Gapped Code Clone Detection with Lightweight Source Code Analysis

Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{h-murakm, k-hotta, higo, igaki, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—A variety of methods detecting code clones has been proposed before. In order to detect gapped code clones, AST-based technique, PDG-based technique, metric-based technique and text-based technique using the LCS algorithm have been proposed. However, each of those techniques has limitations. For example, existing AST-based techniques and PDG-based techniques require costs for transforming source files into intermediate representations such as ASTs or PDGs and comparing them. Existing metric-based techniques and text-based techniques using the LCS algorithm cannot detect code clones if methods or blocks are partially duplicated. This paper proposes a new method that detects gapped code clones using the Smith-Waterman algorithm to resolve those limitations. The Smith-Waterman algorithm is an algorithm for identifying similar alignments between two sequences even if they include some gaps. The authors developed the proposed method as a software tool named CDSW, and confirmed that the proposed method could resolve the limitations by conducting a quantitative evaluation with Bellon's benchmark.**

***Index Terms*—Code Clone, Program Analysis, Software Maintenance, Tool Comparison**

I. Introduction

Recently many studies have investigated the premises of "the presence of code clones makes software maintenance more difficult" quantitatively. Those studies used different detection tools on different experimental targets with different environments. Thus, there is no general result about the harmfulness of code clones. However, to summarise this point then it is said "not all code clones make software maintenance more difficult" [1]. Consequently, removing or not generating all code clones are inappropriate from the perspective of efficient software development or maintenance. It is important to minimize the risk of code clones with low cost.

From a viewpoint of program comprehension, analysis of code clones plays an important role. Refactoring and removing some code clones can improve readability, maintainability and manageability of software systems [2].

Programmers often make some changes to code fragments after cloning to adjust the code fragments to the destination of the cloning [3]. Moreover, cloned fragments often evolve differently from the original fragments [4]. These facts indicate that there often exists some gaps between the original code fragments and pasted fragments. In order to detect code clones appropriately, it is necessary to detect code clones even if they include some gaps. In other words, detecting gapped clones is required for better understanding of clones and software systems.

A number of techniques detecting code clones have been proposed before now [5]. Those techniques can be classified roughly into following categories;

- text-based techniques,
- token-based techniques,
- metric-based techniques,
- AST (Abstract Syntax Tree)-based techniques,
- PDG (Program Dependency Graph)-based techniques.

In those detection techniques, some text-based techniques, metric-based techniques, AST-based techniques and PDG-based techniques can detect gapped code clones. However, each of them has limitations. Some text-based techniques or metric-based techniques cannot find code clones lying in a part of a block. AST-based techniques or PDG-based techniques require much time to detect code clones because transforming source files into ASTs or PDGs and comparing them are NP-hard problem. In order to resolve those limitations, we propose a clone detection method using the Smith-Waterman algorithm [6]. The proposed method detects not only contiguous but also gapped code clones in a shorter time frame than the ASTs or PDGs techniques. The reason is that the proposed method does not use any intermediate representations such as ASTs or PDGs. Furthermore, the proposed method detects code clones that the metric-based or the LCS-based techniques cannot detect because these techniques perform coarse-grained detections such as method-based or block-based. On the other hand, the proposed method performs a fine-grained detection that identifies sentence-based code clones.

We implemented the proposed method and evaluated it by using Bellon's benchmark [7]. However, Bellon's benchmark has a limitation that the reference of gapped code clones does not have information about where gaps are. Bellon's reference represents code clones with only the information about where they start and where they end. We do not consider that gapped parts of code clones should be regarded as code clones. Thus, Bellon's benchmark is likely to evaluate gapped code clones incorrectly when it is used as-is. Therefore, we remade the reference of code clones with information about where gaps are. Moreover, we compared the result by using Bellon's reference with that by using our reference.

Consequently, the contributions of this paper are as following.

978-1-4673-3091-6/13/$31.00 © 2013 IEEE

- We tailored the Smith-Waterman algorithm to code clone detection. First, although the original Smith-Waterman algorithm identifies only one pair of similar subsequences from two sequences, the tailored Smith-Waterman algorithm can identify multiple pairs of them. Second, the tailored the Smith-Waterman algorithm can detect code clones with consideration for the size of them or the gapped code fragments.
- Using the information about where gaps are improved the accuracy of the evaluation of $recall$, $precision$ and $f\text{-}measure$ compared to using only the information about where code clones start and they end.
- We confirmed that the proposed method had higher $F\text{-}measure$ than the existing methods.

The rest of the paper is organized as follows: Section II introduces the concept of the Smith-Waterman algorithm. We provide an overall summary of the proposed method in Section III. Section IV describes the overview of investigation, then Section V, Section VI and Section VII report the evaluations of the proposed method in detail. Section VIII describes threats to validity. Section IX discusses the experimental result or previous techniques. Section X summarizes this paper and refers to the future work.

II. The Smith-Waterman algorithm

The Smith-Waterman algorithm [6] is an algorithm for identifying similar alignments between two base sequences. This algorithm has an advantage that it can identify similar alignments even if they include some gaps. Figure 1 shows an example of the behavior of the Smith-Waterman algorithm applied to two base sequences, "GACGACAACT" and "TACACACTCC". The Smith-Waterman algorithm consists of the following five steps.

Step A (creating a table): a $(N+2)\times(M+2)$ table is created, where N is the length of one sequence $\langle a_1, a_2, \cdots, a_N\rangle$ and M is the length of the other sequence $\langle b_1, b_2, \cdots, b_M\rangle$.

Step B (initializing the table): the top row and leftmost column of the table created in Step A are filled with two base sequences as headers. The second row and column are initialized to zero.

Step C (calculating scores of all cells in the table): scores of all the remaining cells are calculated by using the following formula.

$$v_{i,j}(2 \le i, 2 \le j) = max \begin{cases} v_{i-1,j-1} + s(a_i, b_j), \\ v_{i-1,j} + gap, \\ v_{i,j-1} + gap, \\ 0. \end{cases} \quad (1)$$

$$s(a_i, b_j) = \begin{cases} match & (a_i = b_j), \\ mismatch & (a_i \neq b_j). \end{cases} \quad (2)$$

where $v_{i,j}$ is the value of $c_{i,j}$; $c_{i,j}$ is the cell located at the i^{th} row and the j^{th} column; $s(a_i, b_j)$ is a similarity of matching a_i with b_j; a_i is the i^{th} value of one sequence and b_j is the j^{th} value of the other sequence. Note that $match$, $mismatch$ and gap indicate score parameters.

