PAPER

Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files

Eunjong CHOI[†a)], ***Nonmember***, **Norihiro YOSHIDA**[††], **Yoshiki HIGO**[†], ***and*** **Katsuro INOUE**[†], ***Members***

SUMMARY So far, many approaches for detecting code clones have been proposed based on the different degrees of normalizations (e.g. removal of white spaces, tokenization, and regularization of identifiers). Different degrees of normalizations lead to different granularities of source code to be detect as code clones. To investigate how the normalizations impact the code clone detection, this study proposes six approaches for detecting code clones with preprocessing input source files using different degrees of normalizations. More precisely, each normalization is applied to the input source files and then equivalence class partitioning is performed to the files in the preprocessing. After that, code clones are detected from a set of files that are representatives of each equivalence class using a token-based code clone detection tool named CCFinder. The proposed approaches can be categorized into two types, approaches with non-normalization and normalization. The former is the detection of only identical files without any normalization. Meanwhile, the latter category is the detection of identical files with different degrees of normalizations such as removal of all lines containing macros. From the case study, we observed that our proposed approaches detect code clones faster than the approach that uses only CCFinder. We also found the approach with non-normalization is the fastest among the proposed approaches in many cases.
key words: *code clone, hash function, source code transformation*

1. Introduction

Recently, electronics companies release a new model in regular and rapid rushed intervals [1]–[3]. To release a new model within a short time, a lot of companies develop the model by reusing existing files with or without modifications. This activity provides the benefit of saving time and cost as well as avoidance of a high risk for creating new code logic. Whereas, it generates many identical or similar files between different versions and models, which make software systems difficult to be maintained.

A code clone is a code fragment that has identical or similar code fragments in software systems. It is important to detect code clones from different release versions and models. For example, when a defect is contained in a code clone in the one version/model, all of its cloned code fragments in the other versions/models should be inspected for the same bug. This task takes much time and effort, especially in large-scale software system. Up to date, researchers have proposed code clone detection approaches that use various granularities such as line, token, and abstract syntax tree and evaluated them to find out the effective one [4], [5].

Different degrees of normalizations (i.e. transformation of program elements) for detecting code clones have been also proposed. Each normalization make subtly different similar source code to be detected as code clones. For instance, code clone detection tool named Dup normalizes the input source files by tokenizing each file into a single token sequence [6]. This normalization leads to detect source code with different white space, layout, and comments as code clones. A token-based code clone detection tool named CCFinder normalizes the input source files by replacing identifiers related to types, variables, and constants by a special token and then concatenating all tokens in the same file into a single token sequence [7]. This normalization leads to detect source code with different identifiers, white space, layout, and comments as code clones. Different degrees of normalization cause different granularities of source code to be detect as code clones, but only a little has been known about how the normalizations impact the code clone detection [8].

To investigate how the normalizations impact the code clone detection, this study proposes six approaches that detect code clones with preprocessing using different degrees of normalizations and evaluates them. More precisely, each normalization is applied to the input source files and then equivalence class partitioning is performed to the files based on the MD5 hash function in the preprocessing. The goal of this preprocessing is to avoid irrelevant code clone detection caused by the identical files. These identical files increase computational complexity of code clone detection because code clones are repeatedly detected within them. The proposed approaches can be categorized into two types, approaches with non-normalization and normalization. The former category is the detection of code clones based on the identical files without any normalization. Meanwhile, the latter category is the detection of clones based on the different degrees of normalizations such as removing macro from the input source files. After preprocessing, code clones are detected only on **corpus** (i.e. a set of files that are representatives of each equivalence class) by CCFinder. In case study, the our proposed approaches as well as an approach that uses only CCFinder are applied to different versions of three Open Source Software (OSS) systems. The contributions of this paper are summarized as follows:

- We found that any proposed approach with preprocessing input source files is faster than the approaches that

Manuscript received August 25, 2014.
Manuscript publicized October 28, 2014.
[†]The authors are with Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with Nagoya University, Nagoya-shi, 464–8601 Japan.
a) E-mail: ejchoi@ist.osaka-u.ac.jp
DOI: 10.1587/transinf.2014EDP7292

uses only CCFinder.
- We also found that any normalization takes much time in preprocessing and post-processing and is unable to reduce total detection time in many cases.
- We have developed proposed and implemented code clone detection approaches with preprocessing input source files.

