A Code Clone Curation
— Towards Scalable and Incremental Clone Detection —

Masayuki Doi, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{m-doi, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Code clones have a negative impact on software maintenance. Code clone detection on large-scale source code takes a long time, and even worse, such detections occasionally aborted due to too much target size. Herein, we consider that we detect cross-project code clones from a set of many projects. In such a detection situation, even if a project is updated, we need to detect cross-project code clones again from the whole of the projects if we simply use a clone detection tool. Therefore we need a new clone detection scheme that rapidly detects code clones from a set of many projects with incremental detection functionality. In this paper, we propose an approach that rapidly obtains code clones from many projects. Our approach includes a strategy of multi-stage code clone detection unlike other clone detection techniques: in the first-stage detection, code clones are detected from each of the projects, respectively; then in the second-stage detection, the code clones detected in the first-stage are unified by using our clone curation technique. This multi-stage detection strategy has a capability of incremental clone detection: if the source code of a project is updated, the first-stage detection is applied only to the project and then the second-stage curation is performed. We constructed a software system based on the proposed approach. At present, the system utilizes an existing detection tool, CCFinder. We have applied the system to the large-scale source code (12 million LoC) of 128 projects. We also detected clones from the target source code by simply using CCFinder and compared the detection time. As a result, the clone detection with our system was 17 times shorter than CCFinder's detection. We also compared detection time under an assumption that each project of the targets is updated once. The experimental results showed that our incremental clone detection shortened the detection time by 91% compared to non-incremental clone detection.**

***Index Terms*—code clone detection, large-scale, incrementability**

I. Introduction

A code clone (hereafter, clone) is a code fragment which is identical or similar to another code fragment in source code. Clones may lead to software maintenance problems [1], [2] and bug propagation [3], [4]. Thus, researchers have been actively studying techniques for clone detection. For example, a number of tools for automatic clone detection from source code have been developed and released [5].

The amount of source code is steadily increasing, and large-scale clone detection has become a necessity. Large-scale clone detection can be used for detecting similar mobile applications [6], finding the provenance of components [7], and code search [8]. Thus, scalable clone detectors have been developed [9]. However, such clone detectors seek to detect all present clones in the target software, so when large-scale clone detection is conducted with such a clone detector, the clone detector outputs a huge number of clones as the results of the clone detection. It is impractical to check manually whether or not each of all detected clones is worth more than a glance. Furthermore, if a new project is added to the target projects of clone detection or a part of the target projects is updated, the clone detector needs to detect clones from all the target projects again. Such a clone detection takes a too long time. In order to solve those problems, a new technique for large-scale clone detection is required.

The results of clone detections may include many negligible clones. Negligible clones are worthless when dealing with clone information in software development and maintenance. For example, language-dependent clones are negligible clones [10]. When a specific programming language is used, a programmer cannot help writing some similar code fragments which cannot be merged into code fragments due to language limitations. A language-dependent clone consists of such code fragments, so that language-dependent clones are negligible. Many techniques have been proposed to remove the negligible clones from the clone detection results. The techniques classify clones according to whether code fragments are high cohesion and low coupling [11], according to the ratio of repetitive structures included in code fragments [10], and according to the machine learning using some metrics [12]. By using those techniques, negligible clones can be filtered out from the clone detection results. We thought that incorporating such filtering technique in clone detection process should make it possible to detect clones on a larger scale more efficiently.

In this paper, we propose a clone curation approach which rapidly detects clones from many projects. Our approach detects clones in three stages: (1) detecting intra-project clones, (2) filtering out negligible clones from the detection results, and (3) consolidating different sets of similar intra-project clones into a set of cross-project clones. The above process realizes a scalable clone detection because

- traditional clone detection, which requires a high computational complexity, is performed on small-size source code (source code of a single project)
- negligible clones are filtered out before generating cross-project clones.

We implemented the proposed technique and evaluated the scalability of the proposed technique. As a result, when the proposed technique and CCFinder [13] detected clones from 128 pseudo projects, each of which includes 100 KLoC, the proposed technique was able to detect clones up to 17 times faster than CCFinder. In addition, in order to evaluate whether the proposed technique can detect clones rapidly in the case of updating a part of the target projects, we measured the execution time required to detect clones again.As a result, our technique was able to detect clones 11 times faster than CCFinder in such a situation. Finally, we demonstrated that adjusting the parameters of our technique can detect clone faster.

The remainder of this paper is structured as follows. Section II describes the definitions of the terminology used in this paper. Section III presents our approach. Section IV describes the implementation of our approach. Section V describes the design of the evaluation experiments. Section VI describes the datasets used in the evaluation experiments. Section VII describes the results of the experiments and the discussions. Section VIII describes the threats of validity. Section IX describes the conclusion and future work.

II. Preliminaries

A. Code clone

Given two identical or similar code fragments, a **clone relation** holds between the code fragments. A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code fragments.

If a clone relation is established in given two code fragments, the code fragment pair is called a **clone pair**. An equivalence class of a clone relation is called a **clone set**. That is, a clone set is a maximal set of code fragments where a clone relation exists between any pair of them. A code fragment within a clone set is called a **code clone** or just a **clone**.