$Match$, $mismatch$ and gap can be set all kinds of values freely. In Figure 1, parameters ($match$, $mismatch$, gap) are set (1, -1, -1) for ease of explanation.

While calculating values of each cell in the table, a pointer from the cell that is used for calculating $v_{i,j}$ to the cell $c_{i,j}$ is created. For example, in Figure 1, $v_{9,11}(=5)$ is calculated by adding $v_{8,10}(=4)$ and $s(v_{0,11}, v_{9,0})(=1)$. In this case, a pointer from $c_{8,10}$ to $c_{9,11}$ is created because $v_{9,11}$ is calculated with the value of $c_{8,10}$.

Step D (traceback of the table): traceback means the moving operation from $c_{i,j}$ to $c_{i-1,j}$, $c_{i,j-1}$ or $c_{i-1,j-1}$ using the pointer created in Step C. Tracing the pointer reversely represents traceback. Traceback begins at the cell whose score is maximum in the table. This continues until cell values decreased to zero.

Step E (identifying similar alignments): the array elements pointed by the traceback path are identified as similar local alignments.

In Figure 1, the hatched cells with numbers represent the traceback path. The array elements pointed by the traceback path are regarded as similar local alignments, hence two alignments "ACGACAACT" and "ACACACT" are detected as similar alignments.

III. The Proposed Method

The proposed method takes the followings as its input:

- **source files**,
- **minimal clone length** (number of tokens),
- **maximal gap rate** (ratio of gapped tokens in the detected tokens),
- **score parameters** $match$, $mismatch$ and gap.

In this paper, score parameters ($match$, $mismatch$ and gap) were decided by a preliminary experiment. Section V reports how to decide these parameters.

The proposed method outputs a list of detected clone pairs. The proposed method consists of the following five steps.

Step 1: performing lexical analysis and normalization

Step 2: calculating hash values for every statement

Step 3: identifying similar hash sequences

Step 4: identifying gapped tokens

Step 5: mapping identical subsequences to the source code

Figure 2 shows an overview of the proposed method. Figure 3 shows an example of the detection process using the proposed method. The remainder of this Section explains each step in detail.

A. Step 1: Performing Lexical Analysis and Normalization

All the target source files are transformed into token sequences. User-defined identifiers are replaced with specific tokens to detect not only identical code fragments but also similar ones as code clones even if they include different variables. All modifiers are deleted for the same reason.

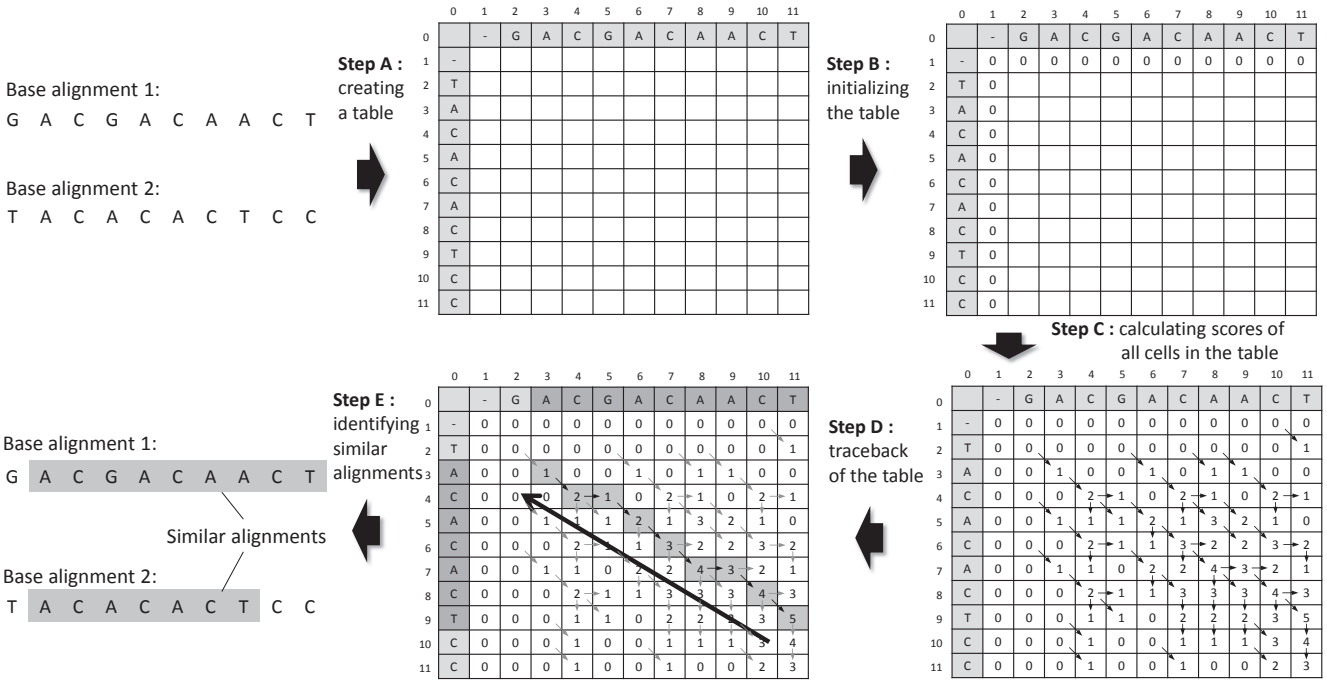


Figure 1. The Smith-Waterman Algorithm Applied to Two Base Sequences, “GACGACAACT” and “TACACACTCC”.

