Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the Eighth International Workshop on Software Clones (IWSC 2014)

How Accurate Is Coarse-grained Clone Detection?: Comparision with Fine-grained Detectors

Keisuke Hotta, Jiachen Yang, Yoshiki Higo, and Shinji Kusumoto

18 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

How Accurate Is Coarse-grained Clone Detection?: Comparision with Fine-grained Detectors

Keisuke Hotta[1], Jiachen Yang[2], Yoshiki Higo[3], and Shinji Kusumoto[4]

[1] k-hotta@ist.osaka-u.ac.jp, [2] jc-yang@ist.osaka-u.ac.jp
[3] higo@ist.osaka-u.ac.jp, [4] kusumoto@ist.osaka-u.ac.jp
Graduate School of Information Science and Technology
Osaka University, Japan

Abstract: Research on clone detection has been quite successful over the past two decades, which produced a number of state-of-the-art clone detectors. However, it has been still challenging to detect clones, even with such successful detectors, across multiple projects or on thousands of revisions of code in limited time. A simple and coarse-grained detector will be an alternative of detectors using fine-grained analysis. It will drastically reduce time required for detection although it may miss some of clones that fine-grained detectors can detect. Hence, it should be adequate for a tentative analysis of clones if it has an acceptable accuracy. However, it is not clear how accurate such a coarse-grained approach is. This paper evaluates the accuracy of a coarse-grained clone detector compared with some fine-grained clone detectors. Our experiment provides an empirical evidence about acceptable accuracy of such a coarse-grained approach. Thus, we conclude that coarse-grained detection is adequate to make a summary of clone analysis and to be a starter of detailed analysis including manual inspections and bug detection.

Keywords: Clone detection, Software evolution, Mining software repositories

1 Introdution

Code clone, also called duplicated code, is known as a typical bad smell for software development and maintenance [FBB+99]. They recieve a considerable amount of attention from developers [YM13], which may be because code cloning is quite easy to do and almost unavoidable [KBLN04].

Such a background encourages researchers to develop a variety of techniques to cope with clone-related problems [Kos08, ZR12, Kos06]. Clone detection is one of the hottest topics in this research area since it plays a fundamental role for managing clones. Research on clone detection has been quite successful because of great effort that many researchers have spent. As a result, a variety of clone detectors have been developed and used [RBS13, RC07, Bak95, BYLB98, KKI02, MLM96, DRD99, HJHC10, RC08, JMSG07].

Nowadays, the success of clone detection research has let clone detection go well beyond a single state of code in a single project. It opened its own application area beyond both project and historical borders. More concretely, it has been applied not only in intra-project but also in inter-project, and it has been used to analyze clone evolution.

Inter-project clone detection opens our eyes to find reusable code, and to detect plagiarism and license violations [Kos13]. Historical analysis of clones has provided a number of useful findings [KSNM05, GK11, Kri08, HSHK10, LW08, HG12, JDHW09] because historical data has rich information that the current states of projects do not have [HMK12, KMM+10].

However, these research areas suffer the following issues.

Time required for detection: although state-of-the-art detectors have been successful to detect clones in single projects, clone detection across multiple projects or on multiple revisions requires a huge amount of time.

A huge number of clones: state-of-the-art detectors tend to report a number of clones even from a single state of code in a single software system. If we apply them to inter-project detection or historical analysis, the number of detected clones will become much larger. This may make it difficult to analyze the results in a limited time period.

It should be an alternative of using state-of-the-art detectors to use simple and coarse-grained clone detectors [HHK13]. Herein, "coarse-grained clone detection" means "*clone detection on coarser units of code*". In this paper we suppose that finer unit of code means sequence of tokens, statements, or lines, and coarser units of code include blocks, methods, functions, classes, and files. In our defenition, fine-grianed detectors can find clones composed of a part of syntactic block, but coarse-grained detectors cannot find them. The main objective of this paper is to reveal the performance and the accuracy of such a coarse-grained detection. We have implemented a simple block-based detector and use it in our investigation.

The most advantageous point of the coarse-grained approach should be in its performance. That is, it will drastically reduce time required for clone detection because it requires a fewer number of comparisons of code units. Hence, it should be suitable for analyzing inter-project cloning or clone evolution. Therefore, we investigate the performance of the coarse-grained detection as our first research question.

RQ1: Is the coarse-grained detector much faster than fine-grained ones?

In addition to that, coarse-grained detectors will report a fewer number of clones than fine-grained ones. Therefore, they may make it possible to analyze the results in limited time. Hence, we are interested in the number of detected clones in the second research question.

RQ2: Does the coarse-grained detector report a fewer number of clones?

Note that this research question considers only the number of detected clones. In other words, this research question is not interested in the correctness of detected clones. Hence, it is unclear how many correct clones are detected and missed by the coarse-grained detector. The remaining research questions focus on the accuracy of the coarse-grained detector.

The next research question is interested in the accuracy of the detection. We think the coarse-grained detection has high precision. This is because if there exists a pair of code fragments that is regarded as a clone pair with such a coarse-grained technique, the pair also will be regarded as a clone pair with a fine-grained approach. On the other hand, it might have low recall compared to fine-grained detectors. This is because it will report a fewer number of clones than fine-grained techniques on the same target software systems. This means that the coarse-grained approach misses some clones that fine-grained techniques can detect.