B. RNR

Many negligible clones are included in the detection results. In our previous research, we proposed a metric RNR to automatically filtering out such negligible clones [10]. $RNR(S)$ means the ratio of non-repeated code sequence in a clone set S. Here, we assume that a clone set S includes a code fragment f. $LOS_{whole}(f)$ represents the length of the whole sequence of fragment f, and $LOS_{repeated}(f)$ represents the length of the repeated sequence of f, then metric $RNR(S)$ is calculated by the following formula:

$$RNR(S) = 1 - \frac{\sum_{f \in S} LOS_{repeated}(f)}{\sum_{f \in S} LOS_{whole}(f)}$$

For example, we assume that we detect clones from following two source files (F_1, F_2). Each source file consists of the following five tokens.

$F_1 : a\ b\ c\ a\ b$
$F_2 : c\ c^*\ c^*\ a\ b$

where the superscript "$*$" indicates that its token is in a repeated code sequence.

We use label $C(F_i, j, k)$ to represent the fragment. The fragment $C(F_i, j, k)$ starts at the j-th token and ends at the k-th token in source file F_i(j must be less than k). In this case, the following two clone sets are detected from the source files.

$S_1 : C(F_1, 1, 2), C(F_1, 4, 5), C(F_2, 4, 5)$
$S_2 : C(F_2, 1, 2), C(F_2, 2, 3)$

A RNR value of each clone set is the following:

$$RNR(S_1) = 1 - \frac{0+0+0}{2+2+2} = 1.0$$
$$RNR(S_2) = 1 - \frac{1+2}{2+2} = 0.25$$

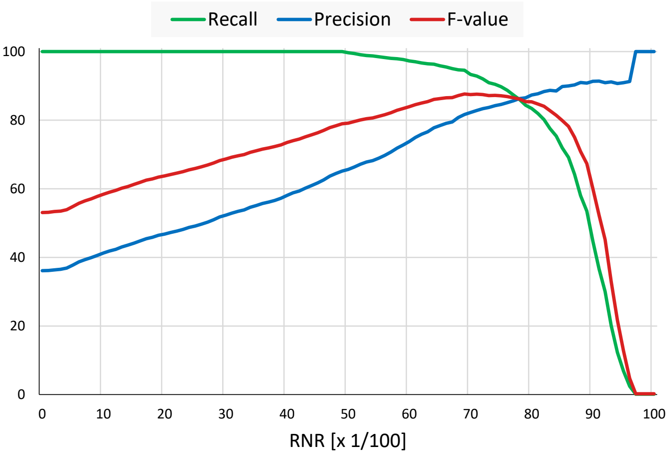


Fig. 1. Transitions of Recall, Precision, and F-value by RNR

The value 0.25 of $RNR(S_2)$ represents that most of the tokens in S_2 are in the repeated code sequence. Our previous research reported that low-RNR clone sets are typically negligible clones such as consecutive variable declarations, consecutive method invocations, and case entries of switch statements.

In our previous research, we examined how well the RNR filtering worked. We calculated precision, recall, and f-value of the filtering. The definitions of the values are the followings:

$$Recall(\%) = 100 \times \frac{|S_{negligible} \cap S_{filtered}|}{|S_{filtered}|}$$
$$Precision(\%) = 100 \times \frac{|S_{filtered}|}{|S_{negligible}|}$$
$$F - value = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

where $S_{filtered}$ is clone sets filtered out by RNR and $S_{negligible}$ is all real negligible clone sets.

Figure 1 illustrates transitions of recall, precision, and f-value when the RNR threshold varies between 0 and 1. Figure 1 means 65% of negligible clone sets are filtered out with no false positive by using 0.5 as the RNR threshold.

III. Proposed Technique

The entire process of our approach is summarized in Figure 2. It is composed of three steps: clone detection on each project, negligible clones exclusion, and clone curation cross projects. The following subsections describe the design of each step.