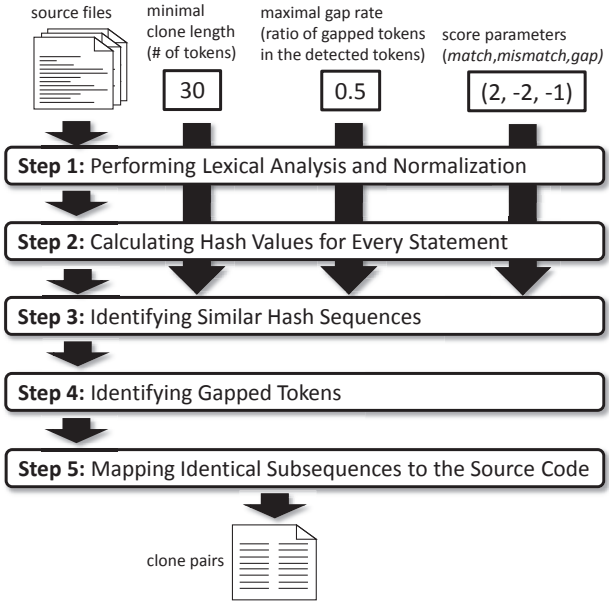


Figure 2. Overview of the Proposed Method.

B. Step 2: Calculating Hash Values for Every Statement

A hash value is generated for every statement in the token sequences. Herein, we define a statement as every subsequence between semicolon (“;”), opening brace (“{”), and closing brace (“}”). Note that every hash has the number of tokens included in its statement. At the end of Step 2, one source file is transformed into one hash sequence.

C. Step 3: Identifying Similar Hash Sequences

Similar hash sequences are identified from hash sequences generated in Step 2 by using the Smith-Waterman algorithm. Note that every pair of hash sequences creates one table. In other words, one table is used for detecting code clones that are included in two source files. In order to detect code clones in target source files, more than one tables are created. Herein, we make following changes to Step D described in Section II to tailor the algorithm for code clone detection.

- Traceback begins at multiple cells in order to detect all clone pairs between two source files. In particular, cells are searched from the lower right to the upper left and cells $c_{i,j}$ that have the following characteristics are selected as start cells of traceback.
  - $v_{i,j} > 0$
  - $v_{i,0} = v_{0,j}$

  Moreover, assume that a traceback starts at $c_{i,j}$ and ends at $c_{k,l}$ $(k \leq i, l \leq j)$, the cells included in the following set S will be out of scope from all the traceback following the current traceback.

$$S = \{c_{m,n} \mid k \leq m \leq i \wedge l \leq n \leq j\} \tag{3}$$

  The purpose of reducing the scope of traceback is in order not to detect redundant code clones.
- The number of tokens and gaps are counted during traceback in order to detect code clones whose token length is greater than the minimal clone length and the ratio of gapped tokens in the detected tokens is less than the maximal gap rate.

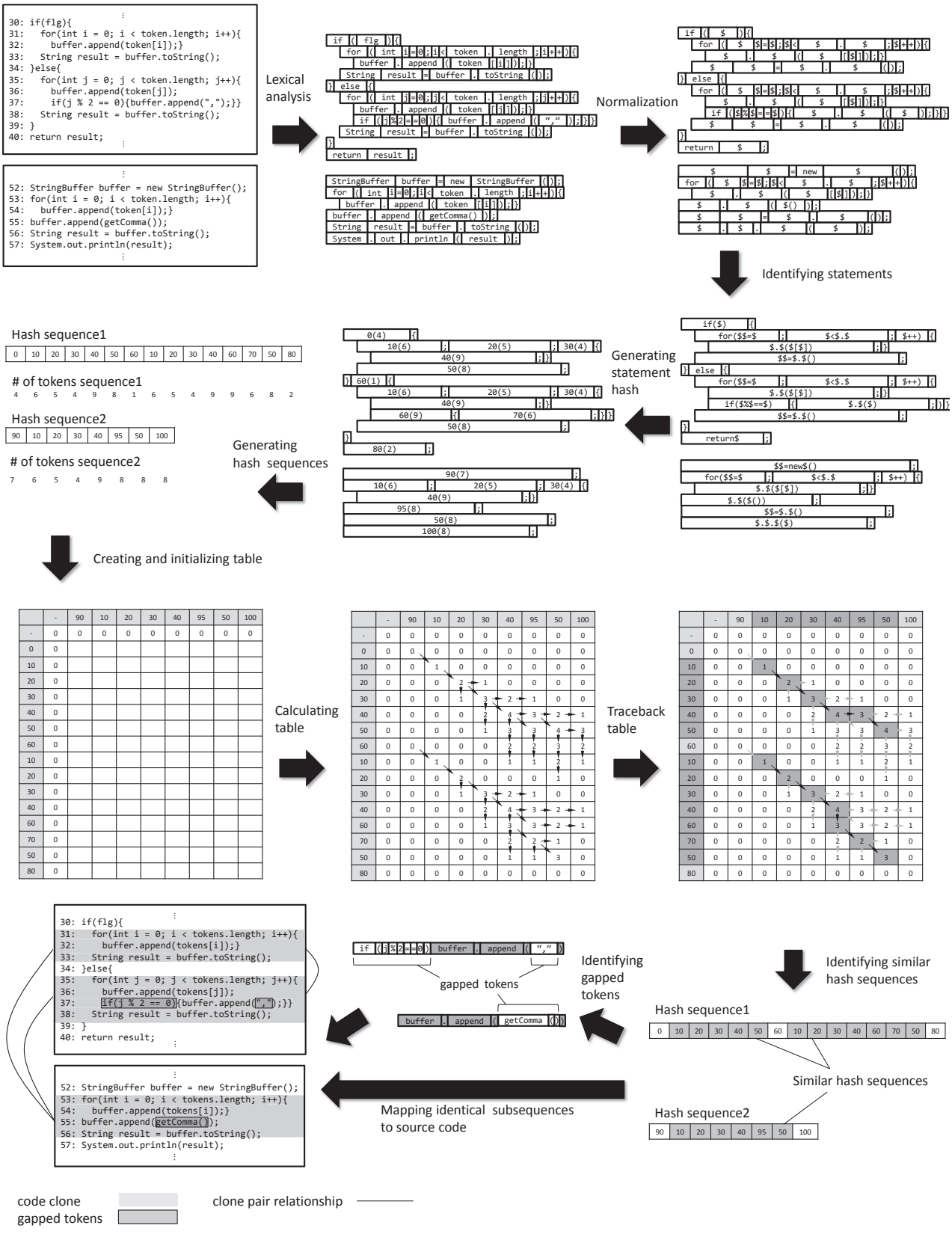


Figure 3. An Example of Detection Process Using the Proposed Method.