RQ3: Does the coarse-grained detector have high precision?

RQ4: Does the coarse-grained detector have high recall?

Our experiment confirmed high scalability of the coarse-grained detection on multiple projects and multiple revisions compared with fine-grained ones. It also showed that the coarse-grained detector reported fewer numbers of clones than fine-grained detectors. In addition, the coarse-grained detector achieved high precision compared to other detectors. On the other hand, it had lower recall than other detectors, but it was not so low.

Based on these findings, we conclude that using such a coarse-grained clone detector as a starter of clone analysis should be an effective way to analyze across multipe projects or multiple revisions. For instance, suppose that we are about to analyze clones among thousands of projects, which requires a vast amount of time to be analyzed with fine-grained detectors. In this case, we can apply a coarse-grained detector to get an overview of clones in the dataset even though there exists a tight limit of time. With the coarse-grained detector, you may find that there are a number of clones in a particular pair of projects. However, we have to note that the coarse-grained detector has a little lower recall compared to fine-grained ones. Hence, it is not sufficient using only coarse-grained detectors because it should miss some clones. Therefore, we suggest a two-staged analysis using both of fine-grained and coarse-grained detectors. That is, at the first stage we use a coarse-grained detector to get a summary of clones and narrow down the analysis target with the results. After the first stage, we use a fine-grained detector to conduct more detailed analysis with the narrowed target. Such a two-staged analysis should be effective for a huge dataset with limited time.

The remainder of this paper is organized as follows. At first, we discuss related work in Section 2. Section 3 describes the coarse-grained detector used in this study. Section 4 gives how to conduct the experiment, and Section 5 shows the experimental results. Section 6 discusses threats to validity of this research, and the final section, Section 7, concludes this paper.

2 Related Work

2.1 Clone Detection across Multiple Projects

Koschke developed a technique to detect inter-system clones with a suffix-tree-based approach [Kos13]. His technique is interested in clones between a subject system and a set of other systems. He achieved high scalability with some heuristics, including to generate suffix trees for either the subject system or the set of other systems, and to use a filter based on hashing.

Ossher et al. proposed a technique to detect file-level clones [OSL11]. Through an empirical study on a dataset that includes over 13,000 projects [LBJP], they revealed that approximately 10% of files were clones. Furthermore, they found that file-level cloning tends to occur in cases of reusing whole of existing projects as a template of a new project.

A similar work was conducted by Sasaki et al. [SYHI10]. They developed a tool named FCFinder to detect file-level clones. They applied FCFinder to FreeBSD Ports Collection, and they reported that 68% of files were file clones.

Ishihara et al. proposed a technique to detect method-level clones for locating library candidates [IHH+12]. They conducted an empirical study on the dataset used in Ossher et al.'s study, and their technique found approximately 2,900,000 cloned methods within four hours.

Keivanloo et al. conducted a set of empirical studies toward building real-time clone search system [KRC11]. They adopted multi level hashing to detect clones in their study. Although they were interested in clone search, they also reported the performance of their light weight clone detection approach on a code base that includes 1,500,000 Java classes. The empirical study revealed that their approach could detect approximately 11 billion clone pairs in 21 minutes.

Keivanloo et al. proposed a framework to improve scalability of existing detectors for large datasets including multiple projects [KRRC12]. Their technique is compirising of shuffling, repetition, and random subset generation of the subject dataset. Their approach is completely independent of detectors, and so it can be applied for any detectors without any modifications on them. The research was followed by a further experiment by Svajlenko et al. on six existing clone detectors [SKR13], and the experiment revealed the effectiveness of the framework.

2.2 Analysis of Clone Evolution

The pioneers of studies on clone evolution are Kim and her colleagues [KSNM05]. They formulated clone genealogies and conducted an empirical study on clone genealogies. The empirical study revealed that most of clones were short-lived.

Göde and Koschke investigated how many times clones were changed during their evolution [GK11]. They showed that most of clones were changed at most once, and changes on them did not cause severe problems in most cases.

There exists some other studies to analyze evolution of clones, which includes analyzing evolution of Type-1 clones [GÖ09], analyzing evolution of near-miss clones [Baz12, SRS11], and analyzing stability or changeability of clones [Kri08, HSHK10, LW08, HG12]. The results did not agree with each other, and so the research community still has a room for discussion about harmfulness of clones. It can be said that, however, there are both harmful clones and harmless ones. Hence, *managing clones is no longer considered a "hunt-and-kill" game* [WG12] and *we thus have to carefully select the clones to be managed to avoid unnecessary effort managing clones with no risk potential* [GK11].

2.3 Comparison between Clone Detectors

One of the benchmarks on clone detectors is the one conducted by Bellon and his colleagues [BKA+07]. It is quite difficult to make a correct set of clones due to the vagueness of definition of clones. Hence, they made their correct set through a manual inspection. They collected clones with six detectors on eight open source projects, and looked them through to judge whether they should be really regarded as clones. The vast amount of collected clones made it impossible to investigate all of them, and so they randomly selected clones to be judged.

