Evaluating the performance of clone detection tools in detecting cloned co-change candidates

Md Nadim *, Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider

Department of Computer Science, University of Saskatchewan, Saskatoon, Canada

ARTICLE INFO

ABSTRACT

Co-change candidates are the group of code fragments that require a change if any of these fragments experience a modification in a commit operation during software evolution. The cloned co-change candidates are a subset of the co-change candidates, and the members in this subset are clones of one another. The cloned co-change candidates are usually created by reusing existing code fragments in a software system. Detecting cloned co-change candidates is essential for clone-tracking, and studies have shown that we can use clone detection tools to find cloned co-change candidates. However, although several studies evaluate clone detection tools for their accuracy in detecting cloned fragments, we found no study that evaluates clone detection tools for detecting cloned co-change candidates. In this study, we explore the dimension of code clone research for detecting cloned co-change candidates. We compare the performance of 12 different configurations of nine promising clone detection tools in identifying cloned co-change candidates from eight open-source C and Java-based subject systems of various sizes and application domains. A ranked list and analysis of the results provides valuable insights and guidelines into selecting and configuring a clone detection tool for identifying co-change candidates and leads to a new dimension of code clone research into change impact analysis.

1. Introduction

Version control systems maintain a history of changes required to keep a software system updated with the changing requirements in its life cycle (Swierstra and Löh, 2014; AlSanad and Chikh, 2015). Software developers make changes to a software system by applying commit operations through its version control system, and the commits may be related to each other or maybe independent (Mondal et al., 2015, 2012). Each of such commits may perform one or more changes in the source code of the software system based on the available change requests. A set of change requests may be for different purposes, such as addressing evolving requirements or fixing bugs (e.g., problems or issues). A single commit operation may contain both related and independent changes. Related changes are known as co-change candidates in the literature (Mondal et al., 2014), which represents a group of changes. Suppose any code fragment of a co-change group is experiencing any update. In that case, all the other code fragments in that group might need to be updated to ensure the consistent evolution of a software system (Mondal et al., 2020; Mondal et al., 2019). When a code fragment

that should co-change with its target fragment is missed or not identified, it may induce bugs or inconsistencies in a software system (Islam et al., 2019; Islam et al., 2019).

We divide the source code fragments from a group of co-change candidates into two categories. First, those fragments that are clones of one another make a group named Cloned Co-change Candidates (CCC). Second, all the other code fragments are dissimilar co-change candidates (DCC), which might be distinct enough but still need to co-change because they are functionally dependent or coupled. Therefore, both the cloned (CCC) and dissimilar (DCC) co-change candidates are the subset of the group of co-change candidates. We identify the Cloned Co-change Candidates (CCC) using different clone detection tools (listed in Table 2) by utilizing clone groups or clone classes in the detected result sets. In this study, we evaluate the performance of clone detection tools based on detecting these Cloned Co-change Candidates (CCC).

Finding the co-change candidates (both CCC and DCC) of a target code fragment is also known as change impact analysis (Arnold, 1996) in the literature. Mondal et al. (2020) investigated whether a clone detection tool can enhance the performance of an evolutionary coupling based tool in finding a change impact set or co-change candidates. Their investigation used Nicad to detect both the regular clones and micro-clones and they found that using clone information significantly enhances the performance of Rolfsnes et al. (2016), an association

* Corresponding author.
E-mail addresses: mdn769@usask.ca (M. Nadim),
mshankar.mondal@usask.ca (M. Mondal), chanchal.roy@usask.ca (C.K. Roy),
kevin.schneider@usask.ca (K.A. Schneider).

rule mining based change impact analysis tool. Since they only used Nicad, we wanted to compare some other promising clone detection tools to find out whether these tools perform better for detecting cloned co-change candidates. As well, Nayrolles and Hamou-Lhadj (2018) used the Nicad clone detection tool to recommend qualitative fixes to developers on how to fix risky commits (commits that create inconsistencies in the system) for 12 Ubisoft systems. They first identified risky commits using a Random Forest Classifier (Breiman, 2001) based detection model and then used the Nicad clone detection tool to find similar commits with a fix already available in the history of the software system. Then they recommended the best-selected fixes to the software developer for fixing the identified risky commit. Their study showed that at least one Ubisoft software developer accepted 66.7% of their recommended fixes. Although their study focused on one specific commercial software system and its developers, we believe clone detectors could also contribute to finding similar buggy commits and their fixes in other commercial and open-source software systems. These studies motivate us to compare 12 clone detection techniques and the findings of our study suggests important guidelines for selecting clone detectors for conducting change impact analysis. Our study's outcomes will help with the successful integration of the best performing clone detection tools with change impact analysis tools to identify risky commits and possible fixes during commit operations.

The comparative performance of different clone detection tools in identifying cloned co-change candidates has yet to be investigated. Detecting cloned co-change candidates is useful during software maintenance to help ensure a system is changed consistently. In addition to determining whether a clone detection tool performs well in detecting clone fragments, it would be useful to know if it performs well in detecting cloned co-change candidates.

Clone detectors combine similar code fragments that meet certain similarity thresholds (e.g., 70% similarity) into clone groups. A clone group may contain two or more similar code fragments known as clone pairs (if the group contains exactly two similar code fragments) or clone classes (if the group contains two or more similar code fragments). Since code fragments in a clone group are similar, the clones also have similar functionality, which implies that if we want to change any of the code fragments in a clone group, the other fragments in that group may also require a similar change (co-change) to maintain consistent behavior in the software system. This assumption leads us to the possibility that all clone class members could be cloned co-change candidates of each other, and whenever any one of those fragments are modified, the developer might make a similar modification to all the other fragments of that class. We use clone classes and pairs extracted from subject systems using different clone detection tools to predict cloned co-change candidates.

We evaluated four different configurations of CloneWorks (Svajlenko and Roy, 2017) and eight additional clone detectors in our investigation. Each configuration of CloneWorks implements a unique mechanism for processing software system source files to detect clone fragments, and so we consider the four different configurations to be different tools. Therefore, we evaluated 12 separate implementations of clone detection tools. We apply these tools to thousands of commit operations from the evolutionary histories of eight open-source software systems that vary in source code size and are from different application domains. The clone detection tools and configurations we used in this study are shown in Table 2 and the subject software systems are reported in Table 1.

Based on this study, we answered the following research questions:

RQ1: How can we compare different clone detection tools based on the performance in detecting cloned co-change candidates?

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

RQ2: What are the deciding factors for the performance variance of different clone detectors in detecting cloned co-change candidates?