While traceback is being performed, gapped statements can be identified. Then, token sequences consisting of gapped statements are obtained. They are used in Step 4.

D. Step 4: Identifying Gapped Tokens

The LCS algorithm is applied to every pair of token sequences included in gapped statements identified in Step 3. The purpose of LCS application is identifying token-level gaps.

E. Step 5: Mapping Identical Subsequences to the Source Code

The identical subsequences detected in Steps 3 and 4 are mapped to the source code (the file path, the start line, the end line and the gapped lines), which are clone pairs. Note that the gapped line represents the line that contains the gapped tokens. This representation can make sense because the location information of code clones is represented by line numbers.

IV. Experimental Design

A. Overview of the Experiments

We have developed a software tool, CDSW based on the proposed method described in Section III. Then, we conducted a preliminary experiment to reveal what combinations of parameters (match, mismatch, gap) in the Smith-Waterman algorithm are appropriate for clone detection. Moreover, we conducted two experiments to answer the following three research questions for confirming the effectiveness of the proposed method.

RQ 1: Should the measures for $recall$ and $precision$ take gaps into account?

RQ 2: Does the proposed method have higher accuracy than existing methods?

RQ 3: Does the proposed method scale to large systems?

Experiment A investigates RQ 1 and Experiment B investigate RQ 2 and RQ 3, respectively.

B. Experimental Targets and Environment

In order to calculate accuracy, reference clones are necessary. The experiments take freely available code clone data in the literature [8] as a reference set (a set of code clones to be detected). The reference set includes code clones information of eight software systems. In this study, we use the software systems as a target. Table I shows an overview of the target systems.

The execution environment in these experiments was 2.27GHz Intel Xeon CPU with 16.0GB main memory.

C. Terms

In the remainder of this paper, we use the following terms.

Clone candidates: code clones detected by clone detectors.

Clone references: code clones included in the reference set.

We use the ok value to decide whether every clone candidate matches any of the clone references or not. The ok value is over 0 including 0 and under 1 including 1. The ok value means intuitively the overlapping ratio of the clone candidate and the clone reference. The more ok value increases, the more the overlapping part of the clone candidate and the clone reference is large. Meanwhile, the $good$ value is also defined in Bellon's benchmark [7]. the $good$ value is much more restrictive for a candidate-reference match. However, we use only ok value because we consider that it is enough to identify the location of code clones roughly. In this investigation, We use 0.7 as the threshold, which is the same value used in Bellon's benchmark.

We calculate $recall$ and $precision$ for evaluating the detection capability. Assume that R is a detection result (a set of clone pair), S_{refs} is the set of the clone references, and S_R is a set of the clone candidates whose ok values with an instance of the clone references are equal to or greater than the threshold in R. $Recall$, $precision$ and $f\text{-}measure$ are defined as follows.

$$Recall = \frac{|S_R|}{|S_{refs}|}. \quad (4)$$

$$Precision = \frac{|S_R|}{|R|}. \quad (5)$$

$$F\text{-}measure = \frac{2 \times Recall \times Precision}{Recall + Precision}. \quad (6)$$

This evaluation has a limitation on $recall$ and $precision$. The clone references used in the experiments are not all the relevant code clones included in the target systems. Consequently, the absolute values of $recall$ and $precision$ are meaningless. $Recall$ and $precision$ can be used only for relatively comparing detection results. Moreover, we have to pay a significant attention to $precision$. A low value only means that there are many clone candidates not matching any of the clone references. As described above, clone references are detected from a part of code clones in the target software systems. In other words, there should exist many clones that might be regarded as references in unchecked clones. Hence, not all the clone candidates that do not match any of the clone references are false positives.

The details of each Experiment are described in Section V, Section VI and VII, respectively.

V. Preliminary Experiment

The purpose of Preliminary Experiment is to obtain the appropriate parameters ($match$, $mismatch$, gap) for each of target software systems when we use the Smith-Waterman

TABLE I
TARGET SOFTWARE SYSTEMS

Name	Language	Lines of code	# of references
netbeans	Java	14,360	55
ant	Java	34,744	30
jdtcore	Java	147,634	1,345
swing	Java	204,037	777
weltab	C	11,460	275
cook	C	70,008	440
snns	C	93,867	1,036
postgresql	C	201,686	555

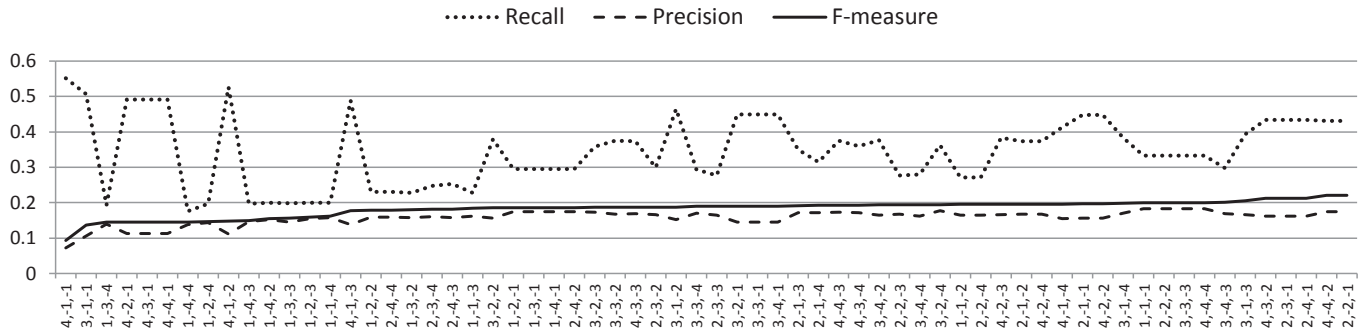


Figure 4. $Recall$, $Precision$ and $F\text{-}measure$ on 3-tuple of Parameters ($match$, $mismatch$, gap).