Roy et al. provided another benchmark from a different standpoint [RCK09]. That is, they compared clone detectors with four distinct senarios on copy/paste/modify operations. Their benchmark showed which tools are adequate for each senario.

Roy and Cordy also provided another benchmark for evaluating clone detectors empirically

[RC09]. The key idea of their study is using mutation. Their technique generates and injects mutants of code fragments, and evaluates detectors with them.

A similar work to our study was conducted by Ducasse et al. [DNR06]. They implemented a clone detector using simple string matching, which can be applied to a number of different languages including COBOL, Java, or C++. They confirmed that the inexpensive detector achieved high recall and acceptable precision. A major difference between their work and this study is in the granularity of clone detection. Their work used a simple string matching to reduce time required for detection, which is a fine-grained detection technique. On the other hand, this study is interested in coarse-grained technique. The simple string matching reduces the cost of each comparison between two elements, but coarse-grained detection, by contrast, reduces not only the cost of each comparison but also the number of comparisons of elements. Hence, coarse-grained technique will be faster than simple fine-grained ones, but it may miss some clones that finer techniques can detect. The objective of this research is to empirically reveal that.

3 Coarse-grained Clone Detection

This section describes the coarse-grained detector used in this study. It detects block-level clones from the given source files. It explains how to detect clones in a set of source files, followed by the description of incremental detection.

3.1 Clone Detection on a Set of Source Files

Input and Output

The detector takes a set of source files as its input, and reports a list of clone pairs among them. Currently, it targets only Java because of the limitations of the implementation.

Procedure

The detection procedure consists of the following four steps.

- **STEP1:** Parse given source files to detect blocks.
- **STEP2:** Normalize every detected block.
- **STEP3:** Calculate a hash value from each block.
- **STEP4:** Group the blocks based on their hash values.

Figure 1 shows an overview of the procedure. The followings describe each step in detail.

STEP1: Detect Blocks

The first step is to detect all the blocks from the given source files. Herein, blocks include classes, methods, and block statements such as `if` or `for` statements. This step requires not only lexical analysis but also syntax analysis. Our implementation uses Java Development Tool (JDT) to perform the syntax analysis.

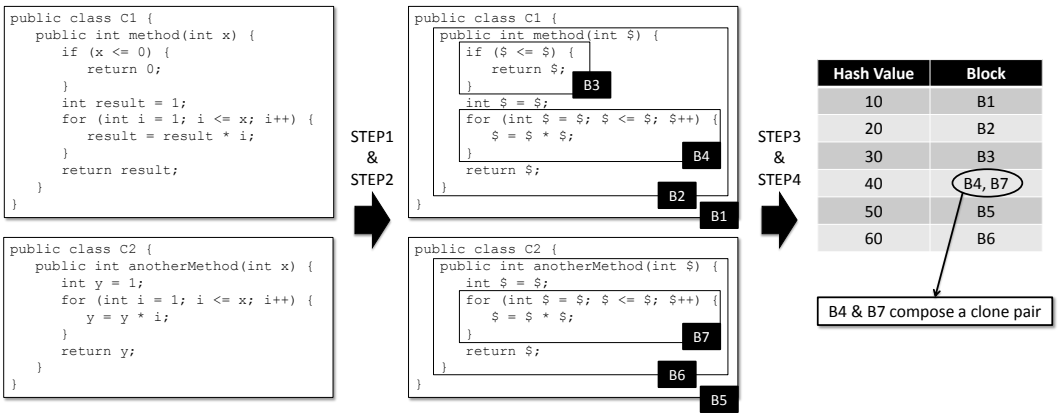

Figure 1: Overview of the Detection Procedure

STEP2: Nomalize Blocks

The next step is normalization for every block detected in the previous step. At first, it reformats every block with a regularized form. This procedure allows the detector to ignore differences of white-spaces or tabs. In addition to that, it replaces each variable name and each literal with a special token, which allows the detector to find Type-2 clones. Hence, the detector can find both of Type-1 and Type-2 clones, but cannot find Type-3 clones.

STEP3: Calculate Hash Values

The third step calculates hash values from texts of respective blocks. Our implementation uses `hashCode()` of `java.lang.String` as the hash function. Any other hash functions, however, can be used instead if they can generate a numerical value from a given string.

STEP4: Group Blocks

The final step is grouping blocks based on their hash values. Two blocks have the same hash value if their text representations after normalized are equal to each other. Hence, a block pair is regarded as a clone pair if the two blocks have the same hash value. The detector groups all the detected blocks and reports all the clone pairs.

3.2 Incremental Detection

In addition to the function of clone detection for a single state of code, the detector has a function for detecting clones incrementally. The detector takes a Subversion repository as its input for performing incremental clone detection, and reports clone pairs in every revision.

The detector is motivated by the idea of the implementation from the literature [HHK13]. That is, it analyzes only files that were modified in a target commit. In other words, it reuses the results of analysis on the past commits of the target commit.