The remainder of this article is organized into the following sections. Section 2 explains background for this study. Next, Sect. 3 details our proposed code clone detection approaches. Section 4 describes our case study on different versions of three OSS systems. Section 5 explains threats to validity. Section 6 reviews related work and finally, Sect. 7 concludes with possible future work.

2. Background

The purpose of this section is to explain background for this research. Code clone will be explained in Sect. 2.1 and then CCFinder, a token-based code clone detection tool will be described in Sect. 2.2.

2.1 Code Clone

A code clone is a code fragment that has identical or similar code fragment(s) to it in the source code [4]. A set of code clones that are identical each other is a clone set. Code clones are categorized into the following types based on the textual similarities between code clones:

Type-1: Identical code fragments except for variations in white space, layout and comments.

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

Detecting code clones is important for not only detecting bugs [9] but also understanding code quality [10], and plagiarism detection [11], [12].

2.2 Code Clone Detection Tool: CCFinder

CCFinder is a representative token-based code clone detection tool developed by Kamiya et al. [7]. It has been adopted in various companies and studies [13]–[15]. It detects Type-1 and Type-2 code clones. The process of CCFinder is comprised of the following four steps [7]:

1. **Lexical Analysis:** Each line of input source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all the 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 input 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 source code with different variable names to become code clone. At the same time, the mapping information from the transformed token sequence in the original token sequences is stored in the **Formatting** step which comes later.
3. **Match Detection:** From all the substrings on the transformed token sequence, equivalent code fragments are detected as code clone. A suffix-tree matching algorithm [16] 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 code clone is converted into line numbers of the original input files.

```
#version: ccfinder 7.3.2
#format: classwise
#langspec: JAVA
#option: -b 30
#option: -e char
#option: -k 30
#option: -r abcdfikmnoprsuv
#option: -c wfg
#option: -y
#begin{file description}
0.0     1564    2811    C:¥SourceFile¥ClassLoader.java
0.1     388     745     C:¥SourceFile¥TypeDefinition.java
    ...

#end{file description}
#begin{clone}
#begin{set}
0.0     776,13,1161     781,41,1196     33
0.0     990,15,1655     996,41,1690     33
#end{set}
#begin{set}
0.0     784,15,1204     797,37,1255     50
0.0     1003,15,1710    1015,37,1761    50
#end{set}
    ...

#end{clone}
```

Fig. 1 Example of an output file of CCFinder.

The output of CCFinder consists of three sections: option, file description, and clone descriptions. Option section contains specified programming language, the minimum length of the token sequence of code clones, and the other options. Users can utilize these options by invoking CCFinder directly at the command line. File description section represents information of each input source file such as file number (i.e. the number assigned to each file), line of source code, size of tokens. Finally, clone description sec-

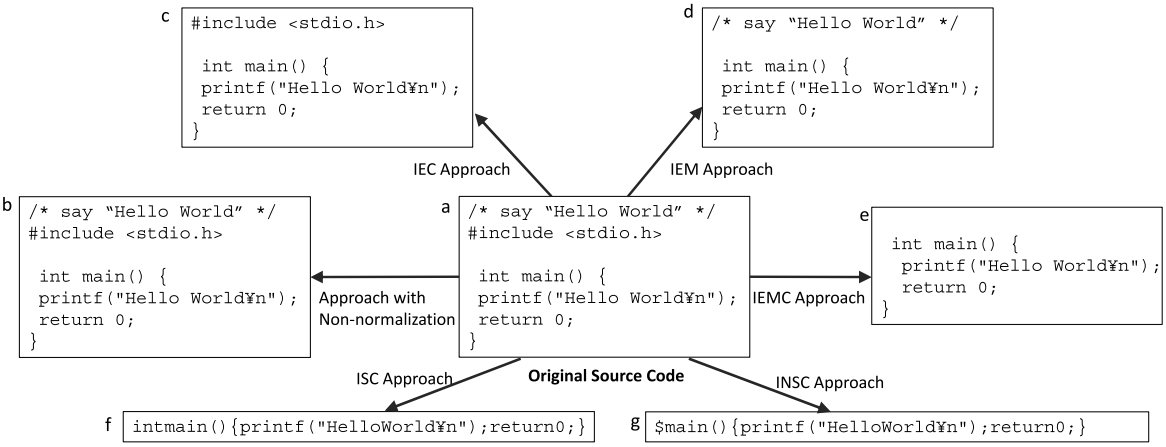


Fig. 2 Example of result of each normalization.

