Evaluating Performance of Clone Detection Tools in Detecting Cloned Co-change Candidates

Abstract

Most of the changes in a software system are done by reusing existing code pieces which creates source code clones in the codebase. To maintain consistency in a software system, these code clones may need to be changed together (co-changed) during software evolution. Detecting cloned co-change candidates is essential for clone tracking. Earlier studies showed that clone detection tools can be used to enhance the performance of finding cloned co-change candidates. Though there are several studies to evaluate the clone detection tools based on their accuracy in detecting cloned fragments, we found no study which compares different clone detection tools in the perspective of detecting cloned co-change candidates. In this study, we explore this dimension of code clone research. We used 12 different configurations of nine promising clone detection tools to identify cloned co-change candidates from eight open-source C and Java-based subject systems of various sizes and application domains and evaluated the performance of those clone detection tools in detecting cloned co-change fragments. Evaluated rank list and relevant analysis of obtained results provide important insights and guidelines about selecting the clone detection tools which can enrich a new dimension of code clone research in change impact analysis of software systems.

Keywords: Clone Detection; Cloned Co-change Candidates; Commit operation; Software Maintenance and Evolution

1. Introduction

Although a large number of clone detection tools currently exist, we found no study for comparing the performance of different tools based on their ability to be used in software maintenance activity such as predicting cloned co-change candidates during software evolution. In this study, we wanted to explore, whether a good clone detector also performs well in detecting cloned co-change fragments? One of the common features of clone detection tools is to combine similar code fragments into a clone group or class. The code fragments in a particular clone class are expected to perform similar functionalities. If we want to make changes to a particular clone fragment in a clone class, the other fragments in the class are likely to have similar changes to ensure consistency of the code-base. Considering this assumption, we can say that all the clone fragments in a clone class have the possibility of being a cloned co-change candidate with any change of that class members. We utilize the clone classes provided by the clone detectors for these types of co-change prediction.

Finding the co-change candidates of a target code fragment is also known as change impact analysis [1] in the literature. Mondal et al. [2] investigated whether a clone detection tool can enhance the performance of an evolutionary coupling based tool in finding change impact set or co-change candidates. They performed their investigation using Nicad for detecting both the regular and micro-clones and found that use of detected clone results significantly enhance the performance of Tarmaq Rolfsnes et al. [3]. As they only analysed the use of Nicad, in this study we wanted to compare some other good clone detection tools to find whether these tools can perform better for detecting co-change candidates. We have evaluated four different configurations of CloneWorks [4] and eight other clone detectors in our investigation. Therefore, we have a total of 12 separate implementations of clone detection tools (we will consider them as 12 separate tools in the rest of this paper). We apply these tools on eight open-source software systems. Configuration of the clone detection tools are given in Table 3 and the software systems used in this study are reported in

Table 1.

During software evolution, a developer makes changes in the code-base to fulfil some change requests. Those change requests could be related to each other or independent [5, 6]. Therefore, all the changes done in a single commit need not be related to each other. Some changes in a single commit may be dependent on each other and some may be independent. The related code fragments are known as the co-change candidates in literature [7]. Some of those co-change candidates may contain similar code-fragments i.e. they are clones of one another, on the other than, other types of co-change candidates may not be cloned fragments but they have a functional dependency or coupling with each other. If a developer makes changes to a target code fragment, those changes might also need to be reflected other similar fragments in the code-base to ensure consistent evolution of the software system [2, 8]. Failing to change a co-change candidate of a target fragment can introduce bugs in the software system [9, 10].

We have analyzed thousands of commit operations from the evolutionary histories of eight subject systems listed in Table 1. While analyzing a commit operation, we identify which code fragments changed together (i.e., co-changed) in that commit. Considering each fragment as the target fragment, we try to predict the other actually co-changed fragments using each of our clone detectors. We found some change fragment which is not detected by any of the clone detectors. We excluded those change fragments from consideration during calculating the performance measures of clone detectors. An example of our detection process is demonstrated in Fig. 1. Let us assume that 21 changes, C1 to C21, occurred in the code-base of a subject system in a particular commit operation. We detect these changes using the UNIX diff operation. If we consider C1 as the target change, the other 20 changes, C2 to C21, are the actual co-change candidates (i.e., co-changed candidates) of C1. We apply different clone detectors to detect these co-change candidates for the target change C1. Let using Deckard we can detect five change fragments (C2, C6, C8, C15, C21) from those 20 fragments, similarly using Nicad we can detect four fragments (C5, C10, C16, C18). We will continue to detect co-change fragments using all the other clone detectors. After getting the results from all the clone detectors, we find 10 unique change fragments (C2, C5, C8, C15, C21, C5, C8, C10, C15, C21) out of 20 fragments by taking a union of the results of all the clone detectors. We will take those 10 unique change fragments as cloned co-change candidates and calculate the precision and recall of each of the clone detectors based on their number of detection among those cloned co-change candidates. For each subject system, we finally calculated average recall, average precision, and F1 Score for each of the clone detector and then compare the clone detectors based on their weighted average F1 Score considering all the subject systems in this study. Figure 2 shows the bar chart of Average Recall and Average Precision drawn from our experimental results and in Table 5 we have given the calculated weighted average of F1 Score. According to our findings and ranking of the clone detectors (Table 6), we can conclude that CloneWorks (both two configurations, Type-3 Pattern and Token), Deckard, and CCFinder outperforms all the other tools. CloneWorks Type-2 Blind, ConQAT, and iClones fall in the following order. From the final rank list, we also see that the clone detection tools which detecting only Type-1 clones (such as Duplo, CloneWorks Type-1) are performing worst in finding co-change candidates. We also calculated the average number of distinct lines detected as cloned lines by each of the clone detectors in all the revisions of all the subject systems (Figure 3) and found that the clone detector which detects more distinct lines as a cloned line in the code-base also performs well in detecting cloned co-change candidates.

Based on this study, we tried to answer the following research questions:

RQ1: What is the comparison scenario of the clone detectors in predicting cloned co-change candidates?

RQ2: Why do different clone detectors perform differently in detecting cloned co-change candidates?

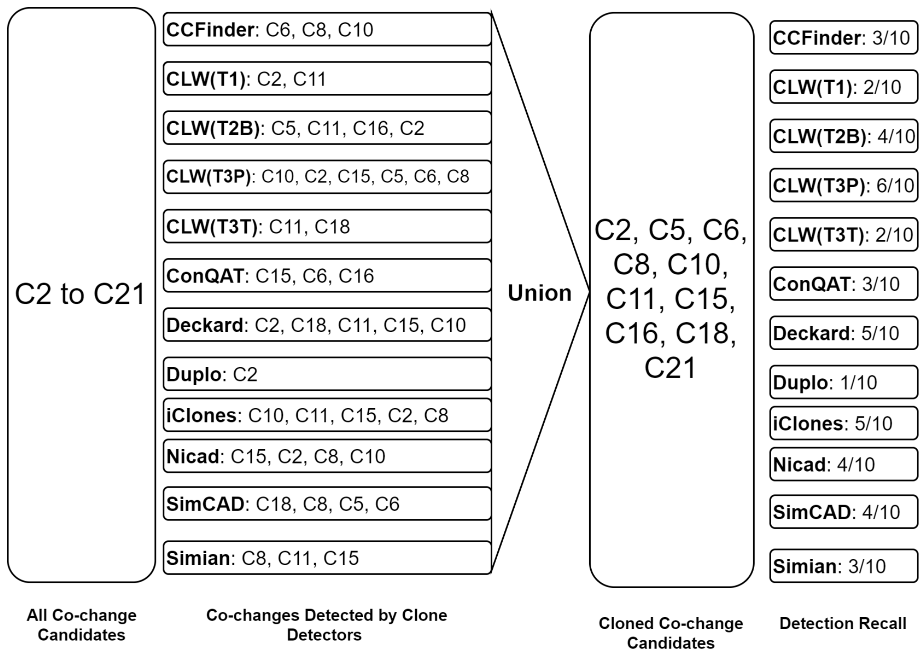

Figure 1: Demonstrating cloned co-change detection process

RQ3: Do the source code processing techniques (Pattern/Token/Text-based processing) of the clone detection tools have any impact on their performance in detecting co-change candidates?

RQ4: Do clone detection tools designed for detecting different types of clones (Type 1, 2, 3) work differently in detecting cloned co-change candidates?

To the best of our knowledge, our study is the first one to compare clone detection tools considering a particular maintenance perspective (e.g., considering their capabilities in successfully suggesting cloned co-change candidates during software evolution). From an initial assumption, it is obvious that a clone detector which is good in detecting cloned fragments should also be good in detecting cloned co-change candidates. In this exploratory study, we wanted to practically verify this assumption. We selected 12 implementations of clone detectors detecting different types of clones in our investigation to verify whether they are also good at detecting co-change candidates. According to our investigation and analysis, we find that the clone detectors which detects Type-3 clones and performs pattern-based source code processing are significantly good in detecting cloned co-change candidates. Our investigation also shows that tools which provide more number of clone fragments and cover more source code lines are also good in detecting cloned co-change candidates. We have created a final rank list of the clone detection tools based on our investigation which is shown in Table 6 where we have considered the ranking of the clone detectors in each of the subject systems to make a final ranking. A clone detector which performed well in most of the subject systems got a higher rank in this ranking table. Considering the rank list in Table 6 we find: (i) the tools which are good in detecting all types (1/2/3) of clones are also good in detecting cloned co-change candidates. (ii) Top two of the tools in final rank list are the Type-3 configurations of CloneWorks (one splits source files with lines and the other split the source file with tokens to process it before detecting clone fragments), and other following clone detectors which perform well are Deckard and CCFinder. Therefore, we can conclude that to detect cloned co-change candidates, those tools (all are pattern and token-based) are the best choices compared to the other tools used in this study. (iii) From this ranking, we can also find that text-based clone detectors (such as Duplo or CloneWorks Type-1) are not good for detecting co-change candidates. (iv) Our comparison in Figure 3 also shows that the clone detectors which detect a higher number of clone fragments and cover a higher number of unique lines in the source files are performing good in detecting cloned co-change candidates. We have also performed The Wilcoxon Signed-Rank Test [11, 12] to verify whether the F1 Scores in all the eight subject systems of the tools which got higher ranks in the final rank list (Table 6) are significantly better compared to the other clone detection tools or not. The results of our significant test are described in Section 4.5. A summary of our significance test results is in Table 7, which shows that four out of the 12 clone detection techniques of this study perform significantly better than the other techniques in detecting cloned co-change candidates.

We organized this paper in the following sections: Some related works are described in Section 2, our methodology is in Section 3, we described the experimental result in Section 4, the discussion is in section 5, Section 6 explains some possible threats to validity, and we conclude our paper in Section 7.