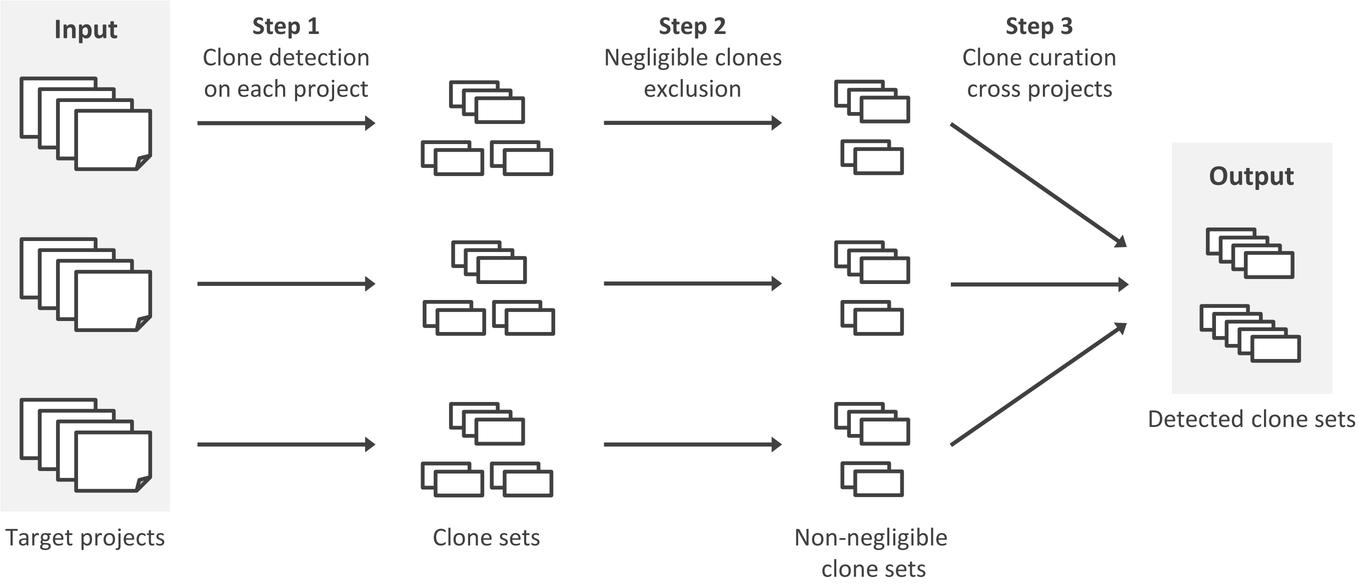

Input
Step 1
Clone detection on each project
Step 2
Negligible clones exclusion
Step 3
Clone curation cross projects
Output
Detected clone sets
Target projects
Clone sets
Non-negligible clone sets

Fig. 2. The Overview of Our Approach

A. Clone detection on each project

In this step, metric RNR is calculated from each of the detected clone sets after they are detected from each of the target projects. This process realizes a scalable clone detection because

- the size of the source code to be detected is reduced, and
- clone detection can be capable of parallel execution in each of the target projects.

In the case of detecting clones from a large project, our approach can detect clones faster by dividing the project into a set of smaller pseudo projects. However, over-division may lead to finding fewer clones to be detected because our approach cannot detect clone pairs that exist only between the projects.

We utilize an existing detection tool. Many detectors output a text file as the detection results. Consequently, our approach can be applied to many detectors such as token-based or PDG-based ones because the clone sets, which are the output of this step, can be obtained by parsing the text file outputted by detecting clones from each of the target projects. Moreover, if a clone detector has a function to calculate RNR as well as detecting clone sets, this step can be performed efficiently. For example, CCFinder [13] has a function to calculate RNR. In this case that a clone detector does not have the function, we need to calculate RNR as a post-process of clone detection.

B. Negligible clones exclusion

It is ineluctable that negligible clones are included in clone detection results. The presence of negligible clones has negative impacts from the following two viewpoints: making it more difficult to analyze clone detection results, and taking a longer time to finish clone detections. Hence, our approach excludes the negligible clones from candidates of cross-project clones. Clone sets to be excluded are satisfying $RNR < 50$ based on the previous research [10]. There are two reasons to exclude low-RNR clones: (1) users can avoid spending their time to analyze negligible clones; (2) the execution time of clone detection gets shortened. Hence the number of clone sets that are the targets of similarity calculation gets reduced, and then similarity calculation can be performed more efficiently.

C. Clone curation cross projects

The curation of cross-project clone sets is performed based on the judgment whether the similarity between clone sets is equal to or greater than the threshold.

Consequently, our technique omits similarity calculation between a pair of clone sets if the pair satisfies both the following conditions.

- The RNR value of a clone set in the pair is largely different from the one of the other clone set of the pair.
- The RNR value of either clone set in the pair is sufficiently large.

RNR is based on the code fragment structure. Similar clone sets may be the almost same RNR values. Thereby the clone detection should get faster by omitting similarity calculation between clone sets which have the big difference between each RNR values and are the one of the clone set pair's RNR is sufficiently large. More completely, RNR_m, RNR_n values of given to clone sets and two parameters θ_{diff} and θ_{max}, our approach determinates whether the similarity between clone sets are satisfying the following formula.

$$\begin{aligned} Omit(\theta_{diff}, \theta_{max}) \quad = \quad & (|RNR_m - RNR_n| > \theta_{diff}) \\ & \wedge (max(RNR_m, RNR_n) > 100 - \theta_{max}) \end{aligned}$$

where max is a function to return the maximum value in the parameters. Figure 3 shows the area of omitting similarity

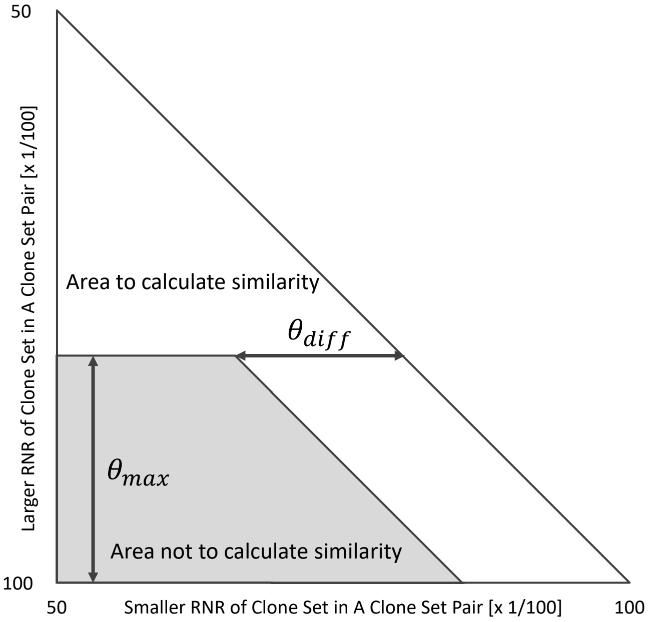

50
100
50
100
Larger RNR of Clone Set in A Clone Set Pair [x 1/100]
Smaller RNR of Clone Set in A Clone Set Pair [x 1/100]
Area to calculate similarity
θ_{diff}
θ_{max}
Area not to calculate similarity