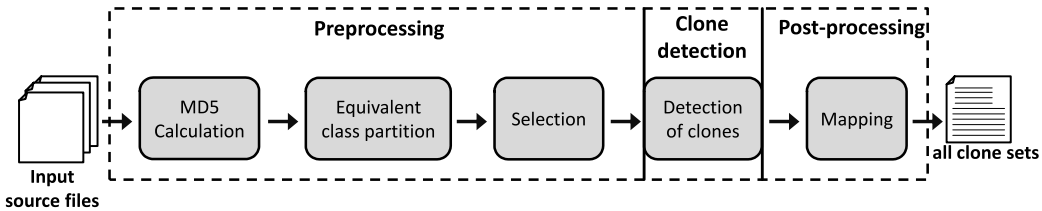


Fig. 3 Overview of approach with non-normalization.

tion represents information of code clones in each detected clone set on the input source files. This section represents clone information such as file number of detected file, location where code clone begins and ends.

Figure 1 shows an example of the output. In this figure, a code clone range from the 1161th token exists in the 13th column of line 776 to the 1196th token exists in the 41th column of line 781 and a code clone range from the 1655th token exists in the 15th column of line 990 to the 1690th token exists in the 41th column of line 996 consist a clone set.

However, it takes much time to detect code clones if the total size of input files is huge. For instance, it would take about 40 days to detect code clones on 400 million lines of input files [17].

3. Proposed Approaches

This study proposes approaches that detect code clones with different preprocessing and evaluate them. Our proposed approaches can be categorized into two categories: approach with non-normalization (see Sect. 3.1) and normalization (see Sect. 3.2). The former is the detection of code clones based on identical files without any normalization. Meanwhile, the latter category is the detection of code clones based on identical files with different degrees of normalizations. All approaches share following pipeline phases:

i. **Preprocessing:** It performs equivalence class (i.e. a set of files that are identical each other based on the hash values) partition and then generates corpus based on the MD5 hash values of the input source files. This study adopted MD5 hash function because its probability of an accidental collision is extremely small.
ii. **Clone detection:** It detects code clones on the corpus using CCFinder. To detect code clones, this study uses CCFinder because of its high accuracy of detecting code clones.
iii. **Post-processing:** It generates all clone sets by mapping output of CCFinder, the equivalence classes, and other information if necessary.

In the following sections, the detailed of the approaches with non-normalization and normalization will be explained in Sects. 3.1 and 3.2 respectively.

3.1 Approach with Non-normalization

This approach identifies identical files without any normalization. For example, Figs. 2 (a) and 2 (b) are non-normalized identical files in this approach. After that, code clones are detected based on the identical files. Figure 3 illustrates an overview of the three phases of this approach. The detail of each phase is as follows:

a. **Preprocessing:** For each input source file, MD5 hash value of the file is calculated. Then, equivalence class partition is performed based on the hash values. Namely, any files that have the same MD5 hash values are partitioned into the same equivalence class. Figure 4 depicts an example of the partition. In this figure, characters written on the files represent the hash

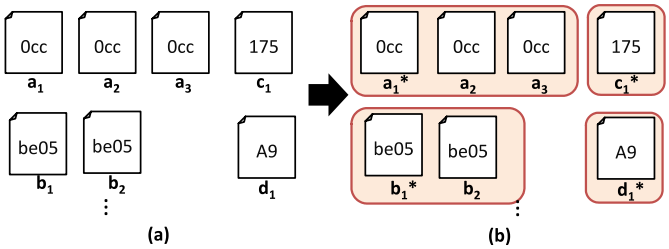


Fig. 4 Example of equivalence class partition and selection.

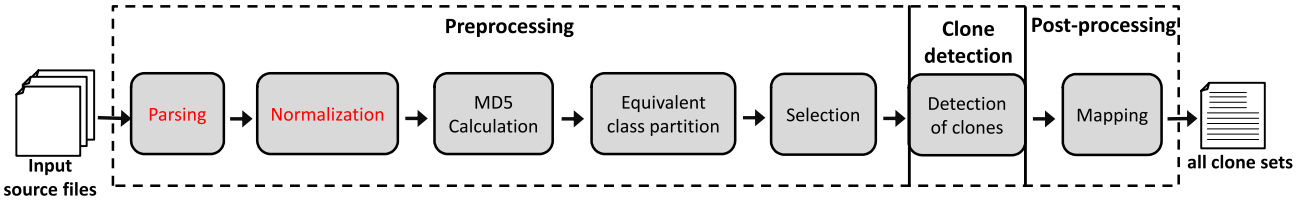


Fig. 5 Overview of approach with normalization.