This paper is a significant extension of our previous work [13] on detecting cloned co-change candidates using different clone detectors. Our previous work answered two research questions by analysing six clone detectors on six open-source software systems. Two research questions in our earlier study showed that even though a tool which is good in detecting clone fragments from software systems may not be good in detecting cloned co-change candidates. The tools which detect more clone fragments and cover more unique lines in the source files are found good in predicting cloned co-change candidates. We extend our previous work by answering two additional research questions (RQ3, RQ4) to find more specific reasons for the variation of the performance by clone detectors in detecting co-change candidates. We have also increased the generalizability of the previous study by adding two more software systems as subject systems and three more clone detection tools with four different configurations of CloneWorks ((Type-1, Type-2 blind, Type-3 pattern, and Type-3 token) totalling eight subject systems and 12 clone detector executions. Therefore, our implementation has been upgraded from 6X6 to 12X8 (Clone detector X Subject Systems) in the current version of the study. In this study, we have shown that the performance of clone detection tools in detecting cloned co-change fragments not only dependent on the number of clone fragments detected and the lines covered in the source file by those fragments but also the type of detected clones and underlying source code processing techniques also have some impacts.

2. Related Work

There are several studies [14, 15, 16, 17] that have been focused on ranking different clone detection tools based on their performance and accuracy in detecting different types of clone fragments. Burd and Bailey [18] did a study for comparing the performance of three clones and two plagiarism detecting tools based on their precision and recall of the ability to detect duplicated codes in a single file or across different files. Bellon et al. [16] evaluated six clone detection tools based on eight large C and Java programs of almost 850 KLOC and made a framework for comparing different clone detection tools with the data validated by one of its authors. Rysselberghe and Demeyer [19] evaluated three representative clone detection techniques from a refactoring perspective where they provided comparative results in terms of portability, kinds of clone reported, scalability, number of false positive, and number of useless clone detection. Svajlenko and Roy [15] evaluated eleven modern clone detection tools using four benchmark frameworks and noted ConQAT, iClones, NiCad and SimCAD as very good tools for detecting clones of all the three types (Type-1, Type-2, Type-3). Roy et al. [14] did a qualitative comparison and evaluation of the latest clone detection approaches and tools, and made a benchmark called BigCloneBench [20] which contains eight million manually validated clone pairs in a large inter-project source dataset of more than 25,000 projects and 365 million lines of code. They categorize, relate and assess different clone detection tools based on two different points of view such as classification based on the overlapping set of attributes in the different code fragments and the scenarios how Type-1, Type-2, Type-3, and Type-4 clones created. They also elaborated the procedure of using the result of their study to select the most suitable clone detection tool or technique in the context of a specific set of areas and limitations.

There are some studies which not only proposed a clone detection mechanism but also did a comparison of their proposed technique with some existing techniques. Koschke et al. [21] provided a technique to detect clone using suffix trees in abstract syntax trees and they also made a comparison to other techniques using the Bellon benchmark for clone detectors. Ducasse et al. [22] and Selim et al. [23] also utilized Bellon's framework for measuring the performance of their proposed clone detection tools based on string comparison and intermediate source transformation respectively. Selim et al. [23] showed that their tool is capable of detecting Type-3 clones and their technique is better than the source-based clone detectors based on the value of recall through a slight drop in the precision using Bellon's corpus where clone group is not complete. Compared to the standalone string and token-based clone detectors, their technique showed a little higher precision.

All the studies which compared different clone detectors have been focused on the precision, recall, computational complexity, and memory used or detecting a specific type of clone fragments such as Type-1, Type-2, Type-3, or Type-4 during the detection approach of duplicated code in a code-base. Our study to compare clone detectors is completely different from the previous comparisons. We do not want to compare clone detection tools based on the capability to detect clones. Our point of interest is to detect co-change candidates during the software commit operations. Mondal et al. [7] did a study to predict and rank the co-change candidates by analyzing evolutionary coupling from previously done change history using generated clone fragments by NiCad but they did not consider the result of other clone detection tools and also did not show any comparative study among different clone detection tools in doing such prediction and rank of co-change candidates. This work is an extended version of our previous study [13] using six clone detection tools on six software systems written in C and Java programming languages to compare those tools based on the performance of detecting clone co-change candidates. We found no other study which has performed a similar comparison of clone detectors. To extend our previous research, we have analyzed the performance of nine clone detection tools in 12 different configurations based on their capabilities in finding co-change candidates during software evolution using their generated clone results. According to our knowledge, this is the first such investigation of performance with clone detection tools.

3. Methodology

We have used eight open-source software systems, having varieties of size and application domain as subject systems in this study. The list of subject systems are in Table 1. To detect cloned co-change candidates from those subject systems, we executed 12 clone detection tools (Table 3) and analyzed obtained results to evaluate the performance of those clone detection tools. Our analysis aims to rank these clone detection tools based on their performance in successfully suggesting actual co-change candidates (ACC) during the software evolution. Before starting our main analysis, we have to resolve some issues and we have taken the following considerations in this regard.

Selection of subject systems: To select subject systems for this study, we considered both the popularity of programming language and availability of a considerable amount of revisions. According to the TIOBE Programming Community index [24] (an indicator of the popularity of programming languages), Java is dominating the list of popular programming languages for more than the last ten years and C is the second most popular programming language within this period. Considering this fact, we wanted to select subject systems written in these two programming languages. Our other consideration was the availability of a considerable amount of revisions of each of the systems. Based on both of the considerations, we have chosen the subject systems listed in Table 1. Four of our eight subject systems are written in C programming language and the other four are in Java. To increase the generalizability of the study we have added systems having diverse size and application domains.

Selection of clone detectors: In this research, we wanted to examine those clone detection tools which are good in detecting all types of clones. To select such tools, we considered some related studies. We have taken CloneWorks [4] as it is considered as a fast and flexible clone detector for large-scale near-miss clone detection experiments. CloneWorks tool provides the ability to change its processing mechanism by changing its configuration files. We applied four different configurations of CloneWorks to detect Type-3 Pattern, Type-3 Token, Type-2 Blind, and Type-1 clones for investigating the impact of the types of clones in detecting co-change candidates. We included Duplo [25] as another type-1 clone detector for making the comparison with type-1 clones of

Table 1: Subject Systems

Systems	**Language**	**Domains**	**Revisions**
Brlcad	C	Computer Aided Design	2115
Camellia	C	Batch Job Server	301
Carol	Java	Game	1700
Ctags	C	Code Def. Generator	774
Freecol	Java	Game	1950
Jabref	Java	Reference Manager	1545
jEdit	Java	Text Editor	4000
Qmailadmin (QMA)	C	Mail System Manager	317

Table 2: Summary of Data Processed

Revisions/ SS	**Brlcad**	**Camellia**	**Carol**	**Ctags**	**Freecol**	**Jabref**	**jEdit**	**QMA**
Processed	2113	301	1700	774	1001	1540	215	317
Experiencing change	660	163	454	447	836	860	145	35
Experiencing more than one change	553	155	430	330	833	755	145	25

CloneWorks in this investigation. ConQAT [26], iClones [27], NiCad [28], and SimCAD [29] have been reported as very good tools for detecting all types of clones in the study of Svajlenko and Roy [15]. Besides these, CCFinder [30], Deckard [31], iClones and NiCad are often considered as common examples of modern clone detectors that support Type-3 clone detection. CCFinder is known as a multi-linguistic token-based code clone detection system for large scale source code. Inclusion of CCFinder enriched the variation of detected clone fragments in the extended study. To make more comparison of the performance of type-1 clones in detecting co-change candidates we added Duplo in our study. The reason of taking Simian [32] in our analysis was its ability to find duplicated code by line-by-line textual comparison supporting identifier renaming with a fast detection speed on the large repository and extensive use in several clone studies [33, 34, 35, 36, 37]. NiCad, SimCAD, and Simian are textual similarity-based clone detection tools. Deckard works using tree comparison based technique. CCFinder, ConQAT and iClones are token-based clone detection tools.

Determining if the extracted co-changes are related to each other or not: Even though we have extracted all the changes between two adjacent revisions (i.e., revision n and n+1), it is not possible to fully guarantee that all the changes are actually co-change candidates of each other. There might be some changes which do not depend on any other changes i.e. they may change independently. The inclusion of such dissimilar changes into our calculation can drop the detection accuracy of clone detectors. To minimize such drops, we excluded those co-changes which are not detected by any of the 12 clone detection techniques in our study. As none of the clone detectors considers them as co-change candidates, we considered those changes as dissimilar or independent changes.

Ensuring if the configuration parameters of all the clone detection tools identical with each other or not: As we wanted to compare different clone detectors based on their capability of successfully suggesting co-change candidates, it was important to configure them identically during detecting clones from our subject systems. Wang et al. [34] introduced confounding configuration choice problem where the configuration of different tools during clone detection may play a vital role and the result may be best or worst depending on the configuration. Our configuration of different tools is shown in Table 3. We have used similar configurations for each of the tools for obtaining a consistent result. We have taken configuration values similar to Svajlenko and Roy [15] which they conducted to compare different clone detectors based on their efficiency in detecting cloned fragments. We provided 70% similarity threshold for all the clone detection tools (except Deckard) which takes similarity dissimilarity value as a parameter. We have used 85% as the similarity threshold for Deckard 85% because we found a lot of unwanted clones in the result if we use the similarity threshold 70%. These results include a lot of duplicated clone fragments and showing a lot of fragments as a clone to itself several times. We also tried some other percentage values such as 75%, and 80% but the detected result of Deckard becomes much desirable when we set it to 85%. Svajlenko and Roy [15] also used 85% similarity while running Deckard for Mutation Framework. We have also selected identical parameter values such as the minimum number of tokens, the minimum number of lines for different clone detection tools. As we wanted to compare different clone detectors based on their capability of successfully suggesting co-change candidates, it was very important to configure them identically during detecting clones from our subject systems.

The overall approach: Our overall processing is performed in some distinct steps. Initially, we downloaded all the source files of all the revisions of all the subject systems from their respective SVN repositories. We then applied **diff** operation between each file of a revision with the respective file in the next revision and extracted the change information such as Name of the File which is changed, the Line where the respective change begins, the Line where the change is ended from the output of **diff**. We did the change extraction for each of the revision (excluding the last one) of all the subject systems. After detecting all the changes, we started the clone detection on all the revisions of all the subject systems using all the clone detection tools. We started our main analysis to find the accuracy of each of the clone detection tools after having the result of all the clone detectors and change information from all the revisions.

The mechanism of calculating accuracy is demonstrated in our introduction using Fig. 1. Suppose, we are examining a particular commit operation. The number of fragments that were changed in this commit operation is n. Now, let us consider one of these n fragments as the target fragment. Then the other $n-1$ fragments are the actually co-changed candidates for the target fragment. We excluded the non-cloned co-change candidates using the approach described in the introduction. After this exclusion, we get the **Actually Cloned Co-change** (ACC) for each of the target fragments.

