** The token sequence is transformed, (i.e., tokens are added, removed, or changed based on the transformation rules) and then, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code portions with different variable names to become clone pairs. At the same time, the mapping information from the transformed token sequence into the original token sequences is stored for the formatting step which comes later.

(3) **Match Detection.** From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs. A suffix-tree matching algorithm [3] is used to compute matching, in which the clone location information is represented as a tree with sharing nodes for leading identical subsequences and the clone detection is performed by searching the leading nodes on the tree.

(4) **Formatting.** Each location of clone pair is converted into line numbers on the original source files.

Moreover, if the total size of source files is too large to build a single suffix-tree on primary store, it provides a 'divide and conquer' approach. The input source files are divided into disjoint subsets. For each pair of the subsets, a sub suffix-tree is built to extract clone-pairs. The total collection of clone-pairs is the final output. However, it still takes numerous time to detect code clones on large-scale software, especially in building and matching suffix-tree.

3. Proposed Approach

The overview of this approach is described in Figure 1. At first, hash values of input files are created. Then two categories, identical file sets (i.e.. sets of files that are identical each other), and target files are created. Then, code clones on target files are detected. Finally, overall clone sets are created by matching detected clone sets and identical file sets. This approach takes a source files as an input and outputs overall clone sets and identical file sets. We will explain each step with an example of input files $F_i = \{F_1, F_2, F_3, \cdots, F_n\}$.

(1) **Calculate MD5 Hash.** This step creates input files' MD5 hash values [11]. We select MD5 as a hash function because it does not require any large substitution tables can be coded quite compactly. Let us assume that as a result of input files F's MD5 hash computations we obtain a set of hash values H = $\{H_a, H_a, H_b, \cdots, H_k\}$ $(k \leqq n)$. This means that MD5 hash values of files F_1, F_2 are the same. Meanwhile, file F_3 has unique MD5 hash value.

(2) **Formatting.** In this step, a hash table is created with calculated MD5 hash values as keys and its corresponding sets of file as values. If files have same MD5 hash value, they are added to the same value as a set. For example, files F_1, F_2 have H_a as a MD5 hash value, they are added as values in hash table whose hash key is H_a. Meanwhile, file F_3

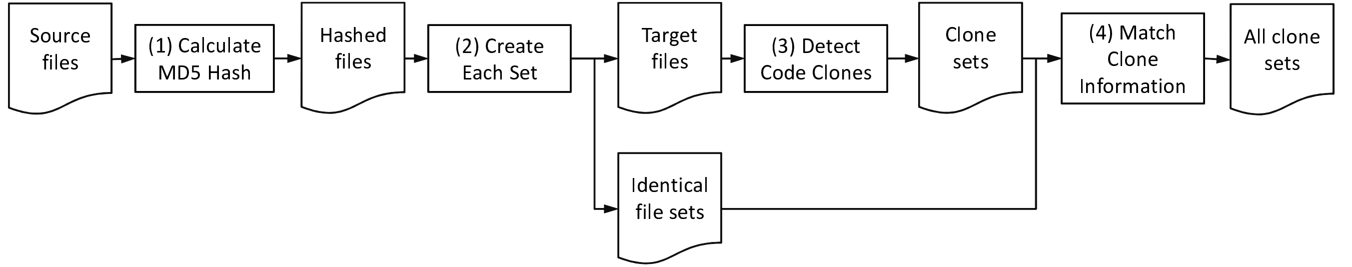

Figure 1 An overview of our research

path is added as a value in a hash table whose hash key is H_b.

(3) **Create Each Set.** This step creates target files T and identical file sets SI based on the information on a hash table. At first, a number of each MD5 hash value in hash table is checked; If the number of MD5 hash value is more than two, their corresponding key in the hash table is added to target files T. Meanwhile, if the number of value is one, its corresponding key of the hash table is added to identical file sets SI. In addition, the first file of each identical file set is added to target files T. This added file will be used in the later step.

In example, first, files F_1, F_2 are added to identical file sets SI and file F_3 is added to target files T. In this situation, each set contains following files;

- Identical file set $SI = \{F_1, F_2, \cdots\}$
- Target files $T = \{F_3, \cdots\}$

Next, F_1,the first file of each identical file set, is added to target files. Therefore, final sets contains following files:

- Identical file set $SI = \{F_1, F_2, \cdots\}$
- Target files $T = \{F_3, F_1, \cdots\}$

(4) **Detect Code Clones.** A clone set on target files T are detected in this step. To detect code clones, we selected CCFinder as a code clone detection tool because it detects code clones with high speed. Moreover, it has high accuracy for detecting code clones. In example, let us assume that a clone set C $= \{c_\alpha, c_\beta, \cdots\}$ are detected from target files T $= \{F_3, F_1, \cdots\}$.