Suppose that the detector is about to analyze a commit c_r, which is a commit between the revisions $r-1$ and r. The commit c_r modified a source file f_k, and the revision r has n source

files ($f_1 \ldots f_n$). In this case, the detector has already finished analyzing all the commits before c_r. Hence, all the analysis results for files $f_1 \ldots f_n$ except for f_k can be used for analyzing c_r because these files did not changed in c_r. Therefore, the detector newly performs STEP1, STEP2, and STEP3 only for f_k, and it performs STEP4 on all the blocks in files $f_1 \ldots f_n$.

4 Experimental Setup

4.1 Terms

This study uses the benchmark of Bellon and his colleagues [BKA+07], which is a well-used benchmark in the research community.

Hereafter, this paper uses the following terms.

Reference: a reference is a clone pair that was judged as correct by Bellon and his colleagues.

Candidate: a candidate is a clone pair that was reported by a clone detector.

This study uses a metric named *OK-value* to judge whether a candidate matches a reference. This metric was defined and used in the original study of Bellon et al..

Before defining *OK-value*, we give the definition of $contained(CF1, CF2)$ for a given code fragments $CF1$ and $CF2$ as follows.

$$contained(CF1, CF2) = \frac{|lines(CF1) \cap lines(CF2)|}{|lines(CF1)|} \tag{1}$$

where, $lines(CF)$ indicates the set of all the lines of code in the code fragment CF.

Now we can define *OK-value* for a given pair of clone pairs $CP1$ and $CP2$.

$$\begin{aligned} OK(CP1, CP2) = min(&max(contained(CP1.CF1, CP2.CF1), \\ &contained(CP2.CF1, CP1.CF1)), \\ &max(contained(CP1.CF2, CP2.CF2), \\ &contained(CP2.CF2, CP1.CF1))) \end{aligned} \tag{2}$$

where, $CP.CF1$ and $CP.CF2$ are the two code fragments of which the clone pair CP consists.

It is regarded that a detector could detect a reference CR if it reported a candidate CC that satisfies the following condition.

$$OK(CR, CC) >= threshold \tag{3}$$

This study uses 0.7 as the *threshold*, which is the same value used in the original work.

Furthermore, this paper uses the following metrics to evaluate the accuracy of detected clones for a given target software system P and a given detector T.

$$Precision(P, T) = \frac{|DetectedRefs(P, T)|}{|Cands(P, T)|} \tag{4}$$

$$Recall(P, T) = \frac{|DetectedRefs(P, T)|}{|Refs(P)|} \tag{5}$$

where,

- *DetectedRefs*(P,T) refers to a set of references in P that were detected with T.
- *Cands*(P,T) refers to a set of candidates that T reported from P.
- *Refs*(P) refers to the set of all the references in P, which is independent of detectors.

We have to note that there exists another metric used in the original benchmark of Bellon and his colleagues, which is named *Good* value. *Good* value is a stronger criterion than *OK* value. *OK* value becomes high in the case that a clone reference(candidate) subsumes a clone candidate(reference) sufficiently. On the other hand, *Good* value becomes high in the case that a clone reference(candidate) sufficiently match a clone candidate(reference). In other words, *Good* value becomes not high in the case that *OK* value becomes high if a large clone reference(candidate) subsumes a small clone candidate(reference).

The reason why we have chosen the looser metric is that we suppose that the main use of the coarse-grained detection is to roughly analyze code clones on a huge data set. Hence, we think that *Good* metric is too strict for the purpose.

4.2 Target Software Systems

The expriment for RQ2, RQ3, and RQ4 targets software systems used in Bellon et al.'s work because there does not exist Bellon et al.'s correct sets of clones on any other software systems. In addition, it omits C projects from the target due to the limitation of our implementation. Therefore, there are four target systems for the expriment, which are shown in Table 1.

The experiment for RQ1 needs another set of targets. This is because the main usage of such a coarse-grained detector should be reducing time required for detection. Hence, it should be reasonable to use a corpus of many projects and historical repositories to answer RQ1.

The experiment uses a dataset called UCI dataset [OSL11, LBJP]. The dataset has approximately 13,000 Java projects, which should be enough large for fine-grained detectors not to complete their tasks. More statistics are shown in Table 2[1].

[1] We ignored branches, tags, test cases, and not parsable files, which results different values from ones in [OSL11, IHH+12]

Table 1: Target Software Systems (for RQ2, RQ3, and RQ4)

Name	Shortened	LOC	# of References
eclipse-ant	ant	34,744	30
eclipse-jdtcore	jdtcore	147,634	1,345
j2sdk1.4.0-javax-swing	swing	204,037	777
netbeans-javadoc	netbeans	14,360	55

Table 2: Overview of UCI Dataset

# of .java files	2,092,739
# of projects	13,193
total LOC of .java files	373,500,402

The experiment also uses the historical code repository of DNSJava. We selected this repository because it has been used in some empirical studies on clone evolution and it has a reasonable length of histories. Table 3 tells some statistics of the repository.

4.3 Detectors Used for the Comparison

Our experiment uses seven detectors shown in Table 4 as comparison targets. We chose four of them, Dup, CloneDR, CCFinder, and Duploc, because they were used in the original work of Bellon et al. and they can handle Java. We omitted CLAN from the list of attendees in this experiment becanse it detects clones based on function metrics, which is categorized a coarse-grained detector in our definition. We add other two detectors, Deckard and CDSW, that were developed recently because the four detectors used in the original work were developed over a decade ago.