Let us assume that the target change fragment intersects a particular clone fragment from a particular clone class. The other fragments in that clone class are considered as the **Predicted Cloned Co-change** (PCC) candidates. We now determine how many of these PCC intersect with the ACC to obtain the number of detected cloned co-change candidates by the clone detector.

These counts of predicted and actually co-changed candidates are considered as the **true positives** to calculate Recall, Precision, and F1 Score. We calculate these using the following equations (Eq. 1, 2, and 3).

$$Recall = \frac{|PCC \cap ACC|}{|ACC|} \tag{1}$$

$$Precision = \frac{|PCC \cap ACC|}{|PCC|} \tag{2}$$

$$F1\ Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

We repeat the calculating process of Recall and Precision for all the changes in each of the subjects systems with the detected clone fragments generated by all the clone detection tools. We then calculate F1 Score of the clone detectors for each of the subject systems by taking the average values of Recall and

Precision which is reported in Table 5. We reported the ranking of the tools considering individual ranks in each of the subject systems in Table 6.

Producing the final rank list: To produced the final rank list of 12 clone detection techniques we considered their performance in all the eight individual subject systems. Our ranking approach is demonstrated in Table 6 which shows both the ranks in individual subject systems and final overall ranking for each of the clone detectors. S1, S2, ..., S8 are the subject systems used in this study and these are in the same order as shown in the Table 5 which shows the F1 Scores of detecting cloned co-change candidates by each of the clone detection tools in the respective software systems. The highest F1 Score in Table 5 got rank-1, and similarly the lowest one got rank-12 in the respective position of Table 6. Therefore, every clone detection technique has eight rank values (smaller value represents the better performance) which are obtained in the eight software systems. We then took the summation of those eight individual ranks for producing the overall ranking for each of the clone detection techniques. The clone detector which got the smaller summation value of individual ranks performed well in most of the subject systems. Based on the summation of individual rankings, we reported the final rank of each clone detection technique in the right-most column of the Table 6.

4. Experimental Result

In this section, we will answer the research questions based on our overall analysis and obtained results by processing each of the eight subject systems using all the 12 clone detection tool executions.

*4.1. Answer to the **RQ1***

What is the comparison scenario of the clone detectors in predicting cloned co-change candidates?

The key experimental results are in Figure 2, Table 4, Table 5, and Table 6 where Fig. 2 shows the average Recall and average Precision of each of

Table 3: Configuration of Participating Clone Detection Tools

Tools	**Configuration for Clone Detection**
CCFinder	min. size: 50 tokens, min. token types: 12
CLW(T1)	termsplit=token, termproc=Joiner
CLW(T2B)	cfproc=rename-blind, cfproc=abstract literal, termsplit=token, termproc=Joiner
CLW(T3P)	cfproc=rename-blind, cfproc=abstract literal, termsplit=line
CLW(T3T)	termsplit=token, termproc=FilterOperators, termproc=FilterSeperators
ConQAT	block clones, clone min-length=5, gap ratio=0.3
Deckard	min. size: 30 tokens, 5 token stride, min. 85% similarity
Duplo	min. size: 10 lines, min. characters/line:1
iClones	minimum block: 30, minimum clone: 50, All Transformation
Nicad	block clones, blind renaming, max. threshold=0.3, minimum lines=5, maximum lines=2500
SimCAD	block clones, Source Transformation= generous
Simian	min. size: 5 lines, normalize literals/identifiers

CLW: *CloneWorks;* ***T1:*** *Type-1;* ***T2B:*** *Type-2, Blind Renaming;* ***T3P:*** *Type-3, Pattern;* ***T3T:*** *Type-3, Token;*

Table 4: Summary of Actual Target and Co-change Candidates

SS	# ATC	# ACC	% ATC	% ACC
Brlcad	2909	33578	7.45	1.89
Camellia	8052	346140	20.61	19.46
Carol	4582	254311	11.73	14.29
Ctags	718	3648	1.84	0.21
Freecol	6865	265213	17.57	14.91
Jabref	8313	455469	21.28	25.60
jEdit	5122	323277	13.11	18.17
QMA	2508	97396	6.42	5.47
Total	**39069**	**1779032**	**100**	**100**

SS: *Subject Systems*

 # ATC: *Number of Actual Taget Changes*

 # ACC: *Number of Actual Co-changes*

the clone detection tools. Table 4 shows the summary of target changes and detected co-change candidates for those target changes in each of the subject systems. We found the highest and lowest percentage of target change and its cloned co-change candidates from Jabref and Ctags respectively. Table 5 shows the F1 Score of each of the clone detectors in each of the subject systems. The F1 Score is calculated using Equation (3). Our experimental results concluded in Table 6 which shows that CLW(T3P), CLW(T3T), and Deckard shows top performance (Rank 1 or 2) in most of the subject systems compared to all the other tools. The summary of the results in the Table 6 shows that among the subject systems, CLW(T3P) is the best in all the subject systems except Camellia and Freecol where Deckard is showing the best performance. CLW(T3T) shows the second-best performance in most of the subject systems. An overall observation on individual rankings of different clone detection techniques reveals that CLW(T3P), Deckard, CLW(T3T), CCFinder show better performance in most of the subject systems compared to the other clone detectors. On the other hand, Duplo, CLW(T1) shows the worst performance in most of the subject systems. Other tools show average performance considering individual ranking in different subject systems. CLW(T1) and Duplo obtained the bottom position in the final rank list.

As our analysis was based on the clone grouping into class or pair provided by the clone detection tools, we found that the efficiency of clone detection tools in suggesting cloned co-change candidates is mostly dependent on its effectiveness in making clone class/ pair. The tool which groups functionally similar clone fragments into a clone class/ pair effectively can perform well in successfully suggesting cloned co-change candidate(s). Different values of the accuracy of different clone detectors indicate the difference in their efficiency in this research domain.

*4.2. Answer to the **RQ2***

Why do different clone detectors perform differently in detecting cloned co-change candidates?

From the answer of our **RQ1**, we found a difference in performance for different clone detection tools in suggesting cloned co-change candidates. We found a clone detection tool which is good in detecting clone fragments may not be good at detecting cloned co-change candidates. This motivates us to find out the reason to answer this research question.

We investigated the number of clone fragments and the number of unique lines covered by those clone fragments by all the 12 clone detectors from all the revisions of all the subject systems. Figure 3 shows the comparison scenario of the number of clone fragments and line covered by those clone fragments from different clone detectors. For better comparison, we bring the values in a single scale (between 0 and 1) where 0 and 1 represent the lowest and highest values respectively compared to all the clone detectors under comparison. Considering both, the number of clone fragments and the number of lines covered by those clone fragments from all the revisions of all the subject systems, if we order the clone detectors from the highest to the lowest, we find Deckard and CLW(T3P)

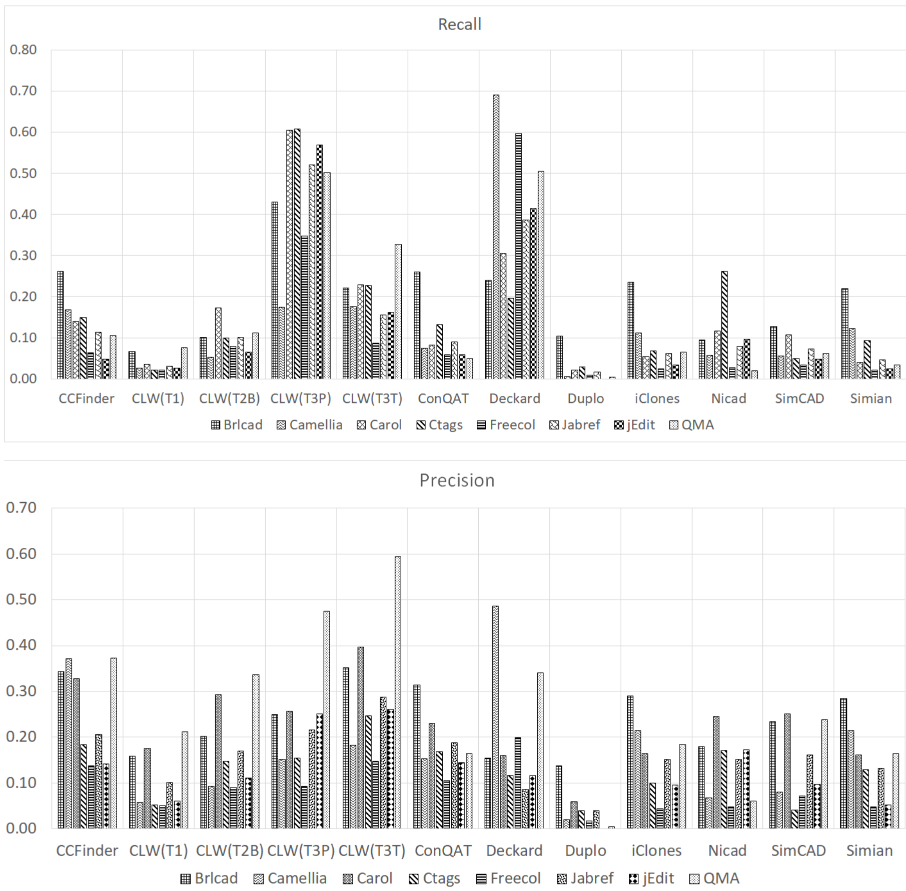

Figure 2: Average recall of different tools

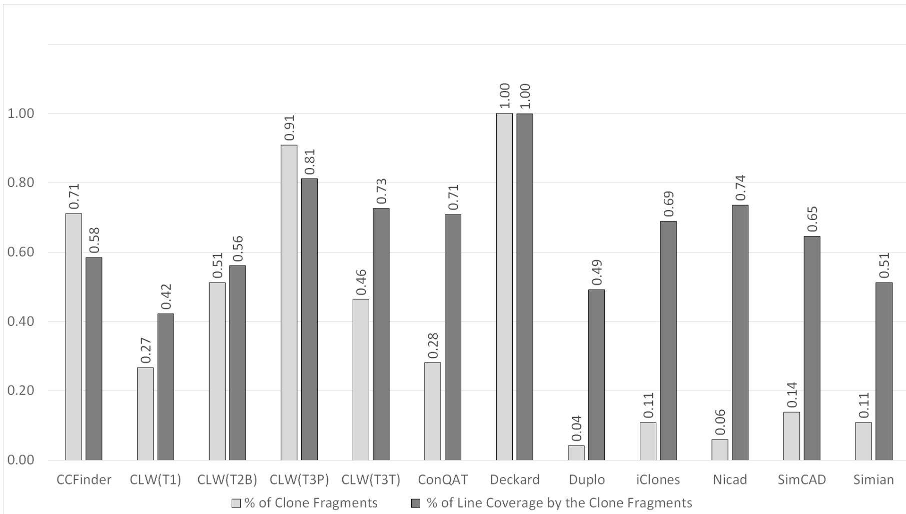

Figure 3: Comparing unique line coverage by clone fragments and number of clone fragments from different clone detectors.

in the top of the list. CLW(T3T) and CCFinder fall in the respective next position in providing the highest number of clone fragments and covering the highest number of unique lines in the source files. This scenario shows that a good clone detector can perform badly in detecting cloned co-change candidates if it does not detect enough clone fragments and does not cover enough unique lines by those clone fragments in the source file. Though, earlier study [7] suggests that NiCad is a very good clone detector, in both of these cases, it falls at the bottom of the list. Despite, NiCad performs very well in detecting clone fragments, it provides a lower number of clone fragments and also the lower number of line coverage by those clone fragments in the software systems. For that reason, while detecting the cloned co-change candidates, NiCad is showing lower F1 Score. The number of clone fragments and line coverage by those fragments seems to be an underlying factor behind the obtained comparison scenario of the clone detectors in predicting cloned co-change candidates, there can be several other factors such as overlapping of code clones and code similarity detection mechanism. We plan to investigate these factors in future.