values and rounded rectangle with red border represents each equivalence class. Taking a closer look at the figure, files a_1, a_2, and a_3 that have the same hash value 'Occ' are partitioned into the equivalence class. Files b_1 and b_2 that have the same hash value 'be05' are also partitioned into the equivalence class. In addition to this, file c_1 and d_1 is partitioned into each singleton equivalence class. After the partition, a file is selected from each equivalence class as a representative and then added to the corpus. Figure 4 depicts an example of the selection. In this figure, files with asterisk indicate that they are contained in the corpus. That is, files a_1, b_1, c_1, and d_1 were selected and then contained in the corpus.

b. **Clone Detection:** Code clones are detected on the corpus using CCFinder.

c. **Post-processing:** It is easy to assume that if a code clone exists at one file in an equivalence class, code clones also exist in the same place at the other files in the same equivalence class. Thus, in this phase, all clone sets is generated based on this assumption. That is, if a code clone was detected at the representative of an equivalence class in the clone detection phase, then code fragments in the same place at the other files in the same equivalence class are also added into the same clone set as code clones.

3.2 Approach with Normalization

This category contains approaches with different degrees of normalizations as follows:

- Identical Except for Comments (IEC) approach
- Identical Except for Macro (IEM) approach
- Identical Except for Macro and Comments (IEMC) approach
- Identical Source Code (ISC) approach
- Identical Normalized Source Code (INSC) approach

These approaches require additional processes compared to the approach with non-normalization explained in Sect. 3.1. That is, they parse the input source files into tokens and then save token information in the preprocessing phase. Figure 5 illustrates an overview of three common phases of these approaches. In this figure, additional processes are shown in the red font. The detail of each phase is as follows:

1. **Preprocessing:** The input source files are parsed into tokens and then following information is extracted from each file:
   - Token list: a list of tuples (token number, start column number, end column number, and line number) of each token, where
     - Token number: the number assigned to each token.
     - Start column number: the column number where token starts.
     - End column number: the column number where token ends.
     - Line number: line number where token exists.

   Then, one of the following normalizations is applied to each input source file:
   - For IEC approach: All lines containing only comments, comments, and white spaces before and after comments are removed from each input source files. This normalization transforms Figs. 2 (a) into 2 (c).
   - For IEM approach: All lines containing only macros are removed from each input source files. This normalization transforms Figs. 2 (a) into 2 (d).
   - For IEMC approach: All lines containing only

macros and comments, comments, and white spaces before and after comments are removed from each input source files. This normalization transforms Figs. 2 (a) into 2 (e).
- For ISC approach: Tokens in same file are concatenated into a single token sequence. This normalization transforms Figs. 2 (a) into 2 (f).
- For INSC approach: Tokens of identifiers, literals, types are replaced by a special token, and then tokens in same file are concatenated into a single token sequence. This normalization transforms Figs. 2 (a) into 2 (g). In this figure, a identifier 'int' is replaced by $.

The rest of this phase are the same with preprocessing in the approach with non-normalization explained in Sect. 3.1-a. Any files that have the same hash values are partitioned into the same equivalence class. After the partition, a file is selected from each equivalence class as a representative and then added to the corpus.

2. **Clone Detection:** Code clones are detected on the corpus using CCFinder.
3. **Post-processing:** As a result of normalizations in the preprocessing phase, code clones might exist in the different places between the files within the same equivalence class. Therefore, if a code clone was detected at the representative of an equivalence class, then the mapping is performed as follows:
   a. Start token number (i.e. the first token number of code clone) and end token number (i.e. the last token number of code clone) at the representative are identified.
   b. Start column number and line number of the corresponding start token number in other files in the same equivalence class are identified based on the token list saved in the preprocessing.
   c. End column number and line number of the corresponding end token number in other files in the same equivalence class are also identified based on the token list.
   d. Code fragments range from the identified start column number at the line number to identified end column number at the line number in other files in the same equivalence class are added into the same clone set as code clones

4. Case Study

In the case study, our proposed approaches as well as the approach that uses only CCFinder are applied to different versions of three OSS systems. Note that our proposed approaches detect code clones by excluding identical files within the same equivalence class, therefore, for case study, we select different versions of the same software system as subject systems because they contain many identical files. In particular, we design our case study to address the following two Research Questions (RQs):

- RQ1. Can proposed approaches detect code clones faster than an approach that uses only CCFinder?
- RQ2. Which approach is the fastest among the proposed approaches?