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 cloned 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, there is no other study comparing clone detection tools considering a particular maintenance perspective (such as their performance in detecting cloned co-change candidates during software evolution). One can assume that a clone detector, which is good in detecting cloned fragments (Roy et al., 2009; Svajlenko and Roy, 2014; Bellon et al., 2007; Roy and Cordy, 2008), might also be good in detecting cloned co-change candidates. We wanted to verify this assumption in this exploratory study. We selected 12 implementations of clone detectors detecting different types of clone fragments to evaluate their performance in detecting cloned co-change candidates. Our investigation and analysis found that the clone detectors that detect Type-3 clones and perform pattern-based source code processing are significantly better in detecting cloned co-change candidates. Our investigation also shows that tools that provide more clone fragments in their clone detection result and cover more source code lines in the software codebase are more suitable for detecting cloned co-change candidates.

We organized this paper in the following sections: we describe related work in Section 2, our methodology is in Section 3, we describe the experimental result in Section 4, the discussion of the research results is in Section 5, Section 6 explains some possible threats to validity, and we conclude our paper with some future directions of this study in Section 7.

This paper is a significant extension of our previous work (Nadim et al., 2020) on detecting cloned co-change candidates using different clone detectors. Our previous work answered two research questions by analyzing six clone detectors on six open-source software systems. Our earlier study's two research questions showed that even though a tool that is good in detecting clone fragments from software systems may not be useful in detecting cloned co-change candidates. The tools that detect a large number of clone fragments and cover more unique lines in the source files are found suitable 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 cloned co-change candidates. We have also increased the previous study's generalizability 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), totaling eight subject systems and 12 clone detector executions. Therefore, our implementation has been upgraded from 6X6 to 12X8 (Clone

Table 2
Clone detection tools and configurations.

Tool	Configuration
CCFinder	min. size: 50 tokens, min. token types: 12
CloneWorksL Type-1(CLW-T1)	termsplit = token, termproc = Joiner
CloneWorks Type-2, Blind renaming (CLW-T2B)	cfproc = rename-blind, cfproc = abstract literal, termsplit = token, termproc = Joiner
CloneWorks Type-3, Pattern (CLW-T3P)	cfproc = rename-blind, cfproc = abstract literal, termsplit = line
CloneWorks Type-3, Token (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
iCones	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

detector X Subject Systems) in the study's current version. In this study, we have shown that the performance of clone detection tools in detecting cloned co-change fragments depends on some specific factors of clone detectors such as (i) the number of discovered clone fragments, (ii) the number of unique lines covered by those clone fragments, (iii) source file processing techniques, and (iv) type of detected clones. All the source code files, datasets, and processed results related to this study are publicly available (Nadim et al., 2021) for other researchers and practitioners to facilitate the continuation and reproducibility of this study.

2. Related work

Several studies (Roy et al., 2009; Svajlenko and Roy, 2014; Bellon et al., 2007; Roy and Cordy, 2008) have focused on ranking different clones detection tools based on their performance and accuracy in detecting different types of clone fragments. Burd and Bailey (2002) has compared three clones and two plagiarism detecting tools based on their performance in detecting cloned code in a single file or across different files. Bellon et al. (2007) made a framework for comparing the performance of different clone detection tools from eight large C and Java programs having the size of source code almost 850 KLOC. One of the authors of this study also manually validated the dataset used in this study. Rysselberghe and Demeyer (2004) compared three representative clone detection techniques from a refactoring perspective. Their criteria for comparison were the portability, kinds of clone reported, scalability, number of false positive, and number of useless clone detection from the results of those clone detection techniques. Svajlenko and Roy (2014) reported ConQAT, iClones, NiCad, and SimCAD as excellent tools for detecting clones of all the three types (Type-1, Type-2, Type-3) based on their evaluation of eleven modern clone detection tools using four benchmark frameworks. Roy et al. (2009) did a qualitative comparison and evaluation of the latest clone detection approaches and tools, and made a benchmark called BigCloneBench (Roy and Cordy, 2018) BigCloneBench included eight million manually verified clone pairs in a large inter-project source code dataset where the number of projects is larger than 25,000 and lines of code are above 365 million. They classify, compare, and evaluate different clone detectors based on the following point of view, (i) how the set of attributes in the different code fragments are overlapped, and (ii) what are the scenarios of creating Type-1, Type-2, Type-3, and Type-4 clones. They also explained the procedure of using the result of their study for selecting the appropriate clone detectors in the context of particular application areas and restrictions.

Besides proposing new clone detection mechanisms, some studies also compared their tools with a few existing tools. Koschke et al. (2006) utilizes suffix trees in abstract syntax trees to detect code clones and compared their technique with other

few techniques using the Bellon benchmark for clone detectors (Ducasse et al., 2006). Two other studies (Ducasse et al., 2006; Selim et al., 2010) also measured the performance of their proposed clone detectors (based on string comparison and intermediate source transformation) utilizing Bellon's framework.

Selim et al. (2010) showed their tool provides improved recall (with a slight drop in precision) compared to the source based clone detectors, and it also detects Type-3 clones. They also utilized Bellon's corpus in their study and compared their technique with other standalone string and token-based clone detectors where they found little higher precision. All the studies that compared different clone detectors 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 entirely 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 cloned co-change candidates during the software commit operations. Mondal et al. (2014) used the detected clone results of Nicad to predict and rank both the cloned and dissimilar co-change candidates (CCC and DCC) by analyzing evolutionary coupling from previously made change history. However, no other clone detection tool is included in their study to compare the performance of different clone detection tools in their prediction and ranking technique. We extended our previous study (Nadim et al., 2020) to compare the performance of 12 implementations of nine different clone detection tools based on the performance of detecting cloned co-change candidates using eight software systems written in C and Java programming languages. We found no other study which has performed a similar comparison of clone detectors. We believe this investigation is the first to compare clone detection tools' in the perspective of specific software maintenance acclivity (doing change impact analysis by detecting cloned co-change candidates).

3. Methodology

We have used eight open-source software systems, having varieties of size and application domains 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 2) and analyzed obtained results to evaluate the performance of those clone detection tools. Our analysis evaluates these clone detection tools' performance in successfully suggesting cloned co-changed candidates (CCC) during the software evolution and determines the ranking of these tools based on this performance evaluation. To complete this study, we performed following set of operations.