Fig. 3. Area to Calculate Similarity Between A Clone Set Pair

calculation by the formula. The first expression is represented that RNR value of a clone set in the pair is largely different from the one of the other clone set of the pair. The θ_{diff} parameter means the threshold of the difference. The second expression is represented that the RNR value of either clone set in the pair is sufficiently large. The θ_{max} parameter means the threshold of the RNR value. For the relationship between the appropriate values of the parameters required for the similarity calculation and the detection accuracy and the speed based on them, the evaluation experiment is described in Section V-C.

IV. IMPLEMENTATION

A. Clone detection tool

Our proposed approach detects clones using a clone detection tool. Our implementation utilizes CCFinder [13] .

B. Coefficient of similarity

We define the similarity between a pair of clone sets. Hereafter we call a pair of clone sets *clone set pair*.

$$Sim(S_m, S_n) = \frac{|N(S_m) \cap N(S_n)|}{max(|N(S_m)|, |N(S_n)|)}$$

where S_m and S_n are a clone set, respectively. N is a function to return a set of n-grams created from the tokens included in a given clone set. The target tokens are extracted by a lexical analysis on the text representation of the clone set and a normalization of the identifiers and literals included in the text. N-gram size of our implementation is 5, which is on Myles et al. [14]. max is a function to return the maximum value in the parameters. The reason why we use n-gram is that we want to calculate text similarity without considering whether each instruction is repeated or not. If two clone sets have a higher similarity than a threshold θ, they are regarded as a single clone set. Otherwise, they are repeated as different clone sets as they are. At this moment, we use 0.9 as a default value. The value of 0.9 is the evaluated in Sec.VII.

V. EXPERIMENTAL DESIGN

This research evaluates our proposed approach by answering the following research questions.

A. *RQ1: How fast does our approach detect cross-project clones?*

One primary goal is to detect cross-project clones as fast as possible. In this evaluation, we create same-size pseudo projects from BigCloneBench dateset [15]. Then, we compare the execution time of two clone detections: our proposed approach and CCFinder. In the formal detection, in-project clones are detected by CCFinder from each pseudo project. And then, cross-project clones are generated by the curation. In the latter detection, cross-project clones are directly detected from the whole of the pseudo projects by using CCFinder.

B. *RQ2: How fast does our approach follow update of target projects?*

Our proposed approach utilizes the results of the clone detection per project. Herein, we assume that the source code of a project updated and we need to obtain cross-project clones again. If we use the proposed approach, we only need to detect clones again from the updated project. And then, the detection results of the updated project and the detection results of the other projects are input to the curation. If we do not use the proposed approach, we need to run CCFinder for the whole of the target projects. In this evaluation, we compare the two kinds of detections: our proposed approach and CCFinder.

C. *RQ3: How effective is parameters adjustment to improve clone detection performance?*

As described in Section III-C, our proposed approach has two parameters θ_{diff} and θ_{max} that specify the area not to calculate similarity. We seek for appropriate parameters by executing the proposed approach with different parameters.

VI. DATA SETS

In this research, we utilized two datasets, Apache [16] and BigCloneBench [15] datasets.

The Apache dataset is composed of 84 Java projects in Apache Software Foundation[1]. Unfortunately, CCFinder reported encoding errors on nine projects. This is because the nine projects included at least a file that is encoded with a character code that CCFinder cannot treat. Consequently, we excluded the nine projects from our target ones.

The BigCloneBench dataset is one of the largest clone benchmarks available to date. It was created from IJaDataset 2.0 [2], which is composed of 25,000 Java systems. The benchmark includes 2.9 million source files with 8 million manually-validated clone pairs. The BigCloneBench dataset was used for

[1] http://www.apache.org

[2] https://sites.google.com/site/asegsecold/projects/seclone

clone evaluations and scalability testing in several large-scale clone detection and clone search studies [8]. In this research, we constructed 256 pseudo projects from the BigCloneBench dataset. The 256 pseudo projects are composed of two groups, 128 projects of 10 KLoC and 128 projects of 100 KLoC. Each pseudo project consists of source files that were randomly selected from the BigCloneBench dataset.

VII. Evaluation

We conducted an evaluation of our proposed approach on a single workstation, which has the following specification.

- Windows 10
- 1.6GHz processor (6 cores)
- 32 GB Memory
- 512 GB SSD

The proposed approach curates clone sets if the similarity between the different clone sets exceeds a threshold. Hence, the choice of the similarity threshold affects the curation performance . To find the optimal threshold for curation, we measured the execution time and the number of the curated clone sets when we curate clone sets from the Apache dataset described in Sec. VI with different thresholds. Table I shows the results.

As the threshold value was reduced, The execution time increased significantly. On the other hand, as the threshold value was increased, the execution time was shortened, but the number of clone set pairs whose similarity exceeds the threshold value decreases. Hence, the optimal value is 0.9. Therefore, we utilized 0.9 as the threshold value for subsequent evaluations.