*4.3. Answer to the **RQ3***

Do the source code processing techniques (Pattern/Token/Text-based processing) of the clone detection tools have any impact on their performance in detecting co-change candidates?

We can answer this research question by analysing our final ranking of the clone detectors in Table 6. Top two clone detectors (Rank 1 and 2) work by extracting source code patterns from the code-base. CLW(T3P) processes the source code terms by splitting into lines and then extracts code patterns. Deckard first generates vectors from the source file and then extracts a tree-like source pattern to match similarity among different source code fragments. The other five tools (Rank 3 to 7) in the rank list perform token-based source code processing and the remaining five tools perform text-based source code processing for detecting clones from the source file. From this result, we can say that text-based clone detection tools are not good to be used in detecting cloned co-change candidates during software evolution. The tools which can detect more generalized clone fragments especially pattern-based clone detectors are very good for detecting co-change candidates.

*4.4. Answer to the **RQ4***

Do clone detection tools designed for detecting different types of clones (Type 1, 2, 3) work differently in detecting cloned co-change candidates?

From the final rank list of our clone detectors, we also find the relation of detected clone types with its ability to detect cloned co-change candidates. The rank list of clone detectors in Table 6 shows that clone detecting tools such as CLW(T1), Duplo, which detects the only Type 1 clone will not perform well in detecting co-change candidates. On the other hand, tools such as CLW(T3P), CLW(T3T), Deckard, CCFinder perform very well in detecting cloned co-change candidates. The significance test results in Table 7 also show that four tools (two configurations of CloneWorks for Type-3, Deckard, and CCFinder) which perform significantly better than the other tools are also known as the clone detectors which detects Type-3 clones (Type-1, 2 also automatically included with type-3 clones). Therefore, our findings of this study suggest that we should choose those clone detectors to be used in detecting co-change candidates which detects Type-3 clones with the other Type-1 and Type-2 clone fragments.

Table 5: F1 Score of Different Tools in Detecting Cloned Co-change

Tools/SS	**Brlcad**	**Camellia**	**Carol**	**Ctags**	**Freecol**	**Jabref**	**jEdit**	**QMA**
CCFinder	0.30	0.23	0.20	0.16	0.09	0.15	0.07	0.16
CLW(T1)	0.09	0.04	0.06	0.03	0.03	0.05	0.04	0.11
CLW(T2B)	0.13	0.07	0.22	0.12	0.08	0.13	0.08	0.17
CLW(T3P)	0.32	0.16	0.36	0.25	0.15	0.30	0.35	0.49
CLW(T3T)	0.27	0.18	0.29	0.24	0.11	0.20	0.20	0.42
ConQAT	0.28	0.10	0.12	0.15	0.08	0.12	0.08	0.08
Deckard	0.19	0.57	0.21	0.15	0.30	0.14	0.18	0.41
Duplo	0.12	0.01	0.03	0.03	0.01	0.02	0.00	0.00
iClones	0.26	0.15	0.08	0.08	0.03	0.09	0.05	0.10
Nicad	0.12	0.06	0.16	0.21	0.04	0.10	0.12	0.03
SimCAD	0.17	0.07	0.15	0.04	0.05	0.10	0.06	0.10
Simian	0.25	0.16	0.06	0.11	0.03	0.07	0.03	0.06

4.5. The Wilcoxon Signed-Rank Test:

We performed The Wilcoxon Signed-Rank Test [11, 12] to verify the hypothesis that the F1 Scores of a tool which has obtained a higher rank in Table 6 are significantly different (better) than the F1 Scores of the tools which have got lower ranks. Here, F1 Scores of each tool contains eight values obtained in all the eight subject systems. For instance, let us assume that we would like to examine whether the F1 Scores obtained by CLW(T3P) are significantly better than the F1 Scores obtained by CLW(T3T). Thus, we take the sets of F1 Scores (see Table 5) from both CLW(T3P) and CLW(T3T) which will be then used to perform Wilcoxon Signed-Rank Test utilizing the SciPy library [38] available

Table 6: Ranks of the clone detectors by considering individual ranking in each of the Subject Systems

Clone Detectors	S1	S2	S3	S4	S5	S6	S7	S8	$\sum_{S1}^{S8}$	Final Rank
CLW(T3P)	1	4	1	1	2	1	1	1	12	**1**
CLW(T3T)	4	3	2	2	3	2	2	2	20	**2**
Deckard	7	1	4	6	1	4	3	3	29	**3**
CCFinder	2	2	5	4	4	3	7	5	32	**4**
CLW(T2B)	9	8	3	7	5	5	5	4	46	**5**
ConQAT	3	7	8	5	6	6	6	9	50	**6**
iClones	11	10	6	3	8	7	4	11	60	**7**
Simian	5	6	9	9	10	9	9	7	64	**8**
Nicad	8	9	7	10	7	8	8	8	65	**9**
SimCAD	6	5	11	8	11	10	11	10	72	**10**
CLW(T1)	12	11	10	11	9	11	10	6	80	**11**
Duplo	10	12	12	12	12	12	12	12	94	**12**

**** S1-S8** represents sequence of eight subject systems used in this study.*

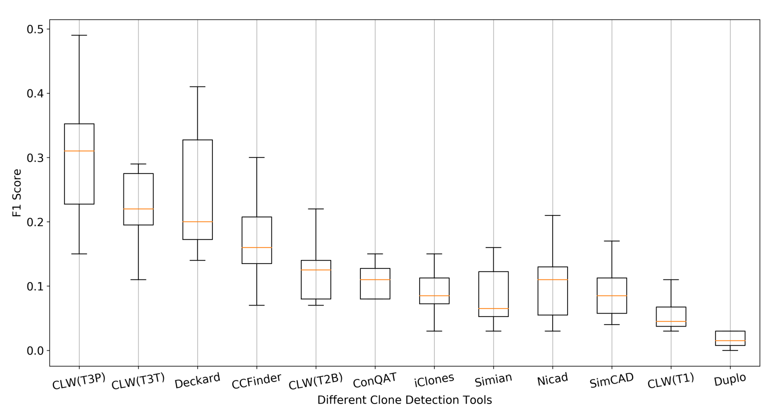

Figure 4: Comparing Distribution of F1 Scores in Different Clone Detectors

Table 7: Wilcoxon Signed Rank Test ($p<0.05$)

Tools in Investigation	**Significantly Better than Tools ($p<0.05$)**	**# of Tools**
CLW(T3P)	CLW(T3T), CCFinder, CLW(T2B), ConQAT, iClones, Simian, Nicad, SimCAD, CLW(T1), Duplo	10
CLW(T3T)	CLW(T2B), ConQAT, iClones, Simian, Nicad, SimCAD, CLW(T1), Duplo	8
Deckard	CLW(T2B), iClones, Simian, Nicad, SimCAD, CLW(T1), Duplo	7
CCFinder	ConQAT, iClones, Simian, SimCAD, CLW(T1), Duplo	6
CLW(T2B)	CLW(T1), Duplo	2
ConQAT	CLW(T1), Duplo	2
iClones	CLW(T1), Duplo	2
Simian	Duplo	1
Nicad	Duplo	1
SimCAD	CLW(T1), Duplo	2

in Python programming language. We did a significance test for each of the possible pairs from all the 12 clone detection tools in our investigation.

A summary of the significant results at $p < 0.05$ obtained from the significant test is given in Table 7. The left-most column of this table contains the tool whose significance is to be tested, and the next column contains the name of the tools, each of them provides significantly different F1 Scores compared to the tool in the investigation. The right-most column of Table 7 shows the number of clone detector whose F1 Scores are significantly different than the F1 Scores of the tool under investigation. Therefore, CLW(T3P) provides significantly different F1 Scores compared to 10 other clone detectors (excluding Deckard). The distribution of F1 Scores in Figure 4 also shows that majority of the F1 Score values of CLW(T3P) lie above all the other clone detectors' F1 Score values (except Deckard). Although some of the F1 Socre values in CLW(T3P) are above the values of Deckard, those are not enough to make the result significantly different. This scenario clearly shows that CLW(T3P) is significantly better than all the other clone detectors except Deckard. Similarly, from the following results of our significance test in Table 7 we can see that F1 Scores of CLW(T3T) are significantly better than the other eight clone detectors, F1 Scores of Deckard are significantly better than the other seven clone detectors, and F1 Scores of CCFinder are significantly better than the other six clone detectors. The following four tools (CLW(T2B), ConQAT, iClones, SimCAD) are significantly better than CLW(1) and Duplo. Simian and NiCad are significantly better than only Duplo. The overall observation of the significance test result helps to conclude that for detecting clone co-change candidates, CloneWorks Type-3 clone detection configuration can be a very good choice, Deckard and CCFinder are also good choices, but the other tools are not significantly better choices to detect co-change candidates during software evolution.

The distribution of F1 Scores in Figure 4 also demonstrates the significance in performance differences of clone detectors used in this study. The clone detectors in this figure are sorted based on the final rank list shown in Table 6 where the ranks of the tools are presented from left to right (rank 1 to 12 in Table 6). This figure shows the clone detectors which got higher ranking in Table 6 also have the higher values of F1 Scores compared to the tools which are below in the rank list. In this diagram, we can see that the F1 Scores of CloneWorks Type-3 Pattern have the distribution in most higher values, and Duplo have the distribution in the most lower values. The performance of any two tools will be significantly different from each other if they share a fewer common range of F1 Scores distribution. From the result of significance test in Table 7 we can see that Deckard is not significantly different than all the other three good clone detectors i.e. CLW(T3P), CLW(T3T), and CCFinder as they share most of the common range of values in the distribution. We can see a similar scenario for Simian and Nicad, e.g., though Simian and Nicad are above four and three other clone detectors respectively, their F1 Scores are significantly better than only Duplo. Simian, Nicad, SimCad, CLW(T1) shares most of the common values in the distribution of F1 Scores, therefore, they do not provide a significantly different result with each other.