3.1. Selecting the subject systems

To select the subject systems, we first consider the popularity of programming languages and the availability of substantial revisions as essential factors. For example, the TIOBE Programming Community index (TIOBE Software, 2021) (an indicator of the popularity of programming languages) recorded C, Python, Java, and C++ as the top-4 most popular programming languages in October 2021. We also consider the availability of substantial revisions and diversity in size and application domain as an essential factor for subject systems to produce a generalizable investigation result for this study. Besides these, we also reviewed some related studies (Mondal et al., 2020, 2015, 2012, 2014) and the programming language used in these subject systems. Considering all of these factors, we have selected the subject systems used in this study. Table 1 includes the list of our eight subject systems having diverse sizes and application domains. Four of them are written in the C programming language, and the other four are in Java. As the C++ programming language share similar source code architecture, we believe the results obtained by subject systems in C also be applicable to the subject systems written in C++. Although the popularity index of the Python programming language is also very high, it does not share similar source code architecture (e.g., prior declaration of variable types, ending statements with a semicolon, use of brace to enclose statement block, etc.) with two other programming languages of this study. Python is usually considered a scripting programming language. The primary purpose of using scripting languages (e.g., Python, Perl, Ruby, and PHP) is to develop software processes and technologies for the speedy production and coordination support for increasingly expanding web applications and web services (Roy and Cordy, 2010). These scenarios motivate us to include subject systems written in C and Java programming languages in this study as they share homogeneous coding structure. However, we plan to extend this study, including a few more subject systems written in scripting languages such as Python, Perl, Ruby, and PHP in the future.

3.2. Selecting the clone detectors

Considering some related studies about the ability of clone detection tools in detecting clone fragments, we have selected some promising mechanisms for this study, which can identify all the Type-1, Type-2, and Type-3 clones. We have taken CloneWorks (Svajlenko and Roy, 2017) as it is considered 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 cloned co-change candidates. We included Duplo (Ducasse et al., 1999) as another Type-1 clone detector for comparing with Type-1 clones of CloneWorks in this investigation. ConQAT (Juergens et al., 2009), iClones (Göde and Koschke, 2009), NiCad (Cordy and Roy, 2011), and SimCAD (Uddin et al., 2013) have been reported as very good tools for detecting all types of clones in the study of Svajlenko and Roy (2014). Besides these, CCFinder (Kamiya et al., 2002), Deckard (Jiang et al., 2007), 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. The 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 cloned co-change candidates, we added Duplo in our

study. We included (Harris, 2003) in our analysis because of its ability to find cloned codes by line-by-line textual comparison supporting identifier renaming with a fast detection speed on the large repository and widespread use in several related studies (Ragkhitwetsagul et al., 2016; Wang et al., 2013; Mondal et al., 2011; Cheung et al., 2016; Krinke et al., 2010). NiCad, Sim-CAD, and Simian extract cloned code fragments from a software system's codebase using textual similarity among different code pieces. Deckard makes vector representation of different code fragments and then utilizes tree representation of those vectors to calculate the similarity among code pieces. CCFinder, ConQAT, and iClones extract tokens from the source code and then use those tokens to find similar fragments.

3.3. Selecting identical parameter configurations of the clone detectors

Taking an identical configuration of different clone detectors while applying them on different subject systems to identify cloned code fragments is essential. We compared them with each other based on their capability to successfully suggest cloned co-change candidates. Identified clones from each clone detectors have a crucial impact on their performance in suggesting cloned co-change candidates. Wang et al. (2013) reported taking different configurations of clone detectors may change the result of detected clones, and the result can be very good or terrible depending on the taken configuration. This scenario is known as a confounding configuration choice problem, and it should be handled very carefully while determining the configuration to be taken to detect cloned fragments using any clone detection tool. Our configuration of different tools is shown in Table 2. As our goal is to compare different tools with each other, configuring them similarly while clone detection will provide consistent results for a fair comparison. We have also tried to have identical configuration values to Svajlenko and Roy (2014), which they conducted to compare different clone detectors based on their efficiency in detecting cloned fragments. We provided a 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 because, in an initial manual inspection of the detected results, we found a massive number of clone fragments in Deckard's results compared to the other clone detectors while using the similarity threshold 70% like the others. We also identified many clone classes having the same fragments (self duplicate fragments) as the clone fragments of that class. To minimize those duplicated fragments, we tried other similarity threshold values such as 75%, 80%, and 85%. We found that increasing the similarity threshold reduces the self duplicated fragments, and we took the detected clone results applying the 85% similarity threshold for Deckard in all the subject systems. Svajlenko and Roy (2014) 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 cloned co-change candidates, it was essential to configure them identically during detecting clones.

3.4. Identifying cloned and dissimilar co-change candidates

We prepared the cloned co-change candidates by processing all the revisions of all the subject systems, which we mention in Table 1. We show the summary of revisions processed from each of the subject systems in Table 3. Initially, we downloaded all the source files of all the subject systems' revisions from their

Table 3
Revisions processed from each of the subject systems.

Revision	Brlcad	Camellia	Carol	Ctags	Freecol	Jabref	jEdit	QMA
Any change	2113	301	1700	774	1001	1540	215	317
≥1 change to C or Java	660	163	454	447	836	860	145	35
>1 change to C or Java	553	155	430	330	833	755	145	25

respective SVN repositories. We then applied the **Unix diff** operation between all the files of each revision with the corresponding files in the next revision to extract changes in these revisions. We prefer the Unix diff over git-diff, as there are four different variants (e.g., Myers, Minimal, Patience, and Histogram) of git-diff[1], and Nugroho et al. (2020) show that different git-diff variants can provide different source code lines as the results of git-diff using the same set of input source files. Besides these, instead of working directly on the mirrored GitHub repositories of the subject systems, we extracted all the source code files of each revision from each subject system to apply the clone detection tools to find the cloned fragments in these revisions. The clone fragments from the results of clone detectors are compared with the results of Unix diff to produce the ground truth of cloned co-change candidates for our investigation and evaluate the performance of clone detectors for detecting cloned co-change candidates. Therefore, we applied the Unix diff directly to each source code file pair from which clone detectors detected the clone fragments. We use the Unix diff command instead of any variant of four available git-diff commands to make the source code fragments consistent in the line numbers in both the output of clone detectors and the Unix diff.

We identify some specific change information such as (i) the name of the files which are changed, (ii) the beginning line numbers of each of the changes, and (iii) the ending line numbers of the corresponding changes. We repeat the change extraction process for each revision (excluding the last one) in all the subject systems. Suppose we find n changed fragments in a software revision. Then, for every single change fragment, we consider the remaining n-1 fragments as the combination of cloned and dissimilar co-change candidates for that revision. In the following step, we describe the method for excluding the dissimilar co-change fragments to produce the ground truth of cloned co-change candidates.