The other one, which is refered as LD (*Lightweight Detector*), was developed by ourselves based on the paper of Hummel et al. [HJHC10]. It aims to detect clones in a short time period with lightweight analysis. The technique creates and uses indexes on N-grams of source lines, which gives it high scalability. Although it uses a lightweight analysis, it is more fine-grained than the coarse-grained detector. Hence, we think that coarse-grained analysis will be faster than it, but it is still unclear. This is a motivation that we use the detector in the experiment.

In addition to that, it has a function of incremental clone detection. Incremental clone detection is one of the way to cope with the issue of time required for clone detection on multiple revisions of code [GK09]. Hence, this detector has high scalability for detecting clones on multiple revisions. This is the other reason that we use this detector as our target.

Table 3: Overview of the Repository of DNSJava

the revision number of the latest revision	1,670
# of commits where at least one .java file was modified	1,432
# of files in the latest revision	7,362
total LOC of .java files in the latest revision	1,237,336

Table 4: Detectors Used as Comparison Targets

Tool	Taxonomy
Dup [Bak95]	Token
CloneDR [BYLB98]	AST
CCFinder [KKI02]	Token
Duploc [DRD99]	Text
Deckard [JMSG07]	AST
CDSW [MHH+13]	Other(Statement)
LD (based on [HJHC10])	Text

5 Results

This section provides the results of our experiment. It answers the four research questions, and gives a summary of the results.

Note that we reused the results of Bellon et al.'s experiment of Dup, CloneDR, CCFinder, CLAN, and Duploc, and so we did not run the five detectors on our own platform.

5.1 RQ1: Is the coarse-grained detector much faster than fine-grained ones?

The answer is **Yes**.

We applied the coarse-grained detector and LD to UCI dataset and the repository of DNSJava to compare the performance of the coarse-grained approach with fine-grained one. Note that both of the two detectors can perform incremental clone detection and parallelized execution.

Each detector ran on an HP workstation Z820 with two Intel 64bit 2.4GHz CPUs and 128GB RAM. The workstation has an SSD that contains the whole of UCI dataset and a copy of the repository of DNSJava. Each detector used 32 threads to perform parallelized execution.

Table 5 shows the result of the comparison. Coarse-grained in the table refers the coarse-grained detector described above, which detects clones in the block-level. Note that LD could not finish its task on UCI dataset within 24 hours, so we aborted it. As the table shows, the coarse-grained detector was faster than fine-grained one. We also applied CCFinder on the latest revision of the repository of DNSJava just for reference. The target has approximately 7,000 files with 1.2 million LOC as shown in Table 3. As a result, CCFinder took 42 minutes for the target. Therefore, it should not be realistic to apply it on the whole of UCI dataset or all the revisions of DNSJava due to the massive amount of time required for the detection.

To summarize the comparison of performance of the two detectors, although LD completed its task in hours, the results show the high scalability of the coarse-grained technique.

5.2 RQ2: Does the coarse-grained detector report a fewer number of clones?

The answer is **Yes**.

Table 6 shows the number of detected clone pairs with each detector on each target. Note that Duploc could not finish its task on swing, and so the corresponding column is filled with N/A.

As shown in the table, the coarse-grained detector reported fewer numbers of clones in most cases, except for the comparison with CloneDR and CDSW. Although the coarse-grained detector did not report the fewest number of clones, we can say that it found fewer numbers of clones.

Table 5: Time Elapsed for Detection

	LD	Coarse-grained
UCI dataset (multiple projects)	N/A	152 [m]
DNSJava (multiple revisions)	212 [m]	17 [m]

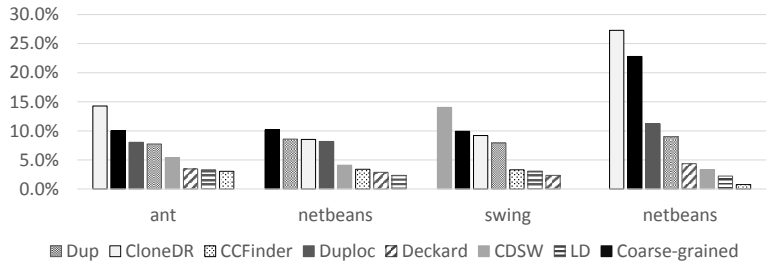

Figure 2: The Results of Comparision (Precision)

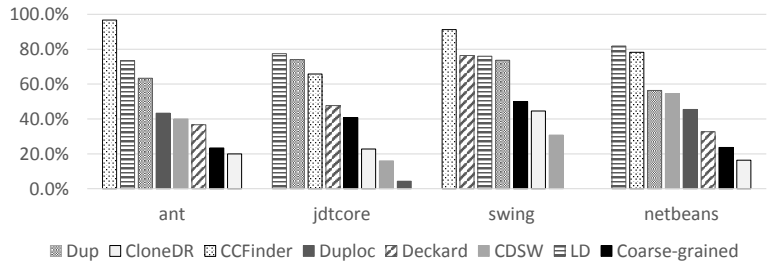

Figure 3: The Results of Comparison (Recall)

5.3 RQ3: Does the coarse-grained detector have high precision?

The answer is **Yes**.