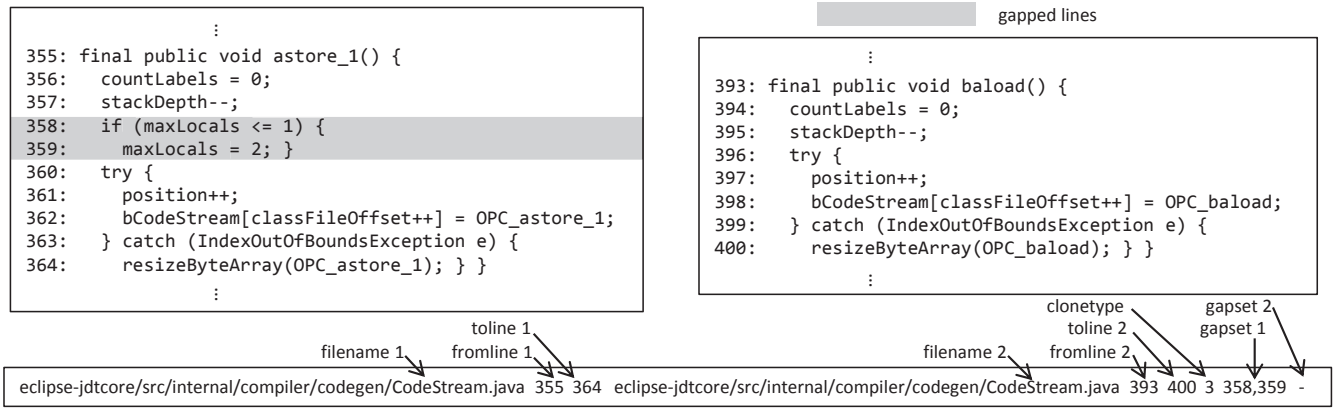


Figure 5. An Example of the Clone Reference We Remade (Clone Reference No. 1101).

algorithm. In this experiment, we investigated following ranges of parameters.

$$match = \{x \in \mathbb{Z} \mid 1 \leq x \leq 4\} \quad (7)$$

$$mismatch = \{y \in \mathbb{Z} \mid -4 \leq y \leq -1\} \quad (8)$$

$$gap = \{z \in \mathbb{Z} \mid -4 \leq z \leq -1\} \quad (9)$$

where $\mathbb{Z}$ represents the set of integers.

We calculated $recall$, $precision$ and $f\text{-}measure$ for each of target software systems on 64 ($= 4 \times 4 \times 4$) cases. Then, we evaluated the median of $recall$, $precision$ and $f\text{-}measure$ for eight target software systems. Figure 4 shows the $recall$, $precision$ and $f\text{-}measure$ on 3-tuples ($match$, $mismatch$, gap) in ascending order by $f\text{-}measure$. $F\text{-}measure$ is the harmonic mean of $recall$ and $precision$. Thus, high $f\text{-}measure$ means both $recall$ and $precision$ are reasonably high.

From Figure 4, it was revealed that $f\text{-}measure$ was the maximum when ($match$, $mismatch$, gap) is (2, -2, -1) or (4, -4, -2). Each of the parameters in (4, -4, -2) is twice from each of that in (2, -2, -1). Therefore, these two tuples of parameters produced same results in the Smith-Waterman algorithm.

In addition, $recall$ tended to be high when $match$ was high. The reason was that high $match$ makes the number of clone candidates large, and many clone reference are likely to be contained in clone candidates. At the same time, $precision$ tended to be high when $mismatch$ and gap were low. The reason was that low $mismatch$ and gap make the number of clone candidates small, and clone candidates were likely to contain many clone references relatively.

Accordingly, we used ($match$, $mismatch$, gap) = (2, -2, -1) in the following Experiment A and Experiment B.

VI. Experiment A

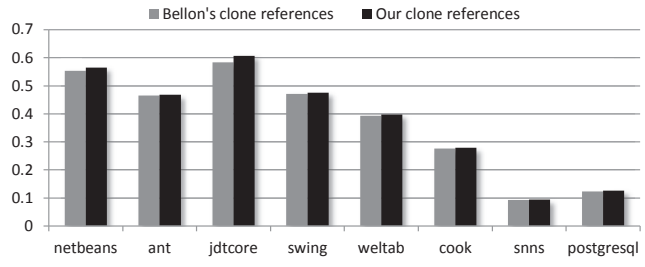
The purpose of Experiment A is to reveal how $recall$, $precision$ and $f\text{-}measure$ are changed by our defined formula. In Bellon's benchmark [7], in order to determine whether a candidate matches a reference, $ok(CP_1, CP_2)$ are used, where CP_1 and CP_2 are clone pairs.

However, these formulae do not consider the gapped fragments included in code clones. Therefore, we remade the clone references with information of gapped lines and made it public on the website[1]. Furthermore, we put the file format of our clone references on the same website.

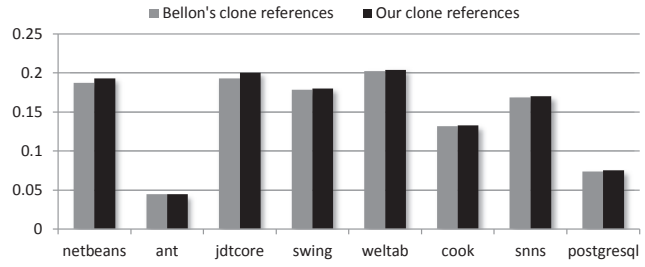
Figure 5 shows an example of our clone references. In Figure 5, the source file in the left has gapped lines 358-359. On the other hand, right one has no gapped lines.