3.5. Identifying cloned co-change candidates (ground truth)

Even though the changes extracted from two adjacent revisions (e.g., revision n and n+1) are co-change candidates of each other, they might not be a set of cloned co-change candidates. Thus, for example, a group of co-change candidates might contain some dissimilar change candidates who changed independently, and they might not be a clone of any other changes of that co-change group. The inclusion of such non-cloned changes into our calculation can drop the detection accuracy of clone detectors. Therefore, we utilize the results from all the 12 clone detectors used in this study to produce the ground truth of cloned co-change candidates before calculating the performance metrics (precision, recall, and f1 scores) for each clone detector. We first considered each code fragment's change as target change and tried to predict all the other changes in the same revision (e.g., co-changes) utilizing each of the 12 clone detection techniques. We find some change fragments where none of the 12 clone detection techniques can detect any cloned co-change candidate. Therefore, we consider those change fragments as non-cloned (dissimilar) co-change fragments for the set of

clone detectors used in this study. Thus, we exclude the identified non-cloned co-change candidates to prepare the ground truth of cloned co-change candidates for comparing the 12 clone detection techniques. Including only the dependent (cloned) co-changes while calculating the performance metrics (precision, recall, and f1 scores) is necessary to make a fair comparison among the clone detection tools. Finally, we compare all the tools based on their calculated performance metrics in detecting cloned co-change candidates.

All the cloned fragments might not be cloned co-change candidates. For example, a clone class might contain a large number of clone fragments. However, whenever we attempt to change a code fragment, we can see that only a few clone fragments could change together with that particular fragment, and in such a case, those few other fragments are the actual cloned co-change candidates. The remaining clone fragments are not the cloned co-change candidates in such cases. Therefore, our ground truth does not directly depend on clone detection tools result. We extracted the co-change history using the Unix diff operation from the code repositories of the subject systems. We use this co-change history to make our ground truth. For example, let us assume that five code fragments changed together in a particular commit operation. Then, we consider the remaining four as the true (combination of cloned and dissimilar) co-change candidates for each of these five code fragments. Then we took the union of detected cloned co-change candidates using all the 12 clone detectors to produce the final ground truth for comparing the tools. As a gold dataset of co-change candidates currently does not exist, we believe our considered ground truth is reasonable for our comparison scenario.

3.6. Demonstrating a running example

Fig. 1 provides a demonstration of a running example in calculating each of the clone detection tools' performance metric values. Let us consider that 21 changes, C1 to C21, occurred in the codebase during a particular commit operation in a subject system. We extract these changes utilizing the UNIX diff operation. If we consider an arbitrary change (e.g., C1) as the target change, the other 20 changes (e.g., C2 to C21) will be considered the co-change candidates (combination of both cloned and dissimilar co-change candidates) of C1. We apply different clone detection tools to detect these co-change candidates considering C1 as the target change. To do so, we first take the clone class from the results of the clone detection tool and find if any code fragment from that clone class intersects with C1. Here, the intersection of any two code fragments means they share at least one common line. These code fragments could be fully overlapped or partially overlapped to consider them as intersecting code fragments. Therefore, both the fully and partially overlapped code fragments would be considered as intersecting code fragments. We consider the intersecting code fragment from the clone class as the equivalent code fragment of the target change C1. Therefore, the other code fragments in that intersecting clone class should be co-change with the target change C1. There code fragments from the clone class are the predicted co-change candidates of C1.

Let us assume, analyzing the detected clone fragments of one of the clone detection tools, Deckard (Jiang et al., 2007), we find 14 code fragments as clone candidates of C1, among those 14

---

[1] https://git-scm.com/docs/git-diff.
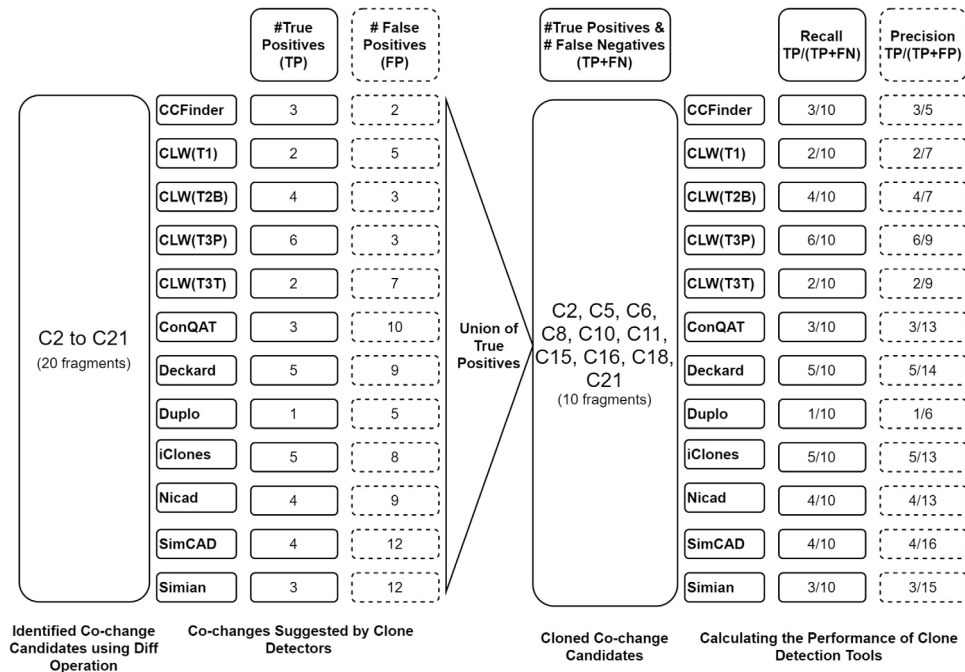
Fig. 1. This diagram explains how we calculated the Recall and Precision in one revision. We first identified all the cloned and dissimilar co-change candidates as ground truth using the Unix diff operation, then we have used the results of each of the clone detection tools to identify those cloned co-change candidates. True Positives (TP) are the co-changes that we successfully identified using the clone detectors. Our process also provided some False Positives (FP) in each revision. To separate the cloned co-change candidates from all the previously identified co-change candidates, we took the union of all the TPs from all the clone detection tools. The Recall is the ratio of TPs and the number of cloned co-change candidates (TP + FN), and the Precision is the ratio of TPs and the number of total co-change suggestions (TP + FP). We repeated the process for all the revisions of all the subject systems used in our investigation.

fragments, five intersects with the five co-change candidates of C1 (e.g., C2, C6, C8, C15, C21) from those 20 fragments. Thus, we will consider those five fragments as true positives (TP) of cloned co-change candidates (CCC), and $14 - 5 = 9$ will be the number of false positive (FP) value. Similarly, using the clone fragments of NiCad, if we detect 13 code fragments as clone candidates of C1, and four of them intersects with four of co-change candidates (e.g., C5, C10, C16, C18), then we find $TP = 4$, and $FP = 13 - 4 = 9$. We continued the same approach for all the clone detection tools to detect co-change candidates of C1. If we take the union of all the true positives obtained from the results of all the clone detectors, we get ten unique change fragments (C2, C5, C8, C15, C21, C5, C8, C10, C15, C21). We will then consider those ten unique change fragments as cloned co-change candidates and this will be used as the number of TP+FN value while calculating the recall of clone detection tools for detecting cloned co-change of C1. Using the number of TP and FP values, we can also calculate the precision of clone detection tools for C1. We finally calculated average recall, average precision in each of the subject systems for all the clone detection tools. We compare the clone detection tools based on the F1 Scores calculated by using the average recall and precision shown in the Eq. (3).