Figure 2 shows the results of comparison of *precision*. Each bar in the graph indicates the value of *precision* with a detector on a target project. The bars are sorted in the descending order of the value of *precision* for each of the target projects. The black bar indicates the coarse-grained detector.

The graph tells us that the coarse-grained detector achieved the highest *precision* in the case of jdt-core. In the other cases, it have the second highest *precision*.

Therefore, we conclude that the coarse-grained detector achieved high precision compared with others.

Table 6: The Number of Detected Clone Pairs

Tool	ant	jdtcore	swing	netbeans
Dup	245	11,589	7,220	344
CloneDR	42	3,593	3,766	33
CCFinder	950	26,049	21,421	5,552
Duploc	162	710	N/A	223
Deckard	319	22,353	25,415	414
CDSW	222	5,247	1,703	1,344
LD	662	44,294	19,253	1,360
Coarse-grained	70	5,398	3,922	57

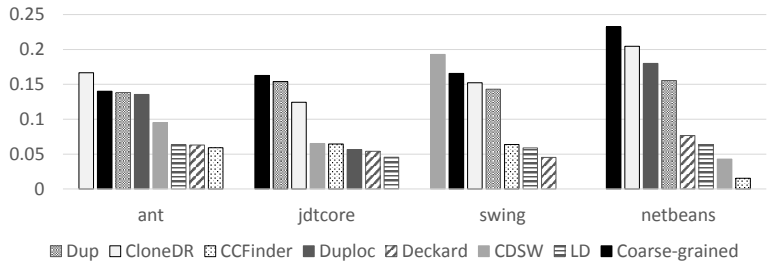

Figure 4: The Results of Comparision (F-Measure)

5.4 RQ4: Does the coarse-grained detector have high recall?

The answer is **No**.

Figure 3 shows the values of *recall*. This graph has the same fashion of the graph shown in Figure 2 except for the difference of represented values.

We can see that the coarse-grained detector could not achieve the highest *recall* in all the targets. Hence, we conclude that the coarse-grained detector cannot achieve high recall.

However, please note that the coarse-grained detector did not have the lowest *recall* in all the cases. Therefore, it could detect a ceartain amount of, not so few, clones.

5.5 Summary of the Results

Our experimental result confirmed the high scalability of a coarse-grained technique on inter-project and multi-revision clone detection. Furthremore, it showed that the coarse-grained detection technique achieved high precision with fewer numbers of clones detected, but it could not achieve high recall.

In general, the values of *precision* and *recall* are in trade-off. The clone detector that have high recall tend to have low precision and vice verse. To evaluate the total accuracy, we compared the values of *F-Measure*, which is the harmonic average of *precision* and *recall*. A high *F-Measure* means that the values of *precision* and *recall* are highly balanced. The formal definition of *F-Measure* is shown below.

$$F-Measure(P,T) = \frac{2 * Precision(P,T) * Recall(P,T)}{Precision(P,T) + Recall(P,T)} \tag{6}$$

Figure 4 shows the values of *F-Measure* in the similar fashion of Figures 2 and 3. This graph shows that the coarse-grained detector achieved the highest *F-Measure* on two out of four target projects. This result indicates that it has highly balanced *precision* and *recall*, and it is no longer a poor detector.

Based on the findings, we recommend it as a choice to use such a coarse-grained detection to get an overview of clones. It has been claimed that a large software system has a considerable number of clones, and it is difficult and not effective to look through all of them. Therefore, we think that using such a coarse-grained technique at first must be helpful to analyze clones effectively. After that, it is necessary to use fine-grained state-of-the-art detectors to get more detailed information of clones. This is because the coarse-grained approach tends to have less recall than fine-grained ones. Using a fine-grained detector will cover the disadvantage of the

coarse-grained one because it can retrieve clones that the coarse-grained one misses. We will need to narrow down our investigation target due to limitations of resources to use a fine-grained detector. In this case, a coarse-grained technique will help us narrow down the investigation target.

6 Threats to Validity

Configurations of Detectors

This study used a particular configuration for each detector, but the configurations have not small effect on the results of detection. There exists an automated way to find a adequate configuration for a given situation with the search based approach [WHJK13]. If we search a better configuration for each detector with such a technique, the results might be different from this study.

Different Core Paradigms of Detectors

This study compared the accuracy of a coarse-grained clone detector with other detectors. However, the detectors adopt different core paradigms and heuristics. Hence, such differences except for granularity might affect the results of comparison. It is impossible, however, to compare the detectors with the same paradigm because the detection granularity plays an important role on the paradigm of clone detection.

Hash Collision

We are threaten by the possibility of hash collision as long as we use hash values for comparing two elements of code. For the ease of implementation, we adopted a simple hash function provided by Java standard libraries. However, Keivanloo et al. revealed that the 32-bit hash function is enough strong with an experiment on a huge data set [KRC11]. We used the same hash function that was used in the experiment conducted by Keivanloo et al., and so we believe that the hash function is enough strong in this experiment.