A. RQ1: How fast does our approach detect cross-project clones?

To answer RQ1, we compared the execution time of two clone detections: our proposed approach and CCFinder. We used the following thresholds in the application of our proposed technique.

- θ_{diff}: 0
- θ_{max}: 0 (Similarity calculation is not omitted.)

We utilized the dataset, which had constructed from BigCloneBench dataset described in Section VI. Figure 4 shows the overview of how to construct the dataset. In order to detect clones with changing the number of the target projects, we extracted seven sets of the pseudo projects randomly from each of the 10 KLoC group and the 100 KLoC group, respectively. The number of pseudo projects in a set i, $i \in [1, 7]$, is 2^i.

The execution time of each clone detection is shown in Figure 5. Our approach can detect clones rapidly in the both cases of 10 KLoC and 100 KLoC. The 1.24 times was the minimum value while the maximum value was 6.24 times in 10 KLoC. The greater the number of target projects is, the higher the ratio of the execution time between our proposed approach and CCFinder one is. We consider there are two reasons for that we obtained such results. Our proposed approach detects clones in parallel and it detects within-project clones from each project instead of cross-project clones from the whole dataset.

TABLE I
Execution Time and Curated Clone Sets with Different Parameterization

Threshold	0.85	0.9	0.95
Execution Time	42m 28s	26m 55s	21m 40s
#Curated Clone Set Pairs	80,104	52,785	29,762

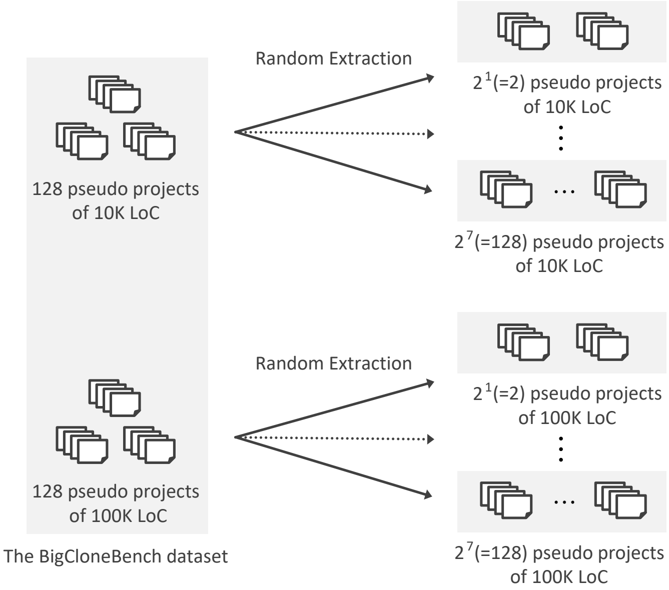
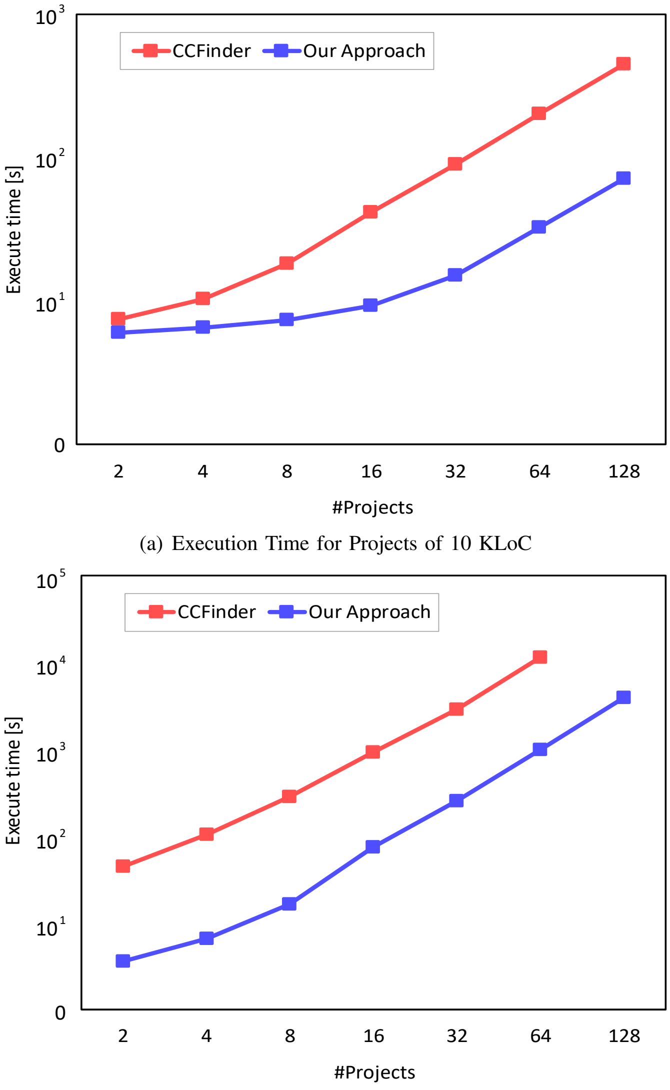

Random Extraction
128 pseudo projects of 10K LoC
2^1(=2) pseudo projects of 10K LoC
2^7(=128) pseudo projects of 10K LoC
Random Extraction
128 pseudo projects of 100K LoC
2^1(=2) pseudo projects of 100K LoC
2^7(=128) pseudo projects of 100K LoC
The BigCloneBench dataset