3.7. Calculating the evaluation metrics

Let us assume, while examining a specific commit operation, we found the number of fragments changed with this commit is n. In each step, we consider one of those fragments as the target fragment and the remaining $n - 1$ fragments as the co-changed candidates for that target fragment. Among the $n - 1$ fragments, there could be some non-cloned fragments that might have changed independently. We exclude such non-cloned fragments (demonstrated in Fig. 1) by taking the union of detected cloned co-change candidates by all the clone detection

tools used in this study. After excluding non-cloned fragments, we get the **Cloned Co-changed Candidate** (CCC) for each of the target fragments. Now, we find the cloned fragment from the results of clone detectors intersecting with the target fragment. The other fragments in the intersecting clone fragment class are considered the **Predicted Cloned Co-change** (PCC) candidates. We now determine how many of these PCC intersect (shares common line number) with the CCC to obtain the number of detected cloned co-change candidates by the clone detector.

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

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

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

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 (Eqs. (1), (2), and (3)).

We repeat the calculating process of Recall and Precision for all the changes in each of the subject systems with the detected clone fragments generated by all the clone detection tools. We then calculate the 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 4. We reported the ranking of the tools considering individual ranks in each of the subject systems in Table 6.

3.8. Producing the final rank list

We produce the final rank list based on the ranks of the clone detectors in each of the subject systems. 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 final and individual ranks of clone detectors. The columns $S1$ to $S8$ indicates individual ranks of clone detectors in each subject system. Individual ranks are measured based on the F1 scores of each clone detector shown in Table 4. The highest F1 Score in Table 4 got rank-1, and similarly the lowest one got rank-12 in the respective position of Table 6. Thus, every clone detection technique has eight rank values (smaller value represents the better performance), obtained in the eight software systems. The right-most column (**Rank**) shows the final rank of the clone detectors considering all the individual ranks. As smaller rank values indicate better performance of the clone detectors in detecting cloned co-change candidates, a clone detector that performed well for more subject systems, will obtain smaller $\sum_{i=1}^{8} S_i$, and received a higher position in the final rank table.

4. Experimental result

We investigate 12 clone detection techniques on the thousands of revisions from eight C and Java programming language-based software systems to detect cloned co-change candidates. In this section, we describe the obtained results and answer the research questions based on this investigation. We produce the final rank list of all the clone detection techniques in Table 6. Considering the final rank list, 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) the top two tools in the final rank list are the Type-3 configurations of CloneWorks (prior to detecting clone fragments one configuration first splits the source files into lines while the other configuration splits source files into tokens), and the next two clone detectors that perform well are Deckard and CCFinder. Therefore, to detect cloned co-change candidates, those tools (all are pattern and token-based) are the best choices compared to the other tools compared in this study. (iii) From this ranking, we also find that text-based clone detectors (such as Duplo or CloneWorks Type-1) are not suitable for detecting cloned co-change candidates.(iv) Our comparison in Fig. 4 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 perform well when detecting cloned co-change candidates. We performed the Wilcoxon Signed-Rank Test (Wilcoxon, 1945; Rosner et al., 2006) to verify whether the F1 Scores in all of 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 significance 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.

4.1. Answer to the **RQ1**

How can we compare different clone detection tools based on the performance in detecting cloned co-change candidates?
The key experimental results are in Figs. 2 and 3, Tables 4, 5, and 6, where Figs. 2 and 3 show the average Recall and average Precision of each of the clone detection tools. Table 5 shows the summary of target changes and detected cloned 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. Table 4 shows the F1 Score of each of the clone detectors in each of the subject systems. The F1 Score is calculated using Eq. (3).

Table 4
F1 score of different tools in detecting cloned co-change.

Tool	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

Table 5
Actual Target Changes (ATC) and Cloned Co-changed Candidates (CCC).

System	# ATC	# CCC	% ATC	% CCC
Brlcad	2909	33,578	7.45	1.89
Camellia	8052	346,140	20.61	19.46
Carol	4582	254,311	11.73	14.29
Ctags	718	3648	1.84	0.21
Freecol	6865	265,213	17.57	14.91
Jabref	8313	455,469	21.28	25.60
jEdit	5122	323,277	13.11	18.17
QMA	2508	97,396	6.42	5.47
Total	**39,069**	**1,779,032**	**100**	**100**

Table 6
Final rank of tools considering individual subject system rankings.

Tool	S1	S2	S3	S4	S5	S6	S7	S8	$\sum_{i=1}^{8} S_i$	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.

Our experimental results concluded in Table 6, which shows that CLW-T3P, CLW-T3T, and Deckard shows top performance (Rank 1 or 2 based on F1 Score) in most of the subject systems compared to all the other tools. The summary of the results in Table 6 also 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 cloned co-change detection 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 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 table.