In the case study, we used 30 tokens (a default setting) as the minimum length of the token sequence of a code clone to CCFinder. During the case study, each approach is executed three times to get reliable results. This case study was performed on a 64 bit Windows 7 Professional workstation equipped with 2 processor, 2.27GHz and 2.26GHz CPUs and 24 gigabytes of main memory.

Table 1 Statistics of subject systems.

Name	#Versions	#Files	Line of code	#Tokens
Apache Ant	29	18,708	4,862,102	8,404,790
Linux kernel	12	7,839	5,690,967	12,537,555
Samsung Galaxy	2	29,573	19,920,387	43,924,235

4.1 Subject Systems

As subject systems, we selected three OSS systems of different size and application domain: *Apache Ant*[†], *Linux kernel*[††], and *Samsung Galaxy*[†††]. An overview of these systems is shown in Table 1.

Apache Ant is a Java library and command-line tool for building system written in Java. From this system, we selected Java files under a directory named *main* from 29 consecutive release versions (release version 1.1 to 1.9.4). *Linux kernel* is a clone of the operating system UNIX written in C. From this system, we selected C files having the file extensions .c, .cc, .cpp, and .cxx under a directory named *fs* from 12 consecutive release versions (release version 2.6.0 to 2.6.10). *Samsung Galaxy* is a Samsung mobile phone named Samsung Galaxy Y Pro written in C. We selected two release versions of this model for different area, Latin America, and China. From this system, we selected C files having the file extensions .c, .cc, .cpp, and .cxx under a directory named *common* from each version. Note that files that are lexically incomplete are excluded.

4.2 Results

This section illustrates results of the case study to answer the RQs.

4.2.1 Comparing with the Approach that Uses only CCFinder

To answer RQ1, this study compares our proposed approach with the approach that uses only CCFinder with respect to the detection time. Tables 2, 3, and 4 list detection time

[†]http://ant.apache.org/
[††]http://www.kernel.org/
[†††]http://opensource.samsung.com/index.jsp

Table 2 Detection time in seconds (Apache Ant).

Approach Names	Total detection time	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	716	-	-	-
Approach with non-normalization	253	3 (1.19%)	248 (97.89%)	2 (0.79%)
IEC Approach	232	17 (7.34%)	103 (44.60%)	111 (48.06%)
ISC Approach	214	13 (6.07%)	100 (46.73%)	101 (47.20%)
INSC Approach	214	14 (6.54%)	100 (46.57%)	100 (46.88%)

Table 3 Detection time in seconds (Linux Kernel).

Approach Names	Total detection time	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	1,058	-	-	-
Approach with non-normalization	175	2 (1.14%)	172 (98.29%)	1 (0.57%)
IEC Approach	336	23 (6.74%)	172 (51.14%)	142 (42.12%)
IEM Approach	344	26 (7.56%)	176 (51.11%)	142 (41.34%)
IEMC Approach	333	22 (6.71%)	172 (51.70%)	138 (41.58%)
ISC Approach	328	18 (5.49%)	168 (51.37%)	141 (43.13%)
NSC Approach	335	21 (6.26%)	172 (51.39%)	142 (42.35%)

Table 4 Detection time in seconds (Samsung Galaxy).

Approach Names	Total detection time	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	19,622	-	-	-
Approach with non-normalization	4,326	7 (0.16%)	4,307 (99.56%)	12 (0.28%)
IEC Approach	8,803	204 (2.31%)	4,686 (53.23%)	3,913 (44.46%)
IEM Approach	9,240	271 (2.93%)	4,601 (49.79%)	4,368 (47.28%)
IEMC Approach	8,711	227 (2.60%)	4,530 (52.01%)	3,954 (45.39%)
ISC Approach	8,513	242 (2.84%)	4,398 (51.67%)	3,873 (45.49%)
INSC Approach	8,894	234 (2.63%)	4,552 (51.18%)	4,108 (46.19%)

of the proposed approach compared to the approach that uses only CCFinder in *Apache Ant*, *Linux kernel*, and *Samsung Galaxy* respectively. Note that "IEM Approach" and "IEMC Approach" are conducted based on the macro in C, thus these approaches only applied to *Linux kernel* and *Samsung Galaxy*. In these tables, the column Total detection time represents detection times needed to complete the each approach. The columns Preprocessing, Clone detection, and Post-processing show detection time of each phase. In these columns, numbers in parenthesis represents their proportion in total detection time. Note that these tables show average of detection time of three executions.