Fig. 4. The Overview of How to Construct The Dataset

The 11.4 times was the minimum value while the maximum value was 17.1 times in 100 KLoC. Our proposed approach was able to detect clones from the 128-project dataset of 100 KLoC. On the other hand, CCFinder was not able to finish detecting clones from the same dataset due to out of memory error. Detecting clones from vast source code requires huge memory, which is the reason why CCFinder failed to detect clones from the dataset. In our approach, CCFinder takes only a small amount of source code in a single clone detection and CCFinder is executed many times in parallel. Consequently, in our approach, out of memory error did not occur. Our approach was able to detect clones from the large dataset where CCFinder was not able to finish detecting clones due to insufficient memory.

We also evaluated the ratio of clones that were not detected by our proposed approach. The approach cannot detect clone pairs that exist only between different projects because the approach curates clones that are detected from each of the projects. Hence, we measured the ratio of clones that the proposed approach could detect. We utilized the 64 pseudo projects of the 100 KLoC as the target projects, and we compared the number of clones that could be detected by the proposed approach and the number of clones that could be detected by CCFinder. The results are shown in Fig.6. It can be seen that the ratio of clones that can be detected decreases



(a) Execution Time for Projects of 10 KLoC

(b) Execution Time for Projects of 100 KLoC

Fig. 5. Execution Time for Each Project Group

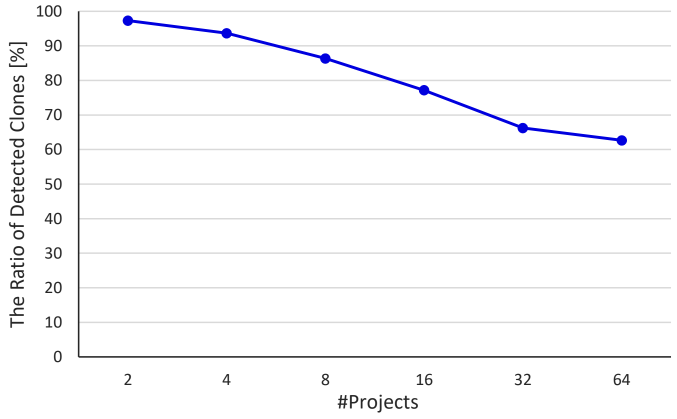


Fig. 6. The Ratio of Detected Clones

as the number of projects increases. That is because clones that exist only between the different projects increase as the number of projects increases. As a result, in the case of 64 projects of 100K LoC, the 38 percent of clones were not be detected, but the remaining 62 percent of clones can be detected 17 times faster than CCFinder.

Our answer to RQ1 is that our proposed approach can detect clones up to 17 times faster than CCFinder. Our approach can detect clones from the large target that cannot detect clones by CCFinder due to insufficient memory.

B. RQ2: How fast does our approach follow update of target projects?

In this evaluation, we assumed a situation: the source code of each target project is updated once in different dates; a user wants to detect cross-project clones as rapid as possible just after any project is updated. Under this assumption, we utilized the 64 pseudo projects of the 100 KLoC, which had been constructed in RQ1. We did not use the dataset of 128 projects because CCFinder cannot detect clones from the dataset. Thus, in the case that we directly detect cross-project clones with CCFinder, a clone detection from the whole of the 64 projects is executed 64 times. On the other hand, in the case that we apply the proposed approach, a clone detection from the updated project and a clone curation for the whole of 64 projects are executed 64 times. We measured the execution time of the both cases.

The measurement results are shown in Table II. CCFinder took 11,207 seconds while our approach took only 1,006 seconds to detect clones on average. Therefore, our approach was able to detect clones 11 times faster than CCFinder. We investigated the ratio of clone detection and clone curation in our approach and found that 97% of the execution time spent on clone curation.

Our answer to RQ2 is that the proposed approach took only 9% execution time of detecting cross-project clones from the whole of 64 pseudo projects with CCFinder.

C. RQ3: How effective is parameters adjustment to improve clone detection performance?

In the process of answering RQ1 and RQ2, we found that the clone curation step accounts for the majority of the execution time. This is because our approach calculates the similarities between all clone sets. To curate clones more rapidly, our approach needs to reduce the number of similarity calculations. We considered that omitting similarity calculation between some clone sets, which have little effect on the curation results, can curate clones more rapidly. We also considered that we could identify such clone set pairs by utilizing RNR, which is described in Section III-C. Thus, to seek for such clone set pairs, we investigated the distribution of similar clone set pairs in the 75 projects of Apache dataset described in Section VI. In Figure 7, the color of each cell