Target Software Systems

The limitation of our implementation enfourced us to target only Java projects. Hence, it is necessary to conduct more experiments on other programming languages to generalize our findings.

References of Clones

As long as we used the benchmark of Bellon et al., we suffered from threats to validity of their study. That is, the references were built on only a part of collected clones. Hence, it is possible that a clone pair that did not match to any references is actually a correct clone.

This threat especially affects *precision*. The benchmark might inadequately shows low values of *precision* because of this limitation. However, the randomness of judged clones should reduce the bias of this threat.

In addition, we have to mention that the references were built from the results of 6 detectors used in Bellon et al.'s benchmark. The results of any other detectors used in this study, including the ones developed by ourselves for this study, were not considered for building the references. For fairer comparision, it is necessary to rebuild the clone references with considering all the results of all the clone detectors attending the competition.

A fairer way of compaison will be sampling each detector's output and making judgements whether they are true clones. However, we are afraid that we make judgements that are better for coarse-grained detectors. Hence, we decide to use Bellon et al.'s benchmark.

Types of Clones

The two detectors that we have implemented for this study do not consider the types of clones. However, the references judged by Bellon et al. have been labeled with their types. Hence, some other findings may be produced if we take into account types of clones.

In addtision, most of clone detectors are not very good at detecting Type-3 clones. Therefore, it has been still unclear whether the coarse-grained approach is useful in the case that Type-3 clones are taken into account.

However, it is difficult for the simple and coarse-grained approach to detect Type-3 clones. NiCad, which is one of the well-used clone detectors in the research community, can detect Type-3 clones at the block level. The granurality of detection of NiCad is the block level as well as the coarse-grained detector that is used in this study. However, NiCad compares token sequences created from each block with the algorithm to detect longest common subseqneuces. In other words, it requires a fine-grained analysis to detect Type-3 clones.

From this aspect, as well as the issue of low recall, it is not sufficient to use only coarse-grained clone detectors for detail analysis. The issue of Type-3 clones can be resolved by using Type-3 clone detector in the second stage of clone analysis.

7 Conclusion

This paper evaluated the accuracy of coarse-grained clone detection compared to other fine-grained clone detectors. We implemented a simple coarse-grained clone detector, and we compared precision and recall of the detection results of it with Bellon et al.'s benchmark.

Our experimental results showed that the coarse-grained detector achieved high precision compared to other seven detectors, with the number of detected clones reduced. They also showed that it could not achieve high recall. However, although such a coarse-grained detector missed some of correct clones, our experimental results indicate that it is not a poor detector.

In addition, we confirmed that the coarse-grained approach drastically reduced the time required to complete its clone detection. We adapted it to a corpus of over 13,000 Java projects, and we confirmed that it completed the detection task on the dataset in 152 minutes. Furthermore, we used it on multiple revisions, and confirmed that it completed the task in 17 minutes for over a thousand revisions.

Based on these findings, we conclude that such a coarse-grained detection is adequate as a first step of clone analysis. It can detect clones in a short time period and it reports fewer clones

than fine-grained detectors, which enables its users to analyze the results easily. However, it will miss some of clones, and so it is necessary to use other fine-grained detectors to perform more detailed analysis after the lightweight analysis. In such a case, the lightweight analysis helps us to narrow down the analysis target to achieve effective analysis of clones.

Acknowledgements: This study has been supported by Grants-in-Aid for Scientific Research (S) (25220003), Grant-in-Aid for Exploratory Research (24650011), Grant-in-Aid for JSPS Fellows (25-1382) from JSPS, and Grand-in-Aid for Young Scientists (A) (24680002) from MEXT.

Bibliography

[Bak95] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE'95*. Pp. 86–95. 1995.

[Baz12] S. Bazrafshan. Evolution of Near-Miss Clones. In *SCAM'12*. Pp. 74–83. 2012.

[BKA+07] S. Bellon, R. Koschke, G. Antniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. on Software Engineering* 31(10):804–818, Oct. 2007.

[BYLB98] I. Baxter, A. Yahin, M. A. L. Moura, L. Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM'98*. Pp. 368–377. 1998.

[DNR06] S. Ducasse, O. Nierstrasz, M. Rieger. On the Effectiveness of Clone Detection by String Matching. *Journal of Software Maintenance and Evolution: Research and Practice* 18(1):37–58, 2006.

[DRD99] S. Ducasse, M. Rieger, S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM'99*. Pp. 109–118. 1999.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[GÖ9] N. Göde. Evolution of Type-1 Clones. In *SCAM'09*. Pp. 77–86. 2009.

[GK09] N. Göde, R. Koschke. Incremental Clone Detection. In *CSMR'09*. Pp. 219–228. 2009.

[GK11] N. Göde, R. Koschke. Frequency and Risks of Changes to Clones. In *ICSE'11*. Pp. 311–320. 2011.

[HG12] J. Harder, N. Göde. Cloned code: stable code. *Journal of Software : Evolution and Process*, Mar. 2012. doi: 10.1002/smr.1551.

[HHK13] Y. Higo, K. Hotta, S. Kusumoto. Enhancement of CRD-based Clone Tracking. In *IWPSE'13*. Pp. 28–37. 2013.