As our analysis was based on the clone grouping into class or pair provided by the clone detection tools, we found that clone detection tools' efficiency 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
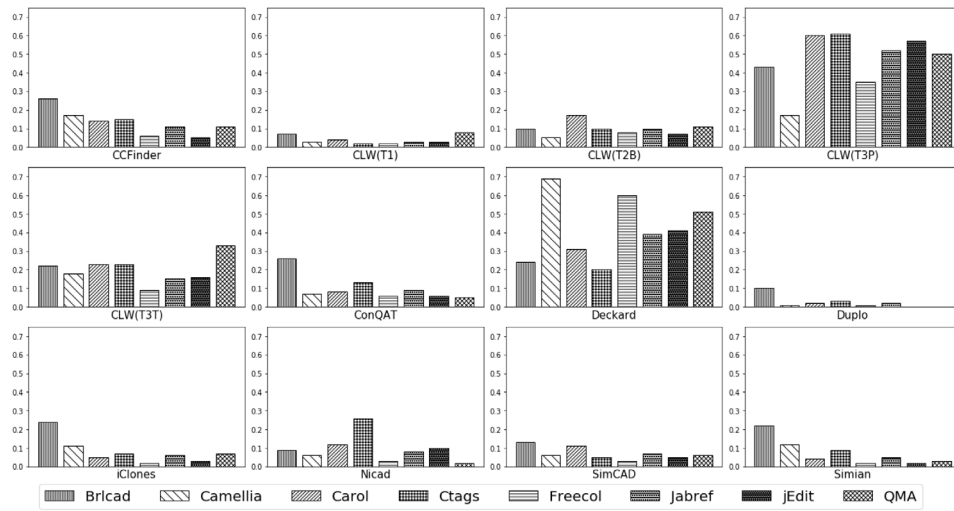
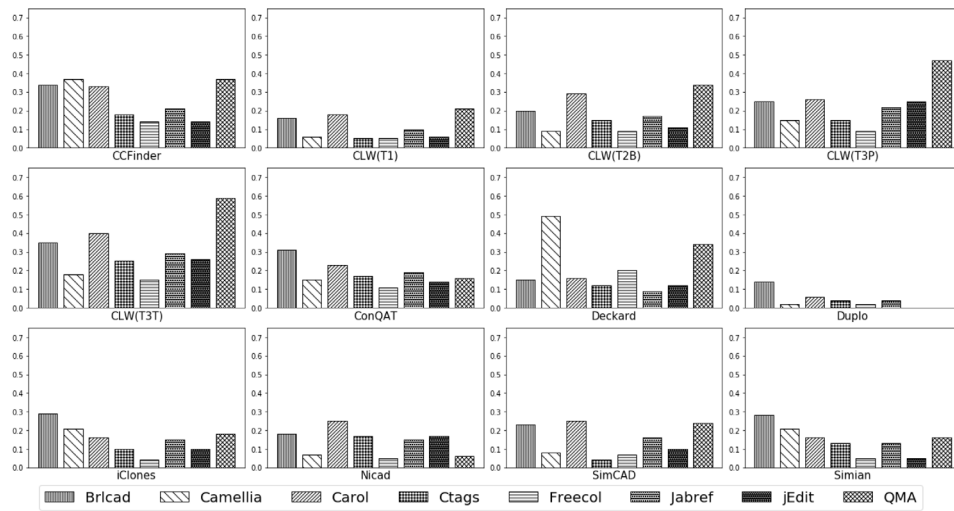**Fig. 2.** Average recall of different tools.

Fig. 3. Average precision of different tools.

in successfully suggesting cloned co-change candidate(s). Different values of different clone detectors' accuracy indicate the difference in their efficiency in this research domain.

4.2. Answer to the **RQ2**

What are the deciding factors for the performance variance of different clone detectors 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. Although we selected all the promising clone detection tools reported in several studies, we found differences in their performance in detecting cloned co-change candidates. Such a scenario 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 subject systems' revisions. Fig. 4 shows the comparison scenario of the number of clone fragments and lines covered by those clone fragments from different clone detectors. For better comparison, we bring the values on 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 in the top of the list. CLW-T3T and CCFinder fall in the respective next position to provide the highest number of clone fragments and cover the highest number of unique lines in the source files. This scenario shows that a good clone detector can perform poorly 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 (Mondal et al., 2014) suggests that NiCad is an excellent clone detector in both of these cases, it falls at the bottom of the list. Even though 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

Table 7
Wilcoxon signed rank test (p < 0.05).

Tools in investigation	Significantly better than tools (p > 0.05)	# of tools
CLW-T3P	CLWL-T3T, CCFinder, CLW-T2B, ConQAT, iClones, Simian, Nicad, SimCAD, CLW-T1, Duplo	10
CLW-T3T	CLWL-T2B, ConQAT, iClones, Simian, Nicad, SimCAD, CLW-T1, Duplo	8
Deckard	CLWL-T2B, iClones, Simian, Nicad, SimCAD, CLW-T1, Duplo	7
CCFinder	ConQATL, 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
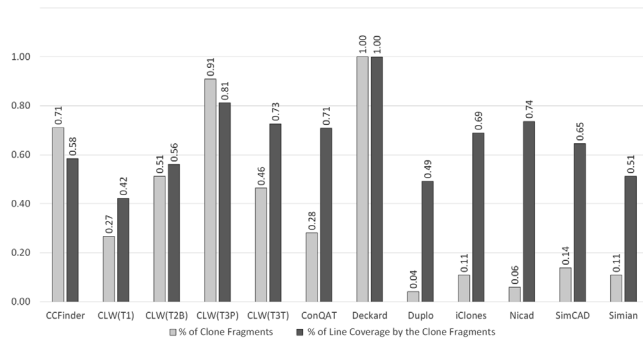
Fig. 4. Comparing unique line coverage by clone fragments and number of clone fragments from different clone detectors.

Fig. 5. Comparing distribution of F1 scores in different clone detectors.

obtained comparison scenario of the clone detectors in predicting cloned co-change candidates. Some other factors such as (i) How clone fragments are overlapped with each other in a clone group? (ii) How is a clone detector determining the similarity among different fragments? (iii) What similarity score is used; it may also have an effect on the performance of predicting cloned co-change candidates. We plan to explore these factors in future studies.
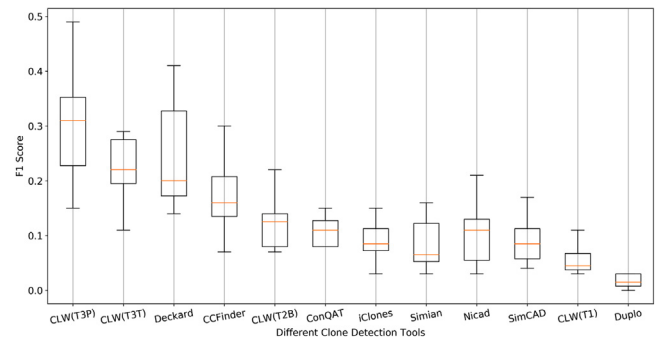
4.3. Answer to the **RQ3**

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

We can answer this research question by analyzing our final ranking of the clone detectors in Table 6. Top two clone detectors of the final rank list (Rank 1 and 2) work by extracting source code patterns from the codebase. 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 suitable 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 perfect for detecting cloned 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 cloned 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 and Type-2 are also automatically included with Type-3 clones). Therefore, our findings suggest that we should choose those clone detectors to be used in detecting cloned co-change candidates that detect Type-3 clones with the other Type-1 and Type-2 clone fragments.