If $recall$, $precision$ and $f\text{-}measure$ are calculated by using the clone references with the information of gapped lines, these values probably would be more precise. In the case of

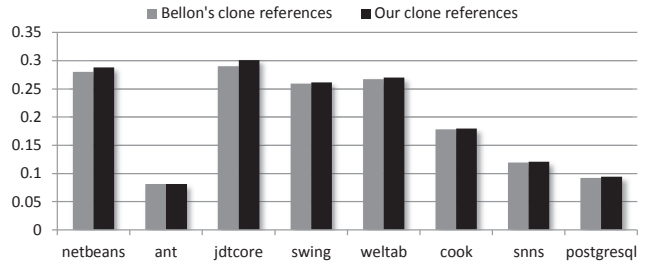
[1]http://sdl.ist.osaka-u.ac.jp/~h-murakm/new-reference/



(a) $Recall$



(b) $Precision$



(c) $F\text{-}measure$

Figure 6. $Recall$, $Precision$ and $F\text{-}measure$ of CDSW Using Both the Clone References.

Bellon's clone references, some gapped code clones contain gapped lines because Bellon's clone references have only the information about where code clones start and where they end. Meanwhile, in the case of our clone references, all the code clones do not contain gapped lines. In other words, our clone references consist of true code clones. Thus, evaluations using our clone references enable us to obtain true $recall$, $precision$ and $f\text{-}measure$.

We calculated $recall$, $precision$ and $f\text{-}measure$ using Bellon's and our clone references. Figure 6(a), (b) and (c) shows $recall$, $precision$ and $f\text{-}measure$ of the CDSW using both the clone references, respectively. For all of the software, $recall$, $precision$ and $f\text{-}measure$ were improved. In the best case, $recall$ increased by 4.1%, $precision$ increased by 3.7% and $f\text{-}measure$ increased by 3.8%. In the worst case, $recall$ increased by 0.49%, $precision$ increased by 0.42% and $f\text{-}measure$ increased by 0.43%.

Consequently we answer RQ 1 as follows. Calculating $recall$ and $precision$ using not only the information about where code clones start and where they end but also information about where the gaps are could evaluate code clones more precisely.

VII. Experiment B

One purpose of Experiment B is to reveal whether CDSW detects code clones more accurately than existing clone detectors or not. The other purpose is to reveal that CDSW detects code clones in practical time. In this experiment, we chose the clone detectors shown in Table II as targets for the comparison. All the clone detectors except NiCad and DECKARD were used in the experiment conducted by Bellon et al., and we calculated $recall$ and $precision$ of all the clone detectors by using our reference with information of gapped lines. All the clone detectors were used in their default configurations. Especially, we used NiCad 3.4 with block-based detection setting.

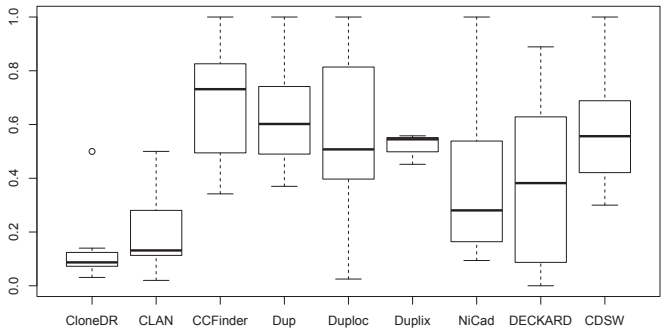
In Section III, we described that CDSW outputs gapped lines in code clones. However, if we use the outputs directly in this experiment, we could not make fair comparisons between CDSW and other clone detectors because they do not output gapped lines in code clones. Therefore, we only use information about where code clones start and where they end.

Figure 7(a) shows the $recall$ of all the clone detectors for only the Type 3 clone references. The median of CCFinder is the best in all the clone detectors, and that of Dup is the next. Coming third is CDSW. Figure 7(b) shows the case of $precision$. CLAN gets the first position, and CDSW gets the second. Figure 7(c) shows the case of $f\text{-}measure$. CDSW ranked first in this case, far surpassing all other clone detectors. To summarize above results, CDSW is not the best in the both case of $recall$ and $precision$. However, in the case of $f\text{-}measure$, CDSW is the best in all the clone detectors. In other words, CDSW achieves a good balance of $recall$ and $precision$ for the Type 3 clone references.

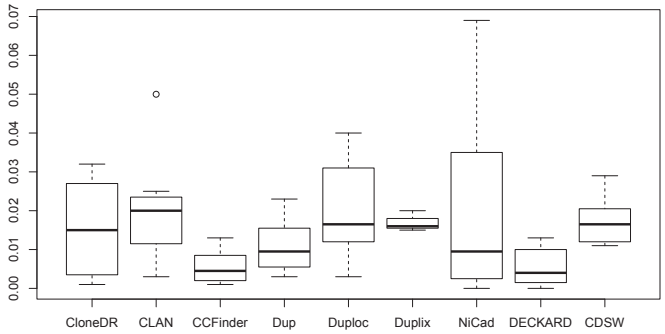
Figure 8(a) shows the $recall$ of all the clone detectors for Type 1, Type 2 and Type 3 clone references. The median of CDSW is the fifth position behind CCFinder, Dup, Duploc and DECKARD. Figure 8(b) shows the case of $precision$. CDSW is the best, and that of CLAN is the next by a mere

TABLE II
Clone detectors used for comparison

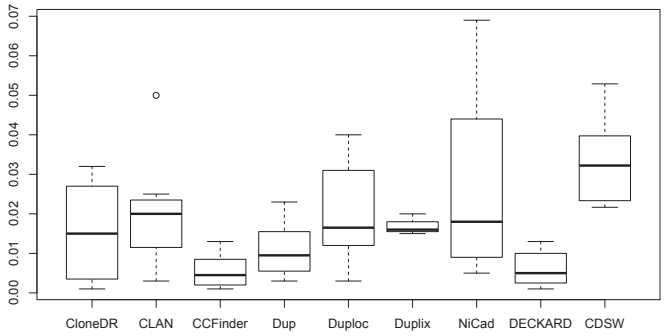
Developer	Clone detector	Detection method
Baker	Dup [9]	token-based
Baxter	CloneDR [10]	AST-based
Kamiya	CCFinder [11]	token-based
Merlo	CLAN [12]	metrics-based
Rieger	Duploc [13]	text-based
Krinke	Duplix [14]	PDG-based
Jiang	DECKARD [15]	AST-based
Roy	NiCad [16]	text-based using the LCS algorithm