[HJHC10] B. Hummel, E. Juergens, L. Heinemann, M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *ICSM'10*. Pp. 1–9. 2010.

[HMK12] H. Hata, O. Mizuno, T. Kikuno. Bug Prediction Based on Fine-Grained Module Histories. In *ICSE'12*. Pp. 210–220. 2012.

[HSHK10] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto. Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software. In *IWPSE/EVOL'10*. Pp. 73–82. 2010.

[IHH+12] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto. Inter-Project Functional Clone Detection toward Building Libraries - An Empirical Study on 13,000 Projects. In *WCRE'12*. Pp. 387–391. 2012.

[JDHW09] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. Do Code Clones Matter? In *ICSE'09*. Pp. 485–495. 2009.

[JMSG07] L. Jiang, G. Misherghi, Z. Su, S. Glondu. DECKARD : Scalable and Accurate Tree-based Detection of Code Clones. In *ICSE'07*. 2007.

[KBLN04] M. Kim, L. Bergman, T. Lau, D. Notokin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *ISESE'04*. Pp. 83–92. 2004.

[KKI02] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. on Software Engineering* 28(7):654–670, July 2002.

[KMM+10] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, A. E. Hassan. Revisiting Common Bug Prediction Findings Using Effort-aware Models. In *ICSM'10*. Pp. 1–10. 2010.

[Kos06] R. Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar*. 2006.

[Kos08] R. Koschke. Frontiers on Software Clone Management. In *ICSM'08*. Pp. 119–128. 2008.

[Kos13] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees and Hashing. *Journal of Software: Evolution and Process*, 2013. doi: 10.1002/smr.1592.

[KRC11] I. Keivanloo, J. Rilling, P. Charland. Internet-scale Real-time Code Clone Search via Milti-level Indexing. In *WCRE'11*. Pp. 23–27. 2011.

[Kri08] J. Krinke. Is Cloned Code More Stable than Non-cloned Code? In *SCAM'08*. Pp. 57–66. 2008.

[KRRC12] I. Keivanloo, C. K. Roy, J. Rilling, P. Charland. Shuffling and Randomization for Scalable Source Code Clone Detection. In *IWSC'12*. Pp. 82–83. 2012.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *ESEC/FSE'05*. Pp. 187–196. 2005.

[LBJP] C. Lopes, S. Bajracharya, J.Ossher, P.Baldi. UCI Source Code Data Sets. http://www.ics.uci.edu/~lopes/datasets/.

[LW08] A. Lozano, M. Wermelinger. Assessing the Effect of Clones on Changeability. In *ICSM'08*. Pp. 227–236. 2008.

[MHH+13] H. Murakami, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto. Gapped Code Clone Detection with Lightweight Source Code Analysis. In *Proceedings of the 21st International Conference on Program Comprehension* (*ICPC 2013*). Pp. 93–102. May 2013.

[MLM96] J. Mayrand, C. Leblanc, E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *ICSM'96*. Pp. 244–253. 1996.

[OSL11] J. Ossher, H. Sajnani, C. Lopes. File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly. In *ICSM'11*. Pp. 283–292. 2011.

[RBS13] D. Rattan, R. Bhatia, M. Singh. Software clone detection: A systematic review. *Information and Software Technology* 55(7):1165–1199, July 2013.

[RC07] C. K. Roy, J. R. Cordy. A Survey on Software Clone Detection Research. *School of Computing Technical Report 2007-541, Queen's University* 115, 2007.

[RC08] C. K. Roy, J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clons Using Flexible Pretty-Printing and Code Normalization. In *ICPC'08*. Pp. 172–181. 2008.

[RC09] C. K. Roy, J. R. Cordy. A Mutation/Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation'09*. Pp. 157–166. 2009.

[RCK09] C. K. Roy, J. R. Cordy, R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74(7):470–495, May 2009.

[SKR13] J. Svajlenko, I. Keivanloo, C. K. Roy. Scaling Classical Clone Detection Tools for Ultra-Large Datasets: An Exploratory Study. In *IWSC'13*. Pp. 16–22. 2013.

[SRS11] R. K. Saha, C. K. Roy, K. A. Schneider. An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies. In *ICSM'11*. Pp. 293–302. 2011.

[SYHI10] Y. Sasaki, T. Yamamoto, Y. Hayase, K. Inoue. Finding File Clones in FreeBSD Ports Collection. In *MSR'10*. Pp. 102–105. 2010.

[WG12] W. Wang, M. W. Godfrey. We Have All of the Clones, Now What? Toward Integrating Clone Analysis into Software Quality Assessment. In *IWSC'12*. Pp. 88–89. 2012.

[WHJK13] T. Wang, M. Harman, Y. Jia, J. Krinke. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *ESEC/FSE'13*. Pp. 455–465. 2013.

[YM13] A. Yamashita, L. Moonen. Do Developers Care about Code Smells? An Exploratory Survey. In *WCRE'13*. Pp. 242–251. 2013.

[ZR12] M. F. Zibran, C. K. Roy. The Road of Software Clone Management: A Survey. *Technical Report, University of Saskatchewan*, 2012.