4.5. The Wilcoxon Signed-Rank Test

We performed The Wilcoxon Signed-Rank Test (Wilcoxon, 1945; Rosner et al., 2006) 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 4) from both CLW-T3P and CLW-T3T, which will be then used to perform Wilcoxon Signed-Rank Test utilizing the SciPy library (Virtanen et al., 2020) available 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 significance 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 detectors whose F1 Scores are significantly different from the F1 Scores of the tool under investigation. Therefore, CLW-T3P provides significantly different F1 Scores than 10 other clone detectors (excluding Deckard). The distribution of F1 Scores in Fig. 5 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 Score values in CLW-T3P are above Deckard's values, 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-T1 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 cloned co-change candidates, CloneWorks Type-3 clone detection configuration can be an excellent choice, Deckard and CCFinder are also good choices. However, the other tools are not significantly better choices to detect cloned co-change candidates during software evolution.

The distribution of F1 Scores in Fig. 5 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 most higher values, and Duplo has the most lower values in their respective distributions. Any two tools' performance will be significantly different from each other if they share a fewer common range of F1 Scores distribution. From the result of the significance test in Table 7, we can see that Deckard is not significantly different from all the other three good clone detectors i.e. CLW-T3P, CLW-T3T, and CCFinder as they share most of the same 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, their F1 Scores are significantly better than only Duplo. Simian, Nicad, SimCad, CLW-T1 shares most of the same range of 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 principally focuses on the clone tracking perspective. A clone tracker's main task is to suggest similar cloned co-change candidates when a programmer attempts to change a code fragment. For suggesting cloned 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 cloned 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 cloned co-change candidates during software evolution. As the clone classes/pairs generated by different clone detectors played an essential 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 cloned 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, Type-2, and Type-3 clones can also perform well in detecting cloned co-change candidates.

When a particular code fragment is changed, we apply the clone detectors to predict which other similar code fragments might need to be co-changed. However, some different fragments might also be changed together with the particular fragment. As we apply only clone detectors, we cannot consider those dissimilar co-change candidates in our research. We only apply our analysis to those change candidates whose co-changes are detected by at least one (out of 12) clone detection techniques in our investigation. We believe, removing the change candidates whose co-change is not detected by any of the clone detectors lead to a fair comparison among the clone detection tools.

Mondal et al. (2020) show that combining Nicad as a clone detection tool with an association rule mining-based change impact analysis tool (Rolfsnes et al., 2016) can significantly enhance the overall performance of detecting change impact set in software systems. As that study only included Nicad in their investigation, our study opens an opportunity to perform a few similar study other clone detectors which show better performance in our investigation. We find the few clone detectors such as CloneWorks, Deckard, CCFinder perform better in detecting cloned co-change candidates compared to Nicad. Therefore, including these tools in combination with Tarmaq might also serve better in change impact analysis. However, a practical evaluation is required to validate this assumption.

In our research, we do not compare the clone detectors considering their clone detection efficiency. We instead compare the clone detection tools based on their ability to suggest cloned co-change candidates. Such a comparison of clone detectors focusing on a particular maintenance perspective was not made previously. However, suggesting both the cloned and dissimilar co-change candidates for a target program entity is a vital impact analysis (Arnold, 1996) task during software evolution. Thus, through our research, we investigate which of the clone detectors can be helpful in change impact analysis to what extent. Findings from our study can identify which clone detector(s) can be promising for change impact analysis by finding the cloned co-change candidates. Furthermore, this study can also contribute to finding possible fixes of inconsistencies in software systems by analyzing historical discrepancies (due to missing the change in cloned co-change candidates) and their solutions (e.g., their fixing patterns).

6. Threats to validity

We use eight subject systems of diverse size and application domains written in C and Java programming languages and process thousands of revisions to rank 12 clone detection techniques for detecting cloned co-change candidates. The inclusion of more subject systems may increase the generalizability of the findings. The decision to select subject systems is based on the variety, popularity, used programming languages, and availability of a considerable number of revisions. We believe our results are not

biased by our choice of subject systems and are distinguished from software maintenance perspectives. As the number of software revisions in our subject systems is extensive, our results should be generalizable to the other subject systems written in C or Java programming languages. Our results should also be generalizable to the programming languages whose structure is similar to C and Java, such as C++ or C#. It might be possible that our results are not generalizable to other programming languages such as different scripting languages (e.g., Python, R, or PHP). We are detecting clone fragments using clone detectors and process that result to suggest cloned co-change candidates. As the clone detectors used in this study are established, and these are the most widely used tools for C and Java software projects (Ain et al., 2019), their results might not be biased to any particular programming language. These tools are expected to work equally to all the subject systems written in C and Java programming languages. Therefore, we expect our results are generalizable to subject systems written in C and Java programming languages. However, in the future, we plan to investigate the generalizability of our technique to other programming languages by including a few more subject systems written in C#, Go, Python, R, PHP, etc.

The parameters to detect clone fragments used in the 12 clone detectors may impact the comparison scenario in this study. However, we considered taking equivalent configurations (such as similarity threshold, length of tokens, renaming variables, splitting the source code lines, etc.) to minimize such impact. Therefore, we believe comparing clone detectors based on their detected cloned fragments and the cloned co-change candidate is made without influencing their clone detection configurations.

We took both the fully and partially overlapped source code fragments as the intersecting code fragments from the results of clone detection tools and the cloned co-change candidates in our ground truth. Taking any other consideration to determine the overlap between two source code fragments might change the number of overall detected cloned co-change candidates. However, the change in the overlapping constraints should affect the results of all the clone detection tools and subject systems equally, and our overall comparison scenario of clone detection tools should still remain the same. We plan to practically verify different overlapping constraints in our future studies.

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 (e.g., change together) for ensuring consistency of the codebase. However, dissimilar code fragments can co-change because of their underlying dependencies, which could impact 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 12 clone detectors used in this study, we discarded the candidate. Excluding these changes, preparing the ground truth of cloned co-change candidates is essential for a fair comparison of the performance of clone detectors in this investigation. Using a different set of clone detectors might change the ground truth of cloned co-change candidates. Still, the ground truth of cloned co-change candidates in this investigation is accurate and complete for the set of 12 cloned detectors of this study. We believe that such a consideration is reasonable in our experiment aiming towards comparing clone detectors. Our findings may inspire more similar research with a different set of clone detectors in the future.

We did not consider a sliding window protocol that views some consecutive revisions in a single window to find cloned co-change candidates. There are some advantages of using the sliding window protocol. The sliding window protocol could provide a good result if some related cloned co-change candidates are scattered in two or more consecutive commits. As in this research, we are comparing 12 clone detectors. The effect of not including a sliding window protocol will be the same for each of the clone detectors, and thus, it does not have any impact on the final rank list of the clone detectors. However, we plan to apply this technique in our future research.