TABLE II
EXECUTION TIME OF UPDATING TARGET PROJECT INCREMENTALLY

	CCFinder	Our Approach
Average Time	11,207s	1,006s

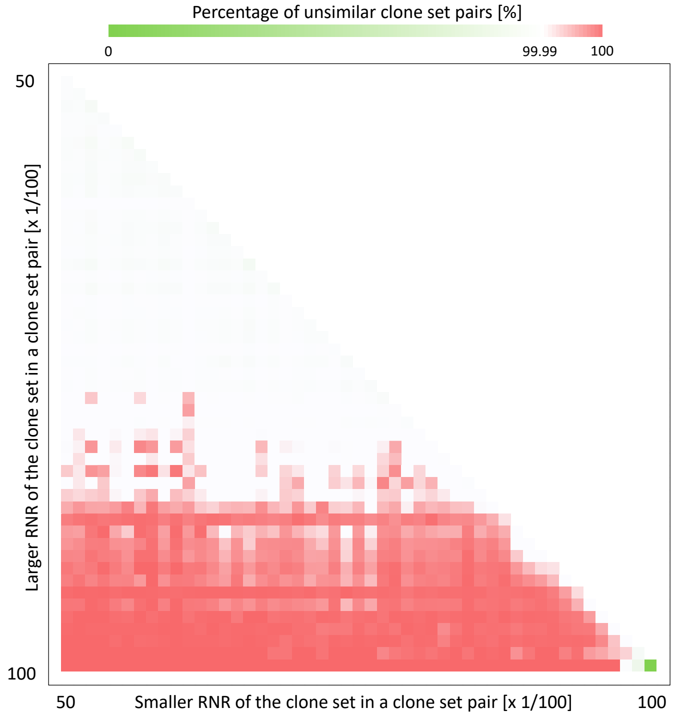


Fig. 7. The Percentage of Similar Clone Set Pairs

indicates a percentage of the clone set pairs whose X-axis and Y-axis clone sets are not similar than 0.9. The while cells are the percentage is 99.99%. The red cell shows the percentages are higher than 99.99% while the green cell means that the percentage is lower than 99.99%. We considered that clone sets in a clone set pair are not similar when the RNR values of the clone sets are largely different from each other and the RNR value of either clone set in the pair is close to large. Therefore, our approach omits the similarity calculations of such clone set pair using two parameters, θ_{diff} and θ_{max}.

In this research question, we seek for appropriate values of two parameters, θ_{diff} and θ_{max} to curate clones rapidly. We attempted to compare the execution time and the precision of clone curation for each of the parameter combinations. However, it takes too long time to curate clones for all combinations of parameters. Hence, we measured the number of similarity calculations instead of the execution time. Namely, by comparing the number of similarity calculations and the precision of clone curation, we evaluated the effectiveness of our approach to omit similarity calculations.

Figure 8(a) is a heatmap which shows the percentage of clone set pairs curated for each parameter. The while cells indicate that the percentage of curated clone set pairs is 90%. The green cell shows that the percentages are higher than 90% while the red cell means that the percentage is lower than 90%. Figure 8(b) is another heatmap which shows how much similarity calculations are omitted for each pair of parameters, θ_{diff} and θ_{max}. Each cell shows that the percentage of the similarity calculations with the given two parameters against the similarity calculations without omitting any calculations. The while cells indicate that the percentage of similarity calculations is 60%. The green cell shows that the percentages are higher than 60% while the red cell means that the percentage is lower than 60%. We can see that there are several value pairs of two parameters where their cells in both Figure 8(a) and Figure 8(b) are white. Namely, there are parameters which can reduce the number of similarity calculations to 60 percents with keeping the 90 percents of curated clone set pairs. Thus, we sought for the optimal pair of parameters using the heatmaps and we measured the number of similarity calculations that could be omitted when the optimized parameters were utilized.

Figure 9 shows the relationship between the ratio of curated clone set pairs and the ratio of similarity calculation reduction. According to Figure 7, we can see that there are multiple pairs of the two thresholds that yield the same ratio of curated clone set pairs. Herein, we focus on the pair of the two parameters that reduces similarity calculation at a maximum, and we investigated the relationship between the two ratios. As a result, we found that 41% of similarity calculations can be omitted by sacrificing 5% of the curations.

Our answer to RQ3 is that using appropriate values of parameters can reduce 41% of similarity calculations only by sacrificing 5% of the clone curations.

VIII. Threats of Validity

A. Clone detector

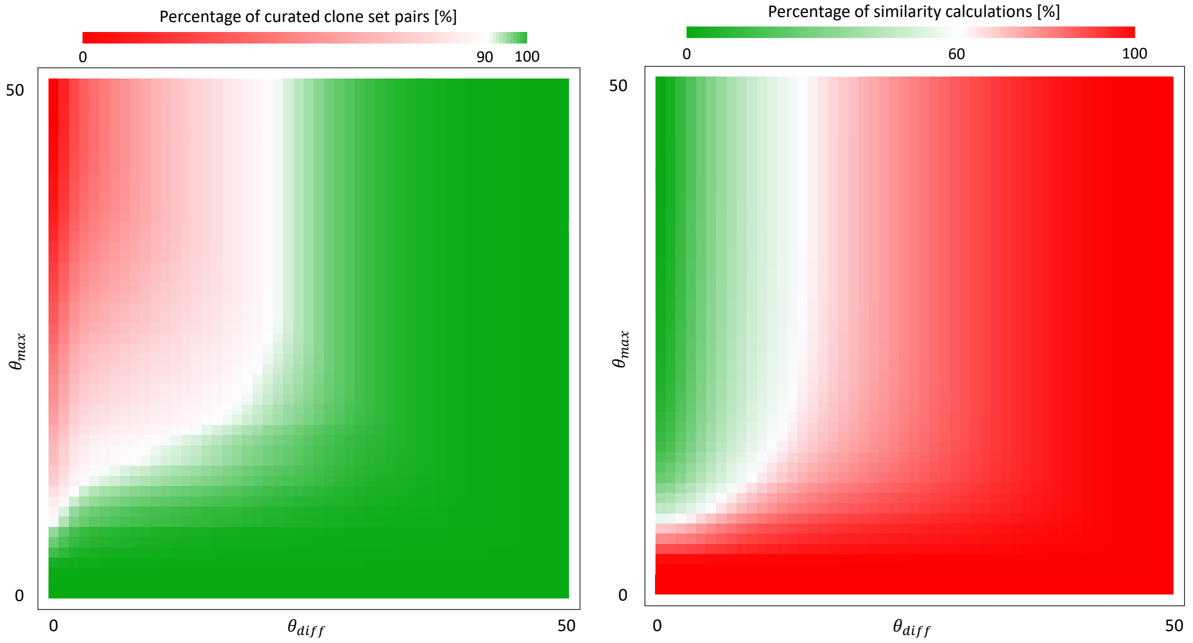
In the evaluations, we utilized CCFinder as a clone detector. However, there are many clone detectors besides CCFinder. Applying other clone detectors in our approach does not necessarily get more rapid than conventional clone detections unlike CCFinder. We are planning on evaluating whether or not other clone detectors work well in our approach.