5. Discussion

There are two primary perspectives of managing code clones: (1) clone tracking and (2) clone refactoring. Our research essentially focuses on the clone tracking perspective. The main task of a clone tracker is to suggest similar co-change candidates when a programmer attempts to change a code fragment. For suggesting co-change candidates, a clone tracker depends on a clone detector. Our research compares 12 promising clone detectors based on their capabilities in suggesting cloned co-change candidates. According to our investigation, CloneWorks (Type-3 Pattern, and Type-3 Token), Deckard, and CCFinder are the most promising tools for suggesting such co-change candidates based on the ranking we obtained in Table 6 and the result of our significance test in Table 7. Based on our overall observation, we can say that the performance of CloneWorks (Type-3 Pattern/ Token), Deckard, and CCFinder are much better compared to the other clone detection tools in detecting co-change candidates during software evolution. As the clone classes/ pairs generated by different clone detectors played an important role in our analysis, we can say that the clone detectors which can group similar clone fragments into a clone class/ pair efficiently will perform better in detecting co-change candidates during the commit operation. From our findings, we can also say that the clone detectors which detect all the clone types such as Type 1, 2, and 3 clones can also perform well in detecting co-change candidates.

In our research, we do not compare the clone detectors considering their clone detection efficiency. We rather compare the clone detection tools based on their ability in suggesting cloned co-change candidates. Such a comparison of clone detectors focusing on a particular maintenance perspective was not done previously. Suggesting co-change candidates for a target program entity is an important impact analysis [1] task during software evolution. Thus, through our research, we investigate which of the clone detectors can be useful in change impact analysis to what extent. Findings from our research can identify which clone detector(s) can be promising for change impact analysis.

6. Threats to Validity

We have investigated eight subject systems in our study. While more subject systems could generalize our findings, we selected our systems focusing on their diversity, popularity of used programming language, and availability of a considerable number of revisions. For example, our systems are of different application domains, sizes, and revision history lengths. Thus, our findings are not biased by our choice of subject systems. We believe that our findings are important from the perspectives of software maintenance.

We have investigated 12 different configurations of nine clone detectors in our study. Detection parameter settings of the clone detectors can have an impact on their comparison. However, the parameters of different clone detectors were selected considering their equivalence. Thus, we believe that we have a fair comparison among the clone detectors.

Several code fragments might change together in a commit operation. While some of these fragments can be similar to one another, and some might be dissimilar. Similar code fragments co-change (i.e., change together) for ensuring consistency of the code-base. However, dissimilar code fragments can co-change because of their underlying dependencies which could have some impact on the generalization of this research outcome. As we aim to compare the clone detection tools, we wanted to discard the dissimilar co-change candidates from our consideration. If a co-change candidate was not detected as a true positive by any of the clone detectors, we discarded the candidate. We believe that such a consideration is reasonable in our experiment aiming towards comparing clone detectors and our findings may inspire more similar research.

7. Conclusion

In this research, we make a comparison among different clone detection tools from the perspective of software maintenance. In particular, we investigate their performances in successfully suggesting (i.e., predicting) cloned co-change candidates during evolution. We used eight open-source subject systems written in C and Java for our analysis. According to our final rank list in Table 6 and summary of significance test result in Table 7, CloneWorks (Type-3, both configurations for Pattern and Token), Deckard, and CCFinder show the most promising results in most of the eight subject systems compared to the other tools. CloneWorks (Type-2), ConQAT, iClones are also showing better performance than the other remaining tools.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by a Canada First Research Excellence Fund (CFREF) grant coordinated by the Global Institute for Food Security (GIFS).

References

[1] R. S. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[2] M. Mondal, B. Roy, C. K. Roy, K. A. Schneider, Associating code clones with association rules for change impact analysis, in: Proc. SANER, 2020, p. 11pp.

[3] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, D. W. Binkley, Generalizing the analysis of evolutionary coupling for software change impact analysis, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, 2016, pp. 201–212.

[4] J. Svajlenko, C. K. Roy, Cloneworks: A fast and flexible large-scale near-miss clone detection tool, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 177–179.

[5] M. Mondal, C. K. Roy, K. A. Schneider, An empirical study on change recommendation, in: Proc. CASCON, CASCON '15, IBM Corp., Riverton, NJ, USA, 2015, pp. 141–150.

[6] M. Mondal, C. Roy, K. Schneider, Connectivity of co-changed method groups: a case study on open source systems, 2012, pp. 205–219.

[7] M. Mondal, C. K. Roy, K. A. Schneider, Prediction and ranking of co-change candidates for clones, in: Proc. MSR 2014, ACM, New York, NY, USA, 2014, pp. 32–41. doi:10.1145/2597073.2597104.

[8] M. Mondal, B. Roy, C. K. Roy, K. A. Schneider, Investigating context adaptation bugs in code clones, in: Proc. ICSME, 2019, pp. 157–168. doi:10.1109/ICSME.2019.00026.

[9] J. F. Islam, M. Mondal, C. K. Roy, K. A. Schneider, Comparing bug replication in regular and micro code clones, in: Proc. ICPC, 2019, pp. 81–92. doi:10.1109/ICPC.2019.00022.

[10] J. F. Islam, M. Mondal, C. K. Roy, A comparative study of software bugs in micro-clones and regular code clones, in: Proc. SANER, 2019, pp. 73–83.

[11] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bulletin 1 (1945) 80–83. URL: http://www.jstor.org/stable/3001968.

[12] B. Rosner, R. J. Glynn, M.-L. T. Lee, The wilcoxon signed rank test for paired comparisons of clustered data, Biometrics 62 (2006) 185–192.

[13] M. Nadim, M. Mondal, C. K. Roy, Evaluating performance of clone detection tools in detecting cloned cochange candidates, in: 2020 IEEE 14th International Workshop on Software Clones (IWSC), 2020, pp. 15–21.

[14] 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 (2009).

[15] J. Svajlenko, C. K. Roy, Evaluating modern clone detection tools, in: Proc. ICSME, 2014, pp. 321–330. doi:10.1109/ICSME.2014.54.

[16] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (2007) 577–591. doi:10.1109/TSE.2007.70725.

[17] C. Roy, J. Cordy, Scenario-based comparison of clone detection techniques, 2008, pp. 153–162. doi:10.1109/ICPC.2008.42.

[18] E. Burd, J. Bailey, Evaluating clone detection tools for use during preventative maintenance, in: Proc. SCAM, 2002, pp. 36–43. doi:10.1109/SCAM.2002.1134103.

[19] F. V. Rysselberghe, S. Demeyer, Evaluating clone detection techniques from a refactoring perspective, in: Proc. ASE, 2004, pp. 336–339.

[20] C. K. Roy, J. R. Cordy, Benchmarks for software clone detection: A tenyear retrospective, in: Proc. SANER, 2018, pp. 26–37. doi:10.1109/SANER.2018.8330194.

[21] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: 2006 13th Working Conference on Reverse Engineering, 2006, pp. 253–262. doi:10.1109/WCRE.2006.18.

[22] S. Ducasse, O. Nierstrasz, M. Rieger, On the effectiveness of clone detection by string matching: Research articles, J. Softw. Maint. Evol. 18 (2006) 37–58. doi:10.1002/smr.v18:1.

[23] G. M. K. Selim, K. C. Foo, Y. Zou, Enhancing source-based clone detection using intermediate representation, in: 2010 17th Working Conference on Reverse Engineering, 2010, pp. 227–236.

[24] T. Software, TIOBE Index — TIOBE - The Software Quality Company, 2020 (accessed July 6, 2020). URL: https://www.tiobe.com/tiobe-index/.

[25] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360), 1999, pp. 109–118.

[26] E. Juergens, F. Deissenboeck, B. Hummel, Clonedetective - a workbench for clone detection research, in: Proc. ICSE, 2009, pp. 603–606. doi:10.1109/ICSE.2009.5070566.

[27] N. Göde, R. Koschke, Incremental clone detection, in: Proc. CSMR, 2009, pp. 219–228. doi:10.1109/CSMR.2009.20.

[28] J. R. Cordy, C. K. Roy, The nicad clone detector, in: Proc. ICPC, 2011, pp. 219–220. doi:10.1109/ICPC.2011.26.

[29] M. S. Uddin, C. K. Roy, K. A. Schneider, Simcad: An extensible and faster clone detection tool for large scale software systems, in: Proc. ICPC, 2013, pp. 236–238. doi:10.1109/ICPC.2013.6613857.

[30] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (2002) 654–670.

[31] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: Proc. ICSE, 2007, pp. 96–105. doi:10.1109/ICSE.2007.30.

[32] S. Harris, Simian - Similarity Analyser — Duplicate Code Detection for the Enterprise—Overview, 2003 (accessed July 6, 2020). URL: http://www.harukizaemon.com/simian/.

[33] C. Ragkhitwetsagul, J. Krinke, D. Clark, Similarity of source code in the presence of pervasive modifications, in: Proc. SCAM, 2016, pp. 117–126. doi:10.1109/SCAM.2016.13.

[34] T. Wang, M. Harman, Y. Jia, J. Krinke, Searching for better configurations: A rigorous approach to clone evaluation, in: Proc. ESEC/FSE, ACM, New York, NY, USA, 2013, pp. 455–465.

[35] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, K. A. Schneider, An empirical study of the impacts of clones in software maintenance, in: Proc. ICPC, 2011, pp. 242–245. doi:10.1109/ICPC.2011.14.

[36] W. T. Cheung, S. Ryu, S. Kim, Development nature matters: An empirical study of code clones in javascript applications, Empirical Softw. Engg. 21 (2016) 517–564.

[37] J. Krinke, N. Gold, Y. Jia, D. Binkley, Cloning and copying between gnome projects, in: Proc. MSR, 2010, pp. 98–101. doi:10.1109/MSR.2010.5463290.

[38] P. Virtanen, R. Gommers, T. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, ... Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nature Methods 17 (2020) 261–272.

Evaluating Performance of Clone Detection Tools in Detecting Cloned Cochange Candidates

Md Nadim Manishankar Mondal Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
{mdn769, mshankar.mondal, chanchal.roy}@usask.ca

***Abstract*—Code reuse by copying and pasting from one place to another place in a codebase is a very common scenario in software development which is also one of the most typical reasons for introducing code clones. There is a huge availability of tools to detect such cloned fragments and a lot of studies have already been done for efficient clone detection. There are also several studies for evaluating those tools considering their clone detection effectiveness. Unfortunately, we find no study which compares different clone detection tools in the perspective of detecting cloned co-change candidates during software evolution. Detecting cloned co-change candidates is essential for clone tracking. In this study, we wanted to explore this dimension of code clone research. We used six promising clone detection tools to identify cloned and non-cloned co-change candidates from six C and Java-based subject systems and evaluated the performance of those clone detection tools in detecting the cloned co-change fragments. Our findings show that a good clone detector may not perform well in detecting cloned co-change candidates. The amount of unique lines covered by a clone detector and the number of detected clone fragments plays an important role in its performance. The findings of this study can enrich a new dimension of code clone research.**

***Index Terms*—Clone Detection, Cloned Co-change Candidates, Commit operation, Software Maintenance.**

I. Introduction

A large number of software tools have already been introduced for detecting cloned code fragments. Two surveys, that were done in 2009 by Roy et al. [25] and in 2013 by Rattan et al. [22] reported 75% increase in the number of clone detection tools in these four years. Roy and Cordy [23] reported the existence of about 200 tools for detecting cloned code fragments. Although a large number of clone detection tools currently exist, we found no study for comparing the performance of different tools based on their ability to be used in software maintenance activity such as predicting cloned co-change candidates during software evolution. In this study, we wanted to explore, whether a good clone detector also performs well in detecting cloned co-change fragments?