7. Conclusion and future work

In this research, we compare different clone detection tools and investigate their performances in predicting cloned co-change candidates during software evolution. We selected eight open-source subject systems written in C and Java for our analysis. Our final rank list in Table 6 and summary of significance test results in Table 7 show that both the Pattern and Token configurations of the CloneWorks clone detection tool for detecting Type-3 clones perform significantly better compared to more than 72% of the other clone detectors used in this study. Deckard and CCFinder are also better compared to more than 55% of the other tools. CloneWorks (Type-2), ConQAT, iClones also show better performance than the other remaining tools. Our source codes, datasets, and other processed results are publicly available (Nadim et al., 2021) for researchers and practitioners to help continue and reproduce the results of this study.

Although we determined some reasons for the better performance of Deckard, CloneWorks, and CCFinder in this extensive study, we plan to do some future work analyzing the internal mechanism of clone detection tools. We also want to find out how the change of these mechanisms affects the detection of cloned co-change candidates. In our future work, we also want to investigate the impact of different similarity scores of different clone detectors in finding cloned co-change candidates. We also want to include other software systems written in different programming languages (e.g., C#, Go, Python, R, PHP). We want to consider a sliding window (Muller et al., 2017) protocol to consider more than one software revisions in a group to find cloned co-change candidates in the future extension of this study.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Ain, Q.U., Butt, W.H., Anwar, M.W., Azam, F., Maqbool, B., 2019. A systematic review on code clone detection. IEEE Access 7, 86121–86144.

AlSanad, A., Chikh, A., 2015. The impact of software requirement change – a review. In: Rocha, A., Correia, A.M., Costanzo, S., Reis, L.P. (Eds.), New Contributions In Information Systems and Technologies. Springer International Publishing, Cham, pp. 803–812.

Arnold, R.S., 1996. Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA, ISBN: 0818673842.

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

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Burd, E., Bailey, J., 2002. Evaluating clone detection tools for use during preventative maintenance. In: Proc. SCAM, pp. 36–43.

Cheung, W.T., Ryu, S., Kim, S., 2016. Development nature matters: An empirical study of code clones in JavaScript applications. Empirical Softw. Eng. 21 (2), 517–564.

Cordy, J.R., Roy, C.K., 2011. The NiCad clone detector. In: Proc. ICPC, pp. 219–220.

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

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

Göde, N., Koschke, R., 2009. Incremental Clone Detection. In: Proc. CSMR, pp. 219–228.

Harris, S., 2003. Simian - similarity analyser – duplicate code detection for the enterprise–overview. 2003 (accessed July 6, 2020). URL: http://www.harukizaemon.com/simian/.

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

Islam, J.F., Mondal, M., Roy, C.K., Schneider, K.A., 2019. Comparing bug replication in regular and micro code clones. In: Proc. ICPC, pp. 81–92.

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

Juergens, E., Deissenboeck, F., Hummel, B., 2009. CloneDetective - A workbench for clone detection research. In: Proc. ICSE, pp. 603–606.

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

Koschke, R., Falke, R., Frenzel, P., 2006. Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. pp. 253–262.

Krinke, J., Gold, N., Jia, Y., Binkley, D., 2010. Cloning and copying between GNOME projects. In: Proc. MSR, pp. 98–101.

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

Mondal, M., Roy, B., Roy, C.K., Schneider, K.A., 2019. Investigating context adaptation bugs in code clones. In: Proc. ICSME, pp. 157–168.

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

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

Mondal, M., Roy, C.K., Schneider, K.A., 2014. Prediction and ranking of co-change candidates for clones. In: Proc. MSR 2014. ACM, New York, NY, USA, pp. 32–41.

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

Muller, O., Butman, A., Butman, M., 2017. Opening a (sliding) window to advanced topics. In: Proceedings of the 2017 ACM Conference on Innovation and Technology In Computer Science Education. In: ITiCSE '17, Association for Computing Machinery, New York, NY, USA, pp. 52–57.

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

Nadim, M., Mondal, M., Roy, C.K., Schneider, K.A., 2021. Research artifact: Evaluating the performance of clone detection tools in detecting cloned co-change candidates. 2021 (accessed December 14, 2021). URL: https://github.com/mnadims/FindCochangeByClone.git.

Nayrolles, M., Hamou-Lhadj, A., 2018. CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). pp. 153–164.

Nugroho, Y.S., Hata, H., Matsumoto, K., 2020. How different are different diff algorithms in git? Empir. Softw. Eng. 25 (1), 790–823. http://dx.doi.org/10.1007/s10664-019-09772-z.

Ragkhitwetsagul, C., Krinke, J., Clark, D., 2016. Similarity of source code in the presence of pervasive modifications. In: Proc. SCAM, pp. 117–126.

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

Rosner, B., Glynn, R.J., Lee, M.-L.T., 2006. The wilcoxon signed rank test for paired comparisons of clustered data. Biometrics 62 (1), 185–192.

Roy, C., Cordy, J., 2008. Scenario-based comparison of clone detection techniques. In: Proc. ICPC, pp. 153–162.

Roy, C.K., Cordy, J.R., 2010. Are scripting languages really different? In: Proceedings of the 4th International Workshop on Software Clones. In: IWSC '10, Association for Computing Machinery, New York, NY, USA, pp. 17–24.

Roy, C.K., Cordy, J.R., 2018. Benchmarks for software clone detection: A ten-year retrospective. In: Proc. SANER, pp. 26–37.

Roy, C.K., Cordy, J.R., Koschke, R., 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. SCIENCE OF COMPUTER PROGRAMMING.

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

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

Svajlenko, J., Roy, C.K., 2014. Evaluating modern clone detection tools. In: Proc. ICSME, pp. 321–330.

Svajlenko, J., Roy, C.K., 2017. 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). pp. 177–179.

Swierstra, W., Löh, A., 2014. The semantics of version control. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. In: Onward! 2014, Association for Computing Machinery, New York, NY, USA, pp. 43–54.

TIOBE Software, 2021. TIOBE index – TIOBE - the software quality company. 2021 (accessed September 27, 2021). URL: https://www.tiobe.com/tiobe-index/.

Uddin, M.S., Roy, C.K., Schneider, K.A., 2013. SimCad: An extensible and faster clone detection tool for large scale software systems. In: Proc. ICPC, pp. 236–238.

Virtanen, P., Gommers, R., Oliphant, T., Haberland, M., Reddy, T., Cournapeau, D., ...Contributors, 2020. SciPy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods 17, 261–272.

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

Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biom. Bull. 1 (6), 80–83.