(5) **Match Cone Information.** This step creates all clone sets by matching detected clone sets on added file that is the first file of each identical file set and identical file set SI. assure that that because the files in the same set of identical files are exact each other, the location of clone of each file should be located in the same location. Therefore, if code clone are detected on the first file of each identical file set, we added the code fragments in other files of the same identical file set to the same clone set.

In example, code clone c_β is detected on file F_1 in previous step. Because files F_1, F_2 are identical each other, we sure that code clone is also located in file F_2 in the same location as code clone c_β in file F_1. Let us assume that code clones in the file F_2 is c_γ, then, contains code clone c_β, Therefore, code clone c_γ is added to the same clone set where code clone c_β is contained. All clone sets C consits of $\{c_\alpha, c_\beta, c_\gamma \cdots\}$

4. Case Study

In case study, we detected code clone on three large-scale open source software(OSS) projects using our approach and only using CCFinder. We used 30 tokens as the minimum length of token sequence of a code clone to CCFinder in both approaches. The goal of this case study is to compare code clone detection time between our approach and only using CCFinder.

This case study was performed on a 64 bits Windows 7 Professional workstation equipped with 2 processor, 2.67GHz and 2.66GHx CPUs and 24 gigabytes of main memory.

4.1 Target System

A summary about the target systems is described in Table 1. We selected three OSS projects, Apache Ant(1), Linux kernel(2) and GT-B5510 model(3). Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. We selected only java files in the src/main directory from 16 sequence versions (1.5.2-1.8.4). Linux kernel is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It originally developed first for 32-bit x86-based PCs (386 or higher), now, it supports multiple architectures(e.g. AMD x86-64, IBM S/390).We selected only c files in the fs directory from 12 sequence versions (2.6.0-2.6.10). GT-B5510 model is a Samsung mobile phone, Samsung Galaxy Y Pro. We selected three versions of this model for different countries, Euro, Brazil and China. We selected only c files in the kernel directory.

4.2 Result and Discussion

Target files and identical file sets are created after "Creat-

(1): http://ant.apache.org/
(2): http://www.kernel.org/
(3): http://opensource.samsung.com/index.jsp

Table 1 Target Systems

Project Name	Language	#Files	Lines Of Code
Apache ant	Java	11,393	2,971,948
Linux kernel	C	10,673	6,191,838
GT-B5510 model	C	86,986	41,982,594

ing Each Set" Step. The number of created target files and identical file sets are shown in Table 2. The colum Identical Files represents the information of detected identical file sets. Apach ant contains 3,083 sets of identical files. Overall number of identical files is 8,667 and they are 76.1 percent of input source files. Linux kernel contains 1,967 sets of identical files. Overall number of identical files is 9,712 and they are 91.0 percent of input source files. GT-B5510 model contains 28,181 sets of identical files. Overall number of identical files is 9,712 and they are 99.8 percent of input source files. Between them, GT-B5510 model contains the most identical files. We assume that this is because many of files are reused in GT-B5510 model and only specific functions for each country are implemented.

The number of detected clone sets by our approach are also shown in Table 3. 10,692, 21,343 and 148,761 clone sets were detected from Apache ant, Linex kernel, and GT-B5510 model, respectively as a result of our approach. The number of clone sets by only using CCFinder are also shown in Table 3.

Henceforth, We will discuss our results in terms of detection time and accuracy of results.

（1） **Detection Time** The code clone detection time of only using CCFinder and our approach is described in Table 3. As shown in Table 3, in Apache ant, detection time of CCFinder is 241 seconds. Meanwhile, detection time of our approach was 89 seconds. Moreover, in Linux kernel and GT-B5510 model, the detection time of CCFinder was 1,119 and 113,445 seconds, respectively. Meanwhile, the detection time of our approach was 168 and 113,445 seconds, respectively.

In Linux kernel and GT-B5510 model, our approach detects code clones ten time faster than only using CCFinder. We believe that our approach detects code clones faster, if the scale of target system is more large.

（2） **Accuracy of Results** To determine accuracy of the results derived by our approach, we arbitrary selected 30 clone sets that are detected by our approach from each OSS project project. Then, we use the output of CCFinder from the same projects. We found that all selected clone sets are also detected by only using CCFinder.