B. Experimental target

In the evaluations, we succeeded in reducing similarity calculations by 41% on the Apache projects dataset. However, if we use other datasets, we may obtain different ratios of similarity calculations.

IX. Conclusion

In this research, we proposed a technique to detect clones rapidly. The proposed technique curates the results of clone detections for each of the projects. As a result of evaluating the performance of our technique, it was possible to detect clones up to 17 times faster with CCFinder. In addition, the execution time of incremental updates was able to be reduced to nine percents of previous ones. Furthermore, we showed that using appropriate values of parameters can reduce 41% of similarity calculations only by sacrificing 5% of the clone curations. We plan to improve our technique so that it can be applied to clone detectors other than CCFinder such as SourcererCC [9] or NiCAD [17] in the future.



(a) The Percentage of Curated Clone Set Pairs

(b) The Number of Similarity Calculations

Fig. 8. Heatmap of Curated Clone Set Pairs and Similarity Calculations for Pairs of Each Parameter

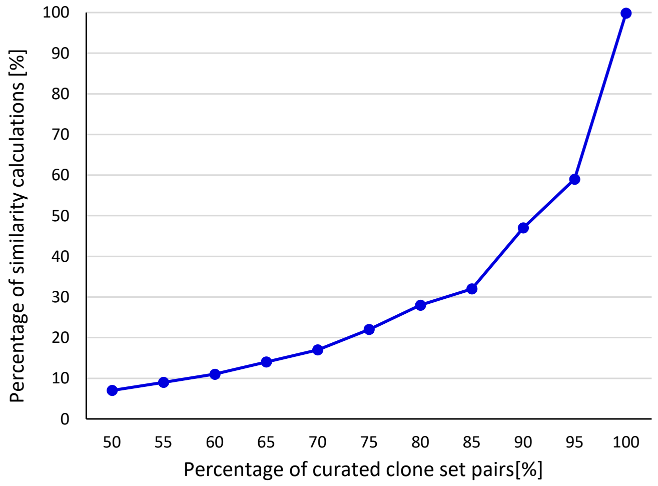


Fig. 9. Transitions of Similarity Calculations and Curated Clone Set Pairs

References

[1] A. Lozano and M. Wermelinger, “Assessing the effect of clones on changeability,” in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 227–236.

[2] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider, “An empirical study of the impacts of clones in software maintenance,” in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 242–245.

[3] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that smell?” in *Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 72–81.

[4] T. Zhang and M. Kim, “Automated transplantation and differential testing for clones,” in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17, 2017, pp. 665–676.

[5] A. Sheneamer and J. Kalita, “A survey of software clone detection techniques,” *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[6] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 175–186.

[7] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, “Software bertillonage: finding the provenance of an entity,” in *Proceedings of the 8th working conference on mining software repositories (MSR)*, 2011, pp. 183–192.

[8] C. Ragkhitwetsagul and J. Krinke, “Siamese: scalable and incremental code clone search via multiple code representations,” *Empirical Software Engineering*, pp. 1–49, 2019.

[9] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcerercc: scaling code clone detection to big-code,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1157–1168.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Method and implementation for investigating code clones in a software system,” *Information and Software Technology*, vol. 49, no. 9, pp. 985–998, 2007.

[11] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, “How to extract differences from similar programs? a cohesion metric approach,” in *2013 7th International Workshop on Software Clones (IWSC)*, May 2013, pp. 23–29.

[12] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, “Automatic clone recommendation for refactoring based on the present and the past,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 115–126.

[13] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[14] G. Myles and C. Collberg, “K-gram based software birthmarks,” in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ser. SAC ’05, 2005, pp. 314–318.

[15] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in *In Proceedings of the Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, 2014, pp. 476–480.

[16] Y. Higo and S. Kusumoto, “How should we measure functional sameness from program source code? an exploratory study on java methods,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 294–305.

[17] J. Cordy and C. Roy, “The nicad clone detector,” *IEEE International Conference on Program Comprehension*, pp. 219–220, June 2011.