(a) *Recall*

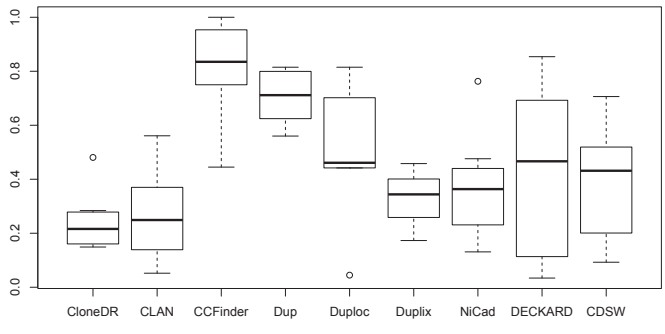


(b) *Precision*

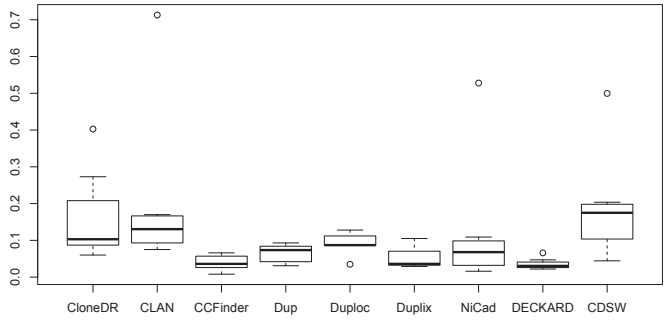


(c) *F-measure*

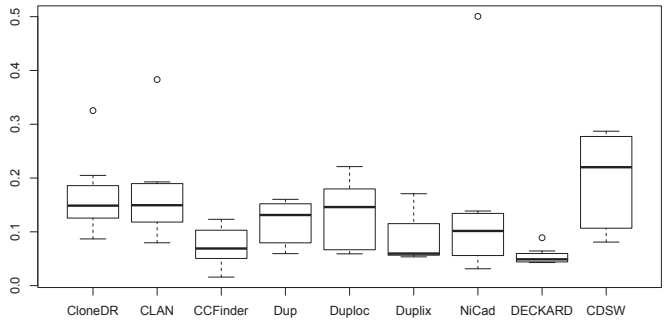
Figure 7. *Recall*, *Precision* and *F-measure* for the Type 3 Clone References.



(a) *Recall*



(b) *Precision*



(c) *F-measure*

Figure 8. *Recall*, *Precision* and *F-measure* for the Type 1, Type 2 and Type 3 Clone References.

touch. Figure 8(c) shows the case of *f-measure*. CDSW ranked first as is the case with only the Type 3 clone references. In short, CDSW achieves a good balance of *recall* and *precision* for not only the Type 3 clone references but also the Type 1 and Type 2 clone references.

We measured the execution time of CDSW. Figure 9 shows the execution time to detect code clones in target software systems. The execution time was calculated in lexical analysis phase (Step 1 and Step 2) and in detecting phase (Step 3, Step 4 and Step 5). CDSW could detect code clones in several seconds to about 30 seconds for all the target software systems. From Figure 9, it is also revealed that the Smith-Waterman algorithm has a significant influence on detecting process in terms of the execution time.

Moreover, we applied CDSW to the latest PostgreSQL (version 9.2.3, 839 files, 930,524 line of code). CDSW could detect code clones from the software in 7 minutes 30 seconds.

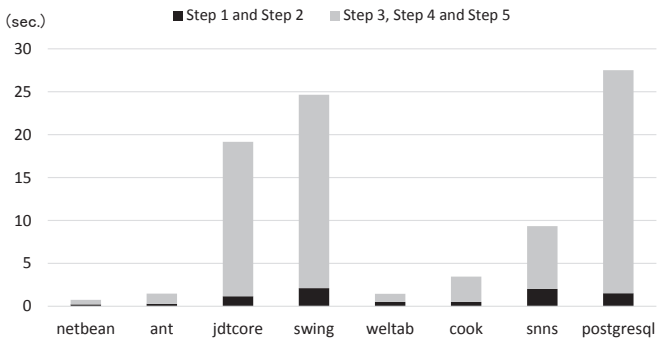


Figure 9. Exection Time of CDSW for the Target Software Systems.

Consequently we answer RQ 2 as follows. CDSW is the best in all the clone detectors used in Bellon's benchmark in the case of $f\text{-}measure$. Since $f\text{-}measure$ is harmonic average of $recall$ and $precision$, it would be said that CDSW has higher accuracy than the existing methods.

Besides, we answer RQ 3 as follows. CDSW could detect code clones from large scale software systems in a short time. In particular, it takes about 30 seconds for 200 KLOC software, and about 8 minutes for 1 MLOC software.

VIII. Threats to validity

A. Clone References

In these experiments, we compared the accuracy of CDSW and those of other clone detectors based on Bellon's clone references. However, they are not identified from all the code clones in the target software systems. Therefore, if all the code clones in the target software systems are used as clone references, we might obtain different results. However, it is almost impossible to validate clone references from all the code clones in the target software systems.

NiCad detects structural code clones such as block-based or function-based code clones. However, Bellon's clone references include many line-based code clones that are more fine-grained than block-based or function-based code clones. Thus, the result of NiCad might be bad excessively when using Bellon's benchmark.

B. Code Normalization

The proposed method replaces all variables and literals with specific token as a normalization. This means that the normalization ignores their types. If the proposed method uses more intelligent normalizations, for example, replacing them considering their type names, the number of detected code clones should be changed. At the same time, if the proposed method does not normalize source code, it cannot detect code clones that have differences of variable names or literals.

C. Three Parameters in the Smith-Waterman Algorithm

In this paper, we investigated appropriate parameters ($match$, $mismatch$, gap), then we compared the accuracy of CDSW to that of existing clone detectors. Thus, $match$, $mismatch$ and gap do not be changed while the clone detectors are identifying the code clones. If changing gap parameter constantly according to the length of code fragments, different results would be obtained.