As shown in these tables, our proposed approaches reduce code clone detection time compared with the approach that uses only CCFinder in any subject system. In particular, we can observe that the detection time of the proposed approaches are at least two times faster than that of the approach that uses only CCFinder. Therefore, we conclude that our propose approaches are able to detect code clones faster than the approach that uses only CCFinder.

4.2.2 Comparison of Our Proposed Approaches

To answer RQ2, this study compares detection time between the proposed approaches and then examines results such as the number of equivalence classes and code clones. In terms of the detection time, the approach with non-normalization is relatively faster than other proposed approaches from Tables 2, 3, and 4.

Tables 5, 6, and 7 shows the number of instances from *Apache Ant*, *Linux kernel*, and *Samsung Galaxy* respectively. In these tables, the number of equivalence classes is shown in the column #Equivalence classes. The column #Files in equivalence classes represents the number of files that are contained in the non-singleton equivalence classes. Meanwhile, the column #Files in singletons represents the number of files contained in the singleton equivalence classes. The column #Clone sets and #Clone clones represents the number of detected clone sets and code clones respectively. In the "Approach with non-normalization", these columns show the number of detected clone sets and code clones by CCFinder. Meanwhile, other approaches describe the number of clone sets and code clones except for clone sets and code clones within the identical files in the same equivalence class.

We found similar tendencies from these tables. In the *Apache Ant*, the least number of equivalence classes, files in singletons, and code clones are detected by "INSC Approach" as shown in Table 5. Similarly, in the *Linux kernel* and *Samsung Galaxy*, the least number of equivalence classes, files in singletons, and code clones are detected by "ISC Approach" and "INSC Approach" as shown in Tables 6, and 7.

4.3 Discussion

This section discusses the results of the case study described in Sects. 4.2.1 and 4.2.2. As mentioned in Sect. 4.2.1, the proposed approaches detect code clones faster than the "Approach that uses only CCFinder". This was caused by the large decrease of the number of unique files. 5.13-21.45%, 9.87-10.77%, and 0.19- 0.22% of the number of unique files

Table 5 Results of Apache Ant.

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singletons	#Clone sets	#Clone clones
Approach that uses only CCFinder	-	-	-	15,626	246,245
Approach with non-normalization	4,174	14,696 (78.55%)	4,012 (21.45%)	14,778	243,211
IEC Approach	3,119	17,652 (94.36%)	1,056 (5.64%)	13,127	234,006
ISC Approach	2,993	17,739 (94.82%)	969 (5.18%)	13,003	233,011
INSC Approach	2,973	17,749 (94.87%)	959 (5.13%)	12,976	232,812

Table 6 Results of Linux Kernel.

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singletons	#Clone sets	#Clone clones
Approach that uses only CCFinder	-	-	-	23,031	306,592
Approach with non-normalization	1,516	6,995 (89.23%)	844 (10.77%)	20,356	293,076
IEC Approach	1,513	7,002 (89.32%)	837 (10.68%)	20,346	293,013
IEM Approach	1,517	7,046 (89.88%)	793 (10.12%)	20,248	292,198
IEMC Approach	1,512	7,056 (90.01%)	783 (9.99%)	20,228	292,026
ISC Approach	1,494	7,065 (90.13%)	774 (9.87%)	20,196	291,766
INSC Approach	1,494	7,065 (90.13%)	774 (9.87%)	20,196	291,766

Table 7 Results of Samsung Galaxy.

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singletons	#Clone sets	#Clone clones
Approach that uses only CCFinder	-	-	-	274,186	2,529,843
Approach with non-normalization	14,737	29,508 (99.78%)	65 (0.22%)	113,929	2,208,830
IEC Approach	14,735	29,518 (99.81%)	55 (0.19%)	113876	2,208,713
IEM Approach	14,640	29,516 (99.81%)	57 (0.19%)	113853	2,208,619
IEMC Approach	14,611	29,518 (99.81%)	55 (0.19%)	113897	2,208,701
ISC Approach	14,576	29,518 (99.81%)	55 (0.19%)	113797	2,208,363
INSC Approach	14,576	29,518 (99.81%)	55 (0.19%)	113797	2,208,363

are decreased in *Apache Ant*, *Linux kernel* and *Samsung Galaxy*, respectively.