We also selected 30 identical file sets from each OSS project. We manually checked all of them and found that files in the same identical file sets are really identical each other.

Table 2 Target files and outputs for each OSS project

Project Name	Target Files		Identical Files	
	#Files	Lines of Code	#Files	# Sets
Apache ant	3,083	1,729,505	8,667	3,083
Linux kernel	2,928	2,383,758	9,712	1,967
GT-B5510 model	28,327	14,152,711	86,840	28,181

5. Related Work

Many technique and its implemented tools have been proposed for detecting code clones on large-scale software.

Baker proposed a tool named Dup to detect text-based and line based code clones [1]. Dup ignores comments, indentation and white space. It detects exactly same clone pairs or parameterized match code fragments (i.e., code fragments that are comprised of neighboring sequences of source lines and variable names or structures member in one code fragment are consistently changed into other code fragment). sf Dup found 20 percent matches of lenght at least 30 lines on the Window System subsystem (SS) which has almos 1.1M lines.. The cpu time of SS was 7.9 minutes on the SS with one 40MHz R3000 processor (primary I and D cache 64KB, secondary 1MB, main 256MB, SGI IRIX 4.1)

CP-Miner, proposed by Li et al. detects token-based code clones based on frequent subsequent mining which is an association analysis technique to discover frequent subsequences in a collection of sequences [10]. Enhanced algorithm, CloSpan [13] allows CP-Miner to tolerate one to two statement insertions, deletions, or modifications in copy-pasted code, ignoring an arbitrarily long different code segment that is unlikely to be copypasted code. CP-Miner found 190,000 and 150,000 copied code clones that account for 20-22 percent of the source code in Linux and FreeBSD within 20 minutes.

Hummel et al. proposed index-based code clone detection approach [5]. It detects Type 1 and Type 2 clones by using MD5 hash value to calculate hash values from normalized statements for each input file and retrieving code clones from the databases where the hash values are stored. In the case study, 100 machines performed clone detection in 73 MLOC of open source code in 36 minutes.

Table 3 Clone sets and detection time(seconds) from CCFinder and our approach.

Project Name	CCFinder		Our approach	
	#Clone Sets	Time	#Clone Sets	Time
Apache ant	11,169	241	10,692	89
Linux kernel	24,235	1,119	21,343	168
GT-B5510 model	325,274	113,445	148,761	11,902

6. Summary and Future Work

In this paper, we suggest an approach for detecting code clones on for a collection of similar large-scale software products. Our approach uses MD5 hash to find identical file sets and CCFinder to detect code clones. The input of this approach is source files and outputs are overall clone sets and identical file sets. In case study, we applied our approach to three OSS projects and compared code clone detection time between only using CCFinder and our approach. We found that our approach takes shorter time to detect code clones.

There are a number of areas in which we would like to expand upon this study. At first, our current approach successfully detected reused identical file sets, but, it cannot categorize files might be reused with slightly modification (i.e., change identifier name or comments) as identical files. Therefore, we plan to improve our approach for detecting reused files with slightly modification as identical file sets.

Although we applied our approach to three OSS projects and we found that our approach faster than CCFinder to detect code clones in large-scale software. The result might be changed to other software projects. Therefore, we plant to apply our approach to various size of software in different domains.

Finally, our approach uses CCFinder to detect code clones and we compared our approach with only CCFinder in the case study in this paper. Using other code cloning detection tools might be efficient than CCFinder in detecting large-scale software and the result of case study also might be changed if we compare other tools. We believe that CCFinder is appropriate for detecting code clones on large-scale software, but we consider introducing other code clones detection tools and compare results from them in the case study.

7. Acknowledgments

This work is partially supported by JSPS, Grant-in-Aid for Scientific Research (A) (21240002).

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of WCRE*, pages 86–95, July 1995.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[3] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[4] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *Proc. of WCRE*, pages 3–12, October 2011.

[5] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proc. of ICSM*, pages 1–9, September 2010.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE*, pages 96–105, May 2007.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[8] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empir Software Eng*, 13(6):645–692, 2008.

[9] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of WCRE*, pages 253–262, October 2006.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[11] R. L. Rivest. The MD5 message digest algorithm. Internet RFC 1321, April 1992.

[12] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. In *Proc. of MSR*, pages 102–105, May 2010.

[13] X. Yan, J. Han, and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *Proc. SIAM Int'l Conf. Data Mining*, pages 166–177, May 2003.