One of the common features of clone detection tools is to combine similar code fragments into a clone group or class. The code fragments in a particular clone class are expected to perform similar functionalities. If we want to make changes to a particular clone fragment in a clone class, the other fragments in the class are likely to have similar changes to ensure consistency of the codebase. Considering this assumption, we can say that all the clone fragments in a clone class have the possibility of being a cloned co-change candidate with any change of that class members. We utilize the clone classes provided by the clone detectors for these types of co-change prediction.

During software evolution, a developer makes changes in the codebase to fulfil some change requests. Those change requests could be related to each other or independent [19, 17]. Therefore, all the changes done in a single commit need not be related to each other. Some changes in a single commit may be dependent on each other and some may be independent. The related code fragments are known as the co-change candidates in literature [18]. Some of those co-change candidates may contain similar code-fragments i.e. they are clones of one another, on the other than, other types of co-change candidates may not be cloned fragments but they have a functional dependency or coupling with each other. If a developer makes changes to a target code fragment, those changes might also need to be reflected to other similar fragments in the codebase to ensure consistent evolution of the software system [20, 16]. Failing to change a co-change candidate of a target fragment can introduce bugs in the software system [10, 9]. In this study, we evaluated the performance of clone detection tools in detecting cloned co-change candidates.

We have analyzed thousands of commit operations from the evolutionary histories of six subject systems listed in Table I. While analyzing a commit operation, we identify which code fragments changed together (i.e., co-changed) in that commit. Considering each fragment as the target fragment, we try to predict the other actually co-changed fragments using each of our clone detectors. We found some change fragment which is not detected by any of the clone detectors. We excluded those change fragments from consideration during calculating the performance measures of clone detectors. An example of our detection process is demonstrated in Fig. 1. Let us assume that 21 changes, C1 to C21, occurred in the codebase of a subject system in a particular commit operation. We detect these changes using the UNIX diff operation. If we consider C1 as the target change, the other 20 changes, C2 to C21, are the actual co-change candidates (i.e., co-changed candidates) of C1. We apply different clone detectors to detect these co-change candidates for the target change C1. Let using Deckard we can detect five change fragments (C2, C6, C8, C15, C21) from those 20 fragments, similarly using Nicad we can detect four fragments (C5, C10, C16, C18). We will continue to detect co-change fragments using all the other clone detectors.

978-1-7281-6269-0/20/$31.00 © 2020 IEEE IWSC 2020, London, ON, Canada

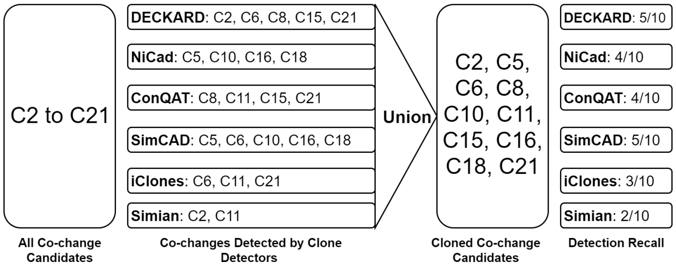

Fig. 1: Demonstrating cloned co-change detection process

After getting the results from all the clone detectors, we find 10 unique change fragments (C2, C5, C8, C15, C21, C5, C8, C10, C15, C21) out of 20 fragments by taking a union of the results of all the clone detectors. We will take those 10 unique change fragments as cloned co-change candidates and calculate the recall of each of the clone detectors based on their number of detection among those cloned co-changes. For each subject system, we finally calculated average recall, average precision, and F1 Score for each of the clone detector for predicting the actually co-changed fragments during system evolution. We then compare the clone detectors based on their F1 Score. Fig. 2 and Fig. 3 shows the bar chart of Average Recall and Average Precision drawn from our experimental results and in Table VI we have given the calculated F1 Score. According to our preliminary findings and ranking of the clone detectors (Table VIII), we can conclude that Deckard outperforms all the other five tools. The performance of ConQAT is next to Deckard. Other tools are in the following order where NiCad and SimCAD provides equal F1 Score, on the other hand, iClones and Simian are very close based on their F1 Score. We also calculated the average number of distinct lines detected as cloned lines by each of the clone detectors in all the revisions of all the subject systems and found that the clone detector which detects more distinct lines as a cloned line in the codebase also performs well in detecting cloned co-change candidates.

Based on this preliminary study, we tried to answer the following research questions:

RQ1: What is the comparison scenario of the clone detectors in predicting cloned co-change candidates?

RQ2: Why do different clone detectors perform differently in detecting cloned co-change candidates?

To the best of our knowledge, our study is the first one to compare clone detection tools considering a particular maintenance perspective (e.g., considering their capabilities in successfully suggesting cloned co-change candidates during software evolution). From an initial assumption, it is obvious that a clone detector which is good in detecting clone should also be good in detecting cloned co-change candidates. In this exploratory study, we wanted to practically verify this assumption. We selected six good clone detectors reported in several earlier studies in our investigation to verify whether they are also good at detecting co-change.

We organized this paper in the following sections: Some related works are described in Section II, our methodology is in Section III, we described the experimental result in Section IV, the discussion is in section V, Section VI explains some possible threats to validity, and we conclude our paper by mentioning future work in Section VII.

II. Related Work

There are several studies [28, 25, 2, 24] that have been focused on ranking different clone detection tools based on their performance in detecting different types of clone fragments and accuracy of those detection tools. Burd and Bailey [3] did a study for comparing the performance of three clones and two plagiarism detecting tools based on their precision and recall of the ability to detect duplicated codes in a single file or across different files. Bellon et al. [2] evaluated six clone detection tools based on eight large C and Java programs of almost 850 KLOC and made a framework for comparing different clone detection tools with the data validated by one of the authors of this. Rysselberghe and Demeyer [26] evaluated three representative clone detection techniques from a refactoring perspective where they provided comparative results in terms of portability, kinds of clone reported, scalability, number of false positive, and number of useless clone detection. Svajlenko and Roy [28] evaluated eleven modern clone detection tools using four benchmark frameworks and noted ConQAT, iClones, NiCad and SimCAD as very good tools for detecting clones of all the three types (Type-1, Type-2, Type-3). Roy et al. [25] did a qualitative comparison and evaluation of the latest clone detection approaches and tools, and made a benchmark called BigCloneBench [23] which contains eight million manually validated clone pairs in a large inter-project source dataset of more than 25,000 projects and 365 million lines of code. They categorize, relate and assess different clone detection tools based on two different points of view such as classification based on the overlapping set of attributes in the different code fragments and the scenarios how Type-1, Type-2, Type-3, and Type-4 clones created. They also elaborated the procedure of using the result of their study to select the most suitable clone detection tool or technique in the context of a specific set of areas and limitations.

There are some studies which not only proposed a clone detection mechanism but also did a comparison of their proposed technique with some existing techniques. Koschke et al. [13] provided a technique to detect clone using suffix trees in abstract syntax trees and they also made a comparison to other techniques using the Bellon benchmark for clone detectors. Ducasse et al. [6] and Selim et al. [27] also utilized Bellon's framework for measuring the performance of their proposed clone detection tools based on string comparison and intermediate source transformation respectively. Selim et al. [27] showed that their tool is capable of detecting Type-3 clones and their technique is better than the source-based clone detectors based on the value of recall through a slight drop in the precision using Bellon's corpus where clone group is not complete. Compared to the standalone string and token-based clone detectors, their technique showed a little higher precision.

All the studies which compared different clone detectors have been focused on the precision, recall, computational complexity, and memory used or detecting a specific type of clone fragments such as Type-1, Type-2, Type-3, or Type-4 during the detection approach of duplicated code in a codebase. Our study to compare clone detectors is completely different from the previous comparisons. We do not want to compare clone detection tools based on the capability to detect clones. Our point of interest is to detect co-change candidates during the software commit operations. Mondal et al. [18] did a study to predict and rank the co-change candidates by analyzing evolutionary coupling from previously done change history using generated clone fragments by NiCad but they did not consider the result of other clone detection tools and also did not show any comparative study among different clone detection tools in doing such prediction and rank of co-change candidates. We found no study which compared different clone detectors in this perspective of software maintenance. In this research, we have analyzed the performance of six clone detection tools based on their capabilities in finding co-change candidates during software evolution using their generated clone result. We have taken four clone detection tools (ConQAT, iClones, NiCad, and SimCAD) suggested as good tools in the study of Svajlenko and Roy [28] and two other tools, one of them is text similarity-based (Simian) and the other is tree similarity-based (Deckard) for evaluating their performance in our study. According to our knowledge, this is the first such investigation of performance with clone detection tools.

III. Methodology

We have used six subject systems listed in Table I and six clone detection tools (Table III) for our analysis. Our analysis aims to rank these clone detection tools based on their performance in successfully suggesting actual co-change candidates (ACC) during the software evolution. Before starting our main analysis, we have to resolve some issues and we have taken the following considerations in this regard.

Selection of subject systems: To select subject systems for this study, we considered both the popularity of programming language and availability of a considerable amount of revisions. According to the TIOBE Programming Community index [29] (an indicator of the popularity of programming languages), Java is dominating the list of popular programming languages for more than the last ten years and C is the second most popular programming language within this period. Considering this fact, we wanted to select subject systems written in these two programming languages. Our other consideration was the availability of a considerable amount of revisions of each of the systems. Based on both of the considerations, we have chosen the subject systems listed in Table I.