Among the proposed approaches, the "Approach with non-normalization" is the fastest in the case of *Linux kernel* and *Samsung Galaxy*. Meanwhile, in the case of *Apache Ant*, "ISC Approach" and "INSC Approach" are faster than other proposed approaches. However, the time difference between the "Approach with non-normalization" is still very small (39 seconds at maximun). This is because in the case of *Linux kernel* and *Samsung Galaxy*, the number of unique files are almost same between the approaches. This leads that the "Approach with non-normalization" that needs the least processes and post-process compared with the other approaches is the fastest. Meanwhile, in the case of *Apache Ant*, "ISC Approach" and "INSC Approach" output the least number of files in singletons. This leads the "ISC Approach" and "INSC Approach" detect code clones faster than other proposed approaches.

Therefore, It is expected that if the files contain many unique files, then "ISC Approach" and "INSC Approach" are the fastest, however, in other cases, the "Approach with non-normalization" is the fastest among the approaches.

5. Threats to Validity

We identified the following threats to the validity of this study. Our proposed approaches rely on the quality of the underlying clone detection tool and hash function to detect code clones. We countered this threat by a careful selection of clone detection tool and hash function. We settled on using CCFinder, which is recognized as a widely-used clone detection tool that is having high accuracy to detect code clones and MD5 hash function is very unlikely to collide.

As case study, we picked three different sizes of OSS systems from diverse domains to achieve generalities of results. However, our results of case study might be different in the other software systems. To alleviate this limitation, we plan to apply our proposed approaches to additional software systems to achieve generality of results of case study.

6. Related Work

Many studies have been proposed for detecting code clones. Baker proposed an approach for detecting code clones then developed a tool named Dup. It detects Type-1 (see Sect. 2.1) and Type-2 code clones based on the similarities of token sequences. CP-Miner detects Type-1, Type-2, and Type-3 code clones based on the similarities of token sequences.. To detect code code clones, it adopted frequent subsequent mining which is an association analysis approach to discover frequent subsequences in a collection of sequences [9].

Several approaches using CCFinder have been proposed. Sasaki et al. proposed an approach that detects identical files and then investigate the characteristics of them [18]. This study identifies identical files without any (or just slight) modifications in comments or headers using MD5 hash values of the tokenized files. Our proposed approaches are similar to this study. We also compute MD5 hash values of the input source files for partitioning equivalence classes. The essential difference between their work and our study is that the goal of their study is to investigating the identi-

cal files but our this study is focus on the proposing the approaches with preprocessing input source files with different degrees of normalizations and then compare them to find out the faster one. A tool named D-CCFinder implemented by Livieri et al. detects code clones at distributed environment based on CCFinder [17]. D-CCFinder partitions the clone search for very large systems into smaller pieces to be distributed to detect code clones with high speed. This goal of study is the same between their study and this study, but our approaches detect code clones on the single PC, whereas D-CCFinder uses distributed approach. To achieve the fast detection time, we consider extending our proposed approach by distributed approach.

Some studies that use MD5 hash function to detect code clones have been proposed. Hummel et al. proposed index-based code clone detection approach [19]. It detects Type-1 and Type-2 clones using MD5 hash function to calculate hash values from normalized statements. After a hash value is computed for each input source file, code clones are retrieved from the databases where the hash values are stored. Koschke proposed code clone detection approach using suffix tree and MD5 hash function [20]. The goal of his research is to detect code clones between a subject systems and a set of other systems for finding potential license violations. This study generates suffix tree for smaller system between a subject systems and a set of other systems and then compares every file of corpus with the generated suffix tree to detect Type-1 and Type-2 code clones.