IX. Discussion

A. The Smith-Waterman Algorithm vs. the LCS Algorithm

The Smith-Waterman algorithm is similar to the LCS algorithm. The LCS algorithm identifies global alignment from two sequences. On the other hand, the Smith-Waterman algorithm identifies local alignment. The largest difference between these two algorithms is that the Smith-Waterman algorithm uses $mismatch$ and gap parameters although the LCS algorithm does not use them. In other words, the Smith-Waterman algorithm can detect code clones in consideration for the information of their gapped lines. Moreover, the proposed method makes some changes to the Smith-Waterman algorithm as described in Section III. The changes enable the Smith-Waterman algorithm to detect one or more similar subsequences from two sequences. Therefore, the proposed method can perform a fine-grained detection.

If one sequence is $\langle a_1, a_2, \cdots, a_n \rangle$ and the other is $\langle b_1, b_2, \cdots, b_m \rangle$, the naive Smith-Waterman algorithm requires $O(mn)$ time and $O(mn)$ space. The LCS algorithm requires the same. Some low complexity strategies of the both algorithm were proposed [17].

Moreover, implementations of the Smith-Waterman algorithm on graphics processing units (GPUs) were proposed [18]. If we use GPUs for implementation of CDSW, the detection time would be reduced.

B. Related Work

There are many detection techniques of code clones using hash tables. One of the advantages of using hash tables is detection speed. Uddin et al. investigated the effectiveness of $simhash$ and adapted it to a code clone detection [19]. They showed that $simhash$ has significant potential for gapped code clone detection from large scale software systems quickly. Hummel et al. suggested an index-based approach for code clone detection [20]. They transformed several statements into hash values, then same hash values are identified in parallel. Their approach could detect code clones quickly from large scale software systems. However, Their approach did not tackle gapped code clones.

These days, some clone detectors are often embedded in workbench such as IDE (Integrated Development Environment). Bazrafshan et al. developed a code search system for identifies code fragments that are similar to a given code fragment [21]. The code search system is intended to identify defective code fragments. Zibran et al. presented an IDE-integrated clone search tool [22]. This tool enables programmers to identify code clones when they write code. Juergens et al. developed CloneDetective, a workbench for clone detection research [23]. By using CloneDetective, they confirmed that if the code clone gets changed inconsistently, such changes might represent faults [24].

Reducing the false positives in detection results is also studied. Koschke proposed a method to detect license violations of source code by using suffix trees [25]. His technique can detect code clones from large-scale software systems quickly. After code clones are detected, decision tree is used for filtering out false positives in order to improve $precision$. Murakami et al. focused on repeated instructions for reducing the number of false positives [26]. They proposed the method that folds repeated instructions. They showed that folding operation improves $precision$ in most cases.

X. Conclusion

This paper proposed a new method to detect not only contiguous but also gapped code clones by using the Smith-Waterman algorithm. The proposed method was developed as a software tool, CDSW. The authors investigated three parameters ($match$, $mismatch$, gap) used in the Smith-Waterman algorithm by conducting experiments for eight open source software systems. The appropriate tuples of ($match$, $mismatch$, gap) might work well for other software systems.

Furthermore, the authors remade the clone references used in Bellon's benchmark by adding information of gapped lines. CDSW was applied to eight open source software systems and calculated $recall$, $precision$ and $f\text{-}measure$ by using the clone references that the authors remade. The following items are confirmed.

- The authors tailored the Smith-Waterman algorithm for code clone detection.
- The accuracy of clone detection results improved by using not only the information about where code clones start and where they end but also information about where the gaps are.
- CDSW was the best in all the clone detectors used in Bellon's benchmark in the case of $f\text{-}measure$. Thus, CDSW achieved a good balance of true positives and false positives.
- CDSW detected code clones in a short time for large-scale software.

As described in Section VI, in this paper, the information of gapped lines that CDSW outputs were not used for accuracy comparison of clone detectors. In the future, the authors are going to conduct experiments using the information about where gaps are. If the information of gapped lines is used for evaluation, more accurate results would be obtained.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 21240002, 23650014, 24650011, 24680002, and 24700030.

References

[1] T. Kamiya, "Classifying code clones with configuration," in *Proc. of the 4th International Workshop on Software Clones*, 2010, pp. 75–76.

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

[3] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proc. of the 2004 International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[4] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proc. of the 33rd International Conference on Software Engineering*, 2011, pp. 311–320.

[5] "Clone detection literature," http://www.cis.uab.edu/tairasr/clones/literature/.

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2007.

[8] "Detection of software clones," http://www.bauhaus-stuttgart.de/clones/.

[9] B. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1343–1362, 1997.

[10] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. of the 14th International Conference on Software Maintenance*, 1998, pp. 368–377.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," *IEEE Trans. on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[12] J. Mayland, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. of the 12th International Conference on Software Maintenance*, 1996, pp. 244–253.

[13] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. of the 15th International Conference on Software Maintenance*, 1999, pp. 109–118.

[14] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. of the 8th Working conference on Reverse Engineering*, 2001, pp. 301–309.

[15] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard : Scalable and accurate tree-based detection of code clones," in *Proc. of the 29th International Conference on Software Engineering*, 2007, pp. 96–105.

[16] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. of the 16th International Conference on Program Comprehension*, 2008, pp. 172–181.

[17] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. of the Seventh International Symposium on String Processing Information Retrieval*, 2000, pp. 39–48.

[18] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the smith-waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[19] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proc. of the 18th Working Conference on Reverse Engineering*, 2011, pp. 13–22.

[20] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Indexbased code clone detection: incremental, distributed, scalable," in *Proc. of the 32th International Conference on Software Engineering*, 2010, pp. 1–9.

[21] S. Bazrafshan, R. Koschke, and N. Göde, "Approximate code search in program histories," in *Proc. of the 18th Working Conference on Reverse Engineering*, 2011, pp. 109–118.

[22] M. F. Zibran and C. K. Roy, "IDE-based real-time focused search for near-miss clones," in *Proc. of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 1235–1242.

[23] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *Proc. of the 31st International Conference on Software Engineering*, 2009, pp. 603–606.

[24] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. of the 31st International Conference on Software Engineering*, 2009, pp. 485–495.

[25] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 309–318.

[26] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Folding repeated instructions for improving token-based code clone detection," in *Proc. of the 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 64–73.