TABLE I: Subject Systems

Systems	Lang.	Domains	LOC	Rev.
Brlcad	C	Computer Aided Design	39,309	2115
Carol	Java	Game	25,091	1700
Ctags	C	Code Def. Generator	33,270	774
Freecol	Java	Game	91,626	1950
Jabref	Java	Reference Manager	45,515	1545
jEdit	Java	Text Editor	191,804	4000

TABLE II: Summary of Data Processed

Category of Information	Brlcad	Carol	Ctags	Freecol	Jabref	JEdit
Number of revisions Processed	2113	1700	774	1001	1540	215
Number of revisions experiencing change	660	454	447	836	860	145
Number of revisions experiencing more than one change	553	430	330	833	755	145

Selection of clone detectors: In this research, we wanted to examine those clone detection tools which are good in detecting all types of clones. To select such tools, we considered some related studies. We have taken ConQAT [12], iClones [7], NiCad [5], and SimCAD [30] as they have been reported as very good tools for detecting all type of clones in the study of Svajlenko and Roy [28]. Besides these, Deckard [11], iClones and NiCad are often considered as common examples of modern clone detectors that support Type-3 clone detection. The reason of taking Simian [8] in our analysis was its ability to find duplicated code by line-by-line textual comparison supporting identifier renaming with a fast detection speed on the large repository and extensive use in several clone studies [21, 31, 15, 4, 14]. NiCad, SimCAD, and Simian are textual similarity-based clone detection tools. Deckard works using tree comparison technique, on the other hand, ConQAT and iClones are token-based clone detection tools.

Determining if the extracted co-changes are related to each other or not: Even though we have extracted all the changes between two adjacent revisions (i.e., revision n and n+1), we cannot guarantee that all the changes are actually co-change candidates of each other. There might be some changes which do not depend on any other changes i.e. they may change independently. The inclusion of such dissimilar changes into our calculation can drop the detection accuracy of clone detectors. To minimize such drops, we excluded those co-changes which are not detected by any of the six clone detectors. As none of the clone detectors in our study considers them as co-change candidates, we considered those changes as dissimilar or independent changes.

Ensuring if the configuration parameters of all the clone detection tools identical with each other or not: As we wanted to compare different clone detectors based on their capability of successfully suggesting co-change candidates, it was important to configure them identically during detecting clones from our subject systems. Wang et al. [31] introduced confounding configuration choice problem where the configuration of different tools during clone detection may play a vital role and the result may be best or worst depending on the configuration. Our configuration of different tools is shown in the Table III. We have used similar configurations for each of the tools for obtaining a consistent result. We have taken configuration values similar to Svajlenko and Roy [28] which they conducted to compare different clone detectors based on their efficiency in detecting cloned fragments. ConQAT, NiCad, and Deckard require similarity parameter which we have taken for ConQAT 70% (gapratio=0.3), for NiCad 70% (threshold=0.3) and Deckard 85%. We analyzed the result obtained from Deckard with the similarity score 70% (as of ConQAT and NiCad) and found that with this similarity score Deckard generates a lot of unwanted clones in the result where most of them are duplicated and showing a lot of fragments as a clone to itself several times. We also tried some other percentage values such as 75%, and 80% but the detected result of Deckard becomes much desirable when we set it to 85%. Svajlenko and Roy [28] also used 85% similarity while running Deckard for Mutation Framework. As we wanted to compare different clone detectors based on their capability of successfully suggesting co-change candidates, it was very important to configure them identically during detecting clones from our subject systems.

The overall approach: Our overall processing is performed in some distinct steps. Initially, we downloaded all the source files of all the revisions of all the subject systems from their respective SVN repositories. We then applied **diff** operation between each file of a revision with the respective file in the next revision and extracted the change information such as Name of the File which is changed, the Line where the respective change begins, the Line where the change is ended from the output of **diff**. We did the change extraction for each of the revision (excluding the last one) of all the subject systems. After detecting all the changes, we started the clone detection on all the revisions of all the subject systems using all the clone detection tools. We started our main analysis to find the accuracy of each of the clone detection tools after having the result of all the clone detectors and change information from all the revisions.

The mechanism of calculating accuracy is demonstrated in our introduction using Fig. 1. Suppose, we are examining a particular commit operation. The number of fragments that were changed in this commit operation is n. Now, let us consider one of these n fragments as the target fragment. Then the other $n-1$ fragments are the actually co-changed candidates for the target fragment. We excluded the non-cloned co-change candidates using the approach described in the introduction. After this exclusion, we get the **Actually Cloned Co-change** (ACC) for each of the target fragments.

Let us assume that the target change fragment intersects a particular clone fragment from a particular clone class. The other fragments in that clone class are considered as the **Predicted Cloned Co-change** (PCC) candidates. We now determine how many of these PCC intersect with the ACC to obtain the number of detected cloned co-change candidates by the clone detector.

These counts of predicted and actually co-changed candidates are considered as the **true positives** to calculate Recall, Precision, and F1 Score. We calculate these using the following equations (Eq. 1, 2, and 3).

$$Recall = \frac{|PCC \cap ACC|}{|ACC|} \tag{1}$$

$$Precision = \frac{|PCC \cap ACC|}{|PCC|} \tag{2}$$

$$F1\ Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

We repeat the calculating process of Recall and Precision for all the changes in each of the subjects systems with the detected clone fragments generated by all the clone detection tools. We then calculate F1 Score of the clone detectors for each of the subject systems by taking the average values of Recall and Precision which is reported in Table VI. We reported both the ranking of the tools in each of the subject systems in Table VII and overall ranking considering all the subject systems in Table VIII. To calculate the overall ranking of the tools we took weighted average (Total number of Changes is the corresponding weighting factor in each subject system) of the performance measures (Precision, Recall, F1 Score) in each individual subject systems.

TABLE III: CONFIGURATION OF PARTICIPATING CLONE DETECTION TOOLS

Tools	Configuration for Clone Detection
ConQAT	block clones, clone min-length=5, gap ratio=0.3
Deckard	min. size: 30 tokens, 5 token stride, min. 85% similarity
iClones	minimum block: 30, minimum clone: 50, All Transformation
NiCad	block clones, blind renaming, max. threshold=0.3, minimum lines=5, maximum lines=2500
SimCAD	block clones, Source Transformation= generous
Simian	min. size: 5 lines, normalize literals/identifiers

TABLE IV: PERCENT OF CLONED CO-CHANGE

Subject Systems	Total Number of Changes	Average Percentage of Cloned co-change (%)
Brlcad	2103	14
Carol	3299	10
Ctags	533	17
Freecol	7514	12
Jabref	6011	8
jEdit	3603	9

IV. EXPERIMENTAL RESULT

In this section, we will answer the research questions based on our overall analysis and obtained results by the processing of each of the six subject systems using all the six clone detection tools.

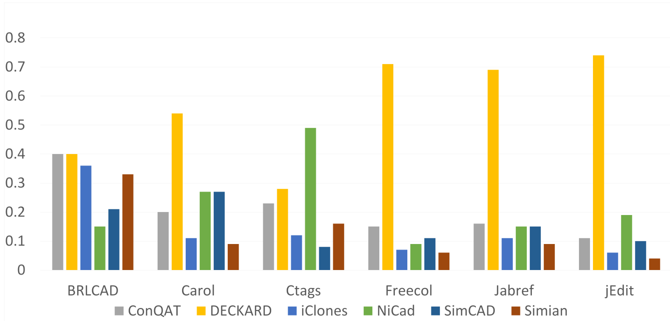

Fig. 2: Average recall of different tools

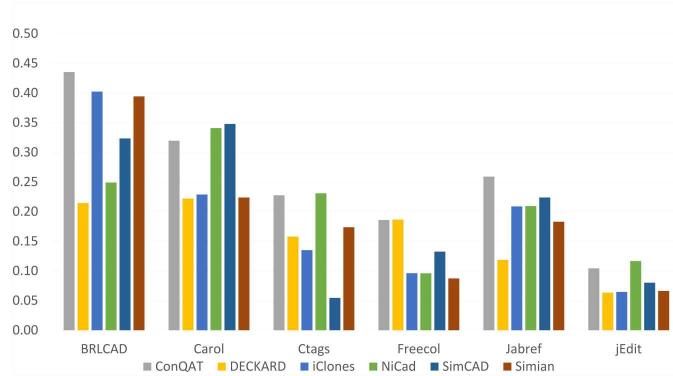

Fig. 3: Average precision of different tools

A. Answer to the ***RQ 1***

What is the comparison scenario of the clone detectors in predicting cloned co-change candidates?

The key experimental results are in Fig. 2, Fig. 3, Table IV, Table VI, Table VII and Table VIII where Fig. 2 and Fig. 3 shows the average Recall and average Precision of each of the clone detection tools. Table IV shows the percent of cloned co-change candidates we found from the six subject systems using the six clone detection tools. We found the highest and lowest percentage of cloned co-change candidates from Ctags and Jabref respectively. Table VI shows the F1 Score of each of the clone detectors in each of the subject systems. The F1 Score is calculated using Equation (3). Our experimental results concluded in Table VII which shows that Deckard and ConQat equally show better performance than the other tools in most of the subject systems. The summary of the results in the Table VII shows that among the subject systems, ConQAT is the best in Brlcad and Jabref and second-best in the other two Ctags and Freecol, on the other hand, Deckard is the best in Carol and Freecol and second-best in the other two Jabref and JEdit. Similarly, NiCad shows the highest accuracy in the result of Ctags and JEdit, second highest in the result of Carol, but not much considerable in all the other subject systems. Performance of SimCad, iClones, and Simian in these criteria is not remarkable. We also calculated overall performance in Table VIII where we can see that the F1 Score of Deckard is highest than the other tools. ConQAT, NiCad, SimCAD, iClones, and Simian are in the following order considering the weighted average of F1 Score.

As our analysis was based on the clone class provided by the clone detection tools, we found that the efficiency of clone detection tools in suggesting cloned co-change candidates is mostly dependent on its effectiveness in making clone class. The tool which groups functionally similar clone fragments into a clone class effectively can perform well in successfully suggesting cloned co-change candidate(s). Different values of the accuracy of different clone detectors indicate the difference in their efficiency in this research domain.

B. Answer to the ***RQ 2***

Why do different clone detectors perform differently in detecting cloned co-change candidates?

From the answer of our **RQ 1**, we found a difference in performance for different clone detection tools in suggesting cloned co-change candidates. We found a good clone detector may not be good at detecting cloned co-change candidates. This motivates us to find out the reason to answer this research question.

We investigated the number of clone fragments and the number of distinct lines covered by those clone fragments by all the six clone detectors from all the revisions of all the subject systems. Table V shows the weighted average of those counts for each of the clone detection tools. Considering both, the weighted average of the number of clone fragments and the weighted average of the number of lines covered by those clone fragments from all the revisions of all the subject systems, if we order the clone detectors from the highest to the lowest, we find Deckard and ConQAT in the top of the list. Though, earlier study [18] suggests that NiCad is a very good clone detector, in both of these cases, it falls at the bottom of the list. Despite, NiCad performs very well in detecting clone fragments, it provides a lower number of clone fragment and also the lower number of line coverage by those clone fragments in the software systems. For that reason, while detecting the cloned co-change candidates, NiCad is showing lower F1 Score. The number of clone fragments and line coverage by those fragments seems to be an underlying factor behind the obtained comparison scenario of the clone detectors in predicting cloned co-change candidates, there can be several other factors such as overlapping of code clones and code similarity detection mechanism. We plan to investigate these factors in future.