7. Conclusion and Future Work

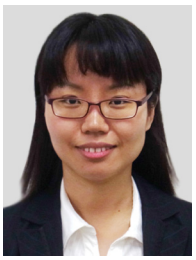
In this paper, we proposed code clone detection approaches with preprocessing input source files using different degrees of normalizations to investigate how the normalizations impact the code clone detection. The proposed approaches perform equivalence class partitioning to the input source files based on the MD5 hash values in the preprocessing. After preprocessing, code clones are only detected from a set of files that are selected from each equivalence class. To detect code clones, this study uses CCFinder which is a token-based code clone detection tool. These approaches can be categorized into two types, approaches with non-normalization and normalization. The former is the detection of code clones based on identical files without any normalization. Meanwhile, the latter category is the detection of code clones based on identical files with different degrees of normalizations such as removal of all lines containing macros.

In case study, we applied the proposed approaches as well as the approach that uses only CCFinder to different versions of three OSS systems and evaluated them with respect to the code clone detection time. For the case study, we found out that our proposed approaches detect code clones faster than the approach that uses only CCFinder. We also discover the approach with non-normalization is the fastest among the proposed approaches in many cases.

As future work, we plan to apply the proposed approaches to various size of software systems in different domains to achieve more accurate results. In addition to that, we consider introducing other code clones detection tools and hash function to extend to this study. Finally, we plan to extend the proposed approaches by adopting the distributed approach to achieve execute time with high speed.

References

[1] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems," J. Syst. Softw., vol.83, no.1, pp.67–76, 2010.
[2] D. Faust and C. Verhoef, "Software product line migration and deployment," Software Practice and Experience, vol.33, no.10, pp.933–955, 2003.
[3] P. Runeson, C. Andersson, and M. Höst, "Test processes in software product evolution: A qualitative survey on the state of practice," J. Softw. Maint. Evol.: Res. Pract., vol.15, no.1, pp.41–59, 2003.
[4] C.K. Roy, J.R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Sci. Comput. Program., vol.74, no.7, pp.470–495, 2009.
[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Trans. Softw. Eng., vol.33, no.9, pp.577–591, 2007.
[6] B.S. Baker, "On finding duplication and near-duplication in large software systems," Proc. of WCRE, pp.86–95, 1995.
[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., vol.28, no.7, pp.654–670, 2002.
[8] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching: Research articles," J. Softw. Maint. Evol.: Res. Pract., vol.18, no.1, pp.37–58, 2006.
[9] 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., vol.32, no.3, pp.176–192, 2006.
[10] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A.E. Hassan, "An empirical study on inconsistent changes to code clones at release level," Proc. of WCRE, pp.85–94, 2009.
[11] T.T. Nguyen, H.A. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone-aware configuration management," Proc. of ASE, pp.123–134, 2009.
[12] J. Li and M.D. Ernst, "CBCD: Cloned buggy code detector," Proc. of ICSE, pp.310–320, 2012.
[13] L. Barbour, F. Khomh, and Y. Zou, "An empirical study of faults in late propagation clone genealogies," J. Softw. Evol. and Proc., vol.25, no.11, pp.1139–1165, 2013.
[14] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," Proc. of ESEC/FSE, pp.187–196, 2005.
[15] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," Proc. of ASE, pp.231–240, 2006.
[16] D. Gusfield, Algorithms on strings, trees, and sequences: Computer science and computational biology, Cambridge University Press, 1997.
[17] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," Proc. of ICSE, pp.106–115, 2007.
[18] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in FreeBSD ports collection," Proc. of MSR, pp.102–105, 2010.
[19] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: Incremental, distributed, scalable," Proc. of ICSM, pp.1–9, 2010.
[20] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," J. Softw. Evol. and Proc., vol.26, no.8, pp.747–769, 2014.



Eunjong Choi received her Master from Osaka University in 2011. She is a Ph.D. candidate at Osaka University since 2012. Her research interests include code clone analysis and refactoring detection. She is a member of the ACM and IPSJ.

Norihiro Yoshida received his B.E. from the Kyushu Institute of Technology in 2004 and his Master and Ph.D. from Osaka University in 2006 and 2009, respectively. He is an associate professor at Nagoya University. Before joining Nagoya University in 2014, he was an assistant professor at the Nara Institute of Science and Technology from 2010. His research interests include program analysis and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Yoshiki Higo received his master's degree and Ph.D. degree in information science and technology from Osaka University in 2004 and 2006, respectively. At present he is an assistant professor at Osaka University. His research interests include code clone analysis, mining software repositories, software metrics, and refactoring support techniques. He is a member of the IEEE, IPSJ, IEICE, and JSSST.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984-1986. He was a research associate at Osaka University from 1984-1989, an assistant professor from 1989-1995, and is a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.