TABLE V: SUMMARY OF DETECTED CLONE RESULTS (WEIGHTED AVERAGE)

Tools	Deckard	ConQAT	SimCAD	iClones	Simian	NiCad
#CF	5792	1747	838	728	635	401
# LCF	15276	13471	13433	11605	11239	9875

#CF: Number of Clone Fragments in Each Revision
#LCF: Number of Unique Lines Covered by Clone Fragments in Each Revision

TABLE VI: F1 SCORE OF DIFFERENT TOOLS IN DETECTING CLONED CO-CHANGE

Subject Systems	Total Number of Changes	Total Number of Cloned Cochange	F1 Score in Detecting Cloned Co-change					
			ConQAT	Deckard	iClones	NiCad	SimCAD	Simian
Brlcad	2103	13821	0.42	0.28	0.38	0.19	0.25	0.36
Carol	3299	69454	0.25	0.31	0.15	0.30	0.30	0.13
Ctags	533	1963	0.23	0.20	0.13	0.31	0.06	0.17
Freecol	7514	246083	0.17	0.30	0.08	0.09	0.12	0.07
Jabref	6011	79417	0.20	0.20	0.14	0.17	0.18	0.12
jEdit	3603	160689	0.11	0.12	0.06	0.14	0.09	0.05

TABLE VII: RANKS OF CLONE DETECTORS BY F1 SCORE IN EACH OF THE SUBJECT SYSTEMS

Tools	BRL-CAD	Carol	Ctags	Freecol	Jabref	JEdit
ConQAT	**1**	4	**2**	**2**	**1**	3
Deckard	4	**1**	3	**1**	**2**	**2**
iClones	**2**	5	5	5	5	5
NiCad	6	**2**	**1**	4	4	**1**
SimCAD	5	3	6	3	3	4
Simian	3	6	4	6	6	6

** Tools are listed in alphabetic order in the left-most column.*
** The numbers under each subject system represent the ranks of the tools for that system.*

TABLE VIII: FINAL RANK OF CLONE DETECTORS CONSIDERING ALL THE SUBJECT SYSTEMS

Clone Detectors	Weighted Average of Precision (p)	Weighted Average of Recall (r)	F1 Score 2pr/(p+r)	Final Rank
Deckard	0.16	0.65	0.25	**1**
ConQAT	0.23	0.18	0.20	**2**
NiCad	0.18	0.16	0.17	**3**
SimCAD	0.19	0.15	0.17	**4**
iClones	0.17	0.11	0.13	**5**
Simian	0.16	0.10	0.12	**6**

V. DISCUSSION

There are two primary perspectives of managing code clones: (1) clone tracking and (2) clone refactoring. Our research essentially focuses on the clone tracking perspective. The main task of a clone tracker is to suggest similar co-change candidates when a programmer attempts to change a code fragment. For suggesting co-change candidates, a clone tracker depends on a clone detector. Our research compares six promising clone detectors based on their capabilities in suggesting cloned co-change candidates. According to our investigation, Deckard and ConQAT are the most promising tools for suggesting such co-change candidates. NiCad and SimCAD are also very good options according to our final ranking demonstrated in Table VIII. Based on our overall observation, we can say that the performance of Deckard is much better compared to the other clone detection tools in detecting co-change candidates during software evolution. As the clone classes generated by different clone detectors played an important role in our analysis, we can say that the clone detectors which can group similar clone fragments into a class efficiently will perform better in detecting co-change candidates during the commit operation. Therefore, from this observation, we can conclude that the performance of Deckard, ConQAT, and NiCad is better compared to the other clone detectors in grouping similar clone fragments into a clone class.

When a particular code fragment is changed, we apply the clone detectors to predict which other similar code fragments might also need to be co-changed. However, some dissimilar fragments might also be changed together with the particular fragment. As we are applying only clone detectors, we cannot consider those dissimilar co-change candidates in our research.

In our research, we do not compare the clone detectors considering their clone detection efficiency. We rather compare the clone detection tools based on their ability in suggesting cloned co-change candidates. Such a comparison of clone detectors focusing on a particular maintenance perspective was not done previously. Suggesting co-change candidates for a target program entity is an important impact analysis [1] task during software evolution. Thus, through our research, we investigate which of the clone detectors can be useful in change impact analysis to what extent. Findings from our research can identify which clone detector(s) can be promising for change impact analysis.

VI. THREATS TO VALIDITY

We have investigated six subject systems in our study. While more subject systems could generalize our findings, we selected our systems focusing on their diversity, popularity of used programming language, and availability of a considerable number of revisions. For example, our systems are of different application domains, sizes, and revision history lengths. Thus, our findings are not biased by our choice of subject systems. We believe that our findings are important from the perspectives of software maintenance.

We have investigated six clone detectors in our study. Detection parameter settings of the clone detectors can have an impact on their comparison. However, the parameters of different clone detectors were selected considering their equivalence. Thus, we believe that we have a fair comparison among the clone detectors.

Several code fragments might change together in a commit operation. While some of these fragments can be similar to one another, and some might be dissimilar. Similar code fragments co-change (i.e., change together) for ensuring consistency of the codebase. However, dissimilar code fragments can co-change because of their underlying dependencies which could have some impact on the generalization of this research outcome. As we aim to compare the clone detection tools, we wanted to discard the dissimilar co-change candidates from our consideration. If a co-change candidate was not detected as a true positive by any of the clone detectors, we discarded the candidate. We believe that such a consideration is reasonable in our experiment aiming towards comparing clone detectors and our findings may inspire more similar research.

VII. Conclusion and Future Works

In this research, we make a comparison among different clone detection tools from the perspective of software maintenance. In particular, we investigate their performances in successfully suggesting (i.e., predicting) cloned co-change candidates during evolution. We used six open source subject systems written in C and Java for our analysis. According to our findings (Table VII & VIII) on thousands of revisions of these systems, Deckard and ConQAT show the most promising results in four (in two best, and the other two second-best) out of the six subject systems compared to the other tools. NiCad also shows better performance in three (in two best, and the other second-best) but it does not show good enough result in the other three tools. Although we have figured some reasons of the better performance of Deckard, ConQat, and NiCad in the Discussion section of our study, we planned to extend this research by analyzing the clone detection mechanism of the clone detectors to find out some other reasons for their performance. We also want to investigate the impact of different similarity score of different clone detectors in finding co-change candidates in our future studies. Besides this, we want to include some other clone detection tools of different detection mechanism (i.e., tree/ token/ text-based) and subject systems written in some different programming languages (i.e. C/ C++, C#, Python) for extending our research.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by a Canada First Research Excellence Fund (CFREF) grant coordinated by the Global Institute for Food Security (GIFS).

References

[1] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818673842.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sep. 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70725.

[3] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proc. SCAM*, pages 36–43, Oct 2002. doi: 10.1109/SCAM.2002.1134103.

[4] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in javascript applications. *Empirical Softw. Engg.*, 21(2):517–564, April 2016.

[5] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proc. ICPC*, pages 219–220, June 2011. doi: 10.1109/ICPC.2011.26.

[6] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching: Research articles. *J. Softw. Maint. Evol.*, 18(1):37–58, January 2006. ISSN 1532-060X. doi: 10.1002/smr.v18:1.

[7] N. Göde and R. Koschke. Incremental clone detection. In *Proc. CSMR*, pages 219–228, March 2009. doi: 10.1109/CSMR.2009.20.

[8] Simon Harris. *Simian - Similarity Analyser — Duplicate Code Detection for the Enterprise — Overview*. URL `http://www.harukizaemon.com/simian/`.

[9] J. F. Islam, M. Mondal, and C. K. Roy. A comparative study of software bugs in micro-clones and regular code clones. In *Proc. SANER*, pages 73–83, Feb 2019.

[10] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider. Comparing bug replication in regular and micro code clones. In *Proc. ICPC*, pages 81–92, May 2019. doi: 10.1109/ICPC.2019.00022.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. ICSE*, pages 96–105, May 2007. doi: 10.1109/ICSE.2007.30.

[12] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *Proc. ICSE*, pages 603–606, May 2009. doi: 10.1109/ICSE.2009.5070566.

[13] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262, Oct 2006. doi: 10.1109/WCRE.2006.18.

[14] J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between gnome projects. In *Proc. MSR*, pages 98–101, May 2010. doi: 10.1109/MSR.2010.5463290.

[15] 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 *Proc. ICPC*, pages 242–245, June 2011. doi: 10.1109/ICPC.2011.14.

[16] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Investigating context adaptation bugs in code clones. In *Proc. ICSME*, pages 157–168, Sep. 2019. doi: 10.1109/ICSME.2019.00026.

[17] Manishankar Mondal, Chanchal Roy, and Kevin Schneider. Connectivity of co-changed method groups: a case study on open source systems. pages 205–219, 11 2012.

[18] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Prediction and ranking of co-change candidates for clones. In *Proc. MSR 2014*, pages 32–41, 2014. ISBN 978-1-4503-2863-0.

[19] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on change recommendation. In *Proc. CASCON*, CASCON '15, pages 141–150, Riverton, NJ, USA, 2015. IBM Corp.

[20] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Associating code clones with association rules for change impact analysis. In *Proc. SANER*, page 11pp, 2020.

[21] C. Ragkhitwetsagul, J. Krinke, and D. Clark. Similarity of source code in the presence of pervasive modifications. In *Proc. SCAM*, pages 117–126, Oct 2016. doi: 10.1109/SCAM.2016.13.

[22] Dhavleesh Rattan, Rajesh Kumar Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information Software Technology*, (7):1165–1199.

[23] C. K. Roy and J. R. Cordy. Benchmarks for software clone detection: A ten-year retrospective. In *Proc. SANER*, pages 26–37, March 2018. doi: 10.1109/SANER.2018.8330194.

[24] Chanchal Roy and J.R. Cordy. Scenario-based comparison of clone detection techniques. pages 153–162, 07 2008. ISBN 978-0-7695-3176-2. doi: 10.1109/ICPC.2008.42.

[25] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *SCIENCE OF COMPUTER PROGRAMMING*, 2009.

[26] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proc. ASE*, pages 336–339, Sep. 2004.

[27] G. M. K. Selim, K. C. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *2010 17th Working Conference on Reverse Engineering*, pages 227–236, Oct 2010.

[28] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *Proc. ICSME*, pages 321–330, Sept 2014. doi: 10.1109/ICSME.2014.54.

[29] TIOBE Software. Tiobe index — tiobe - the software quality company, 2019. URL `https://www.tiobe.com/tiobe-index/`. [Online; accessed 01-April-2019].

[30] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Proc. ICPC*, pages 236–238, May 2013. doi: 10.1109/ICPC.2013.6613857.

[31] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proc. ESEC/FSE*, pages 455–465, New York, NY, USA, 2013. ACM.