Master Thesis

Title

Type-3 Code Clone Detection Using The Smith-Waterman Algorithm

Supervisor
Prof. Shinji KUSUMOTO

by
Hiroaki MURAKAMI

February 5, 2013

Department of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

Type-3 Code Clone Detection Using The Smith-Waterman Algorithm

Hiroaki MURAKAMI

Abstract

Recently, code clones have received much attention. Code clones are defined as source code fragments that are identical or similar to each other. Code clones are introduced into source code by various reasons such as copy-and-paste operations. It is generally said that the presence of code clones makes software maintenance more difficult. This is because if we modify a code fragment, it is necessary to check its correspondents and verify whether they need the same modifications simultaneously or not.

In this thesis, we focus on code clones that is generated by copy-and-paste operations and then made modifications such as adding, deleting and changing statements. In order to detect such code clones, AST-based technique, PDG-based technique, metric-based technique and token-based technique using LCS can be used. However, each of these detection techniques has limitations. For example, existing AST-based techniques and PDG-based techniques require additional costs for transforming source files into intermediate representations such as ASTs or PDGs, and existing metric-based techniques and token-based techniques using LCS cannot detect code clones that locate in a part of modules. In this thesis, we propose a new detection method using the Smith-Waterman algorithm to resolve these limitations. The Smith-Waterman algorithm is an algorithm for identifying similar alignments between two sequences even if they include some gaps. We developed the proposed method as a software tool and confirmed that the proposed method could resolve the limitations by conducting a quantitative evaluation of our tool with Bellon's benchmark.

Keywords

Code Clones
Program Analysis
Software Maintenance
Tool Comparison

Contents

List of Figures

List of Tables

1 Introduction

Recently, code clones have received much attention. Code clones are identical or similar code fragments to one another in source code. Code clones are generated by various reasons such as copy-and-paste operations. Bellon et al. categorized code clones into following three types by the degree of their similarities [1].

Type-1 is an exact copy without modifications (except for white space and comments).

Type-2 is a syntactically identical copy; only variable, type, or function identifiers were changed.

Type-3 is a copy with further modifications; statements were changed, added, or removed.

A number of techniques detecting Type-3 code clones have been proposed before now [2]. The detection techniques can be categorized as follows.

AST (Abstract Syntax Tree)-based technique

After generating ASTs from source files, subtrees having the same structure are detected as code clones. One of the disadvantages when using this technique is that it requires additional costs for transforming source files into ASTs and identifying common subtrees.

PDG (Program Dependency Graph)-based technique

After building PDGs from source files, isomorphic subgraphs are detected as code clones. One of the disadvantages when using this technique is that it requires similar additional costs as AST-based technique.

Metric-based technique

After calculating metric values on every program module (such as classes or methods), modules which have the same or almost the same metric values are regarded as code clones. This technique can detect code clones quickly. However, this technique cannot find code clones lying in a part of a class or a method.

Token-based technique using LCS (Longest Common Subsequence) algorithm

After using LCS algorithm to token sequences transformed from every pair of blocks or functions, token sequences whose similarities are larger than a threshold are regarded as code clones. This technique can detect code clones quickly as well as the metric-based technique. However, this technique cannot find code clones lying in a part of a block or a function.

Each of those detection techniques has limitations as described previously. In order to resolve these limitations, we propose a detection method using the Smith-Waterman algorithm [3]. The proposed method detects not only Type-1 and Type-2 but also Type-3 code clones at lower cost 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 or LCS techniques cannot detect because these techniques perform coarse-grained detections such as method-level or block-level. On the other hands, the proposed method performs a fine-grained detection that identifies sentence-level code clones.

We implemented the proposed method and evaluated it by using Bellon's benchmark [1]. However, Bellon's benchmark has a limitation that the reference of Type-3 code clones does not have the information where gaps are. Bellon's reference represents code clones with only the information where they start and where they end. We do not consider that gapped parts of code clones should be regarded as a part of code clones. Thus, Bellon's benchmark is likely to evaluate Type-3 code clones incorrectly when it is used as-is. Therefore, we remade the reference of code clones with the information where gaps are. Moreover, we compared the result by using Bellon's reference with that by using our reference.

Consequently, the contributions of this thesis are as follows:

- We confirmed that evaluating $precision$ and $recall$ by using the information where gaps are had higher accuracy than using only the information where code clones start and where they end.
- We confirmed that the proposed method had high accuracy and finished detecting code clones in practical time.

The rest of this thesis is organized as follows: In Section 2, we introduce preliminaries related to this work. Section 3 describes the outline of the proposed method. We describe the overview of investigation in Section 4, then Section 5 and Section 6 report the evaluations of the proposed method in detail. In section 7, we describe threats to validity. Finally, Section 8 summarizes this thesis and refers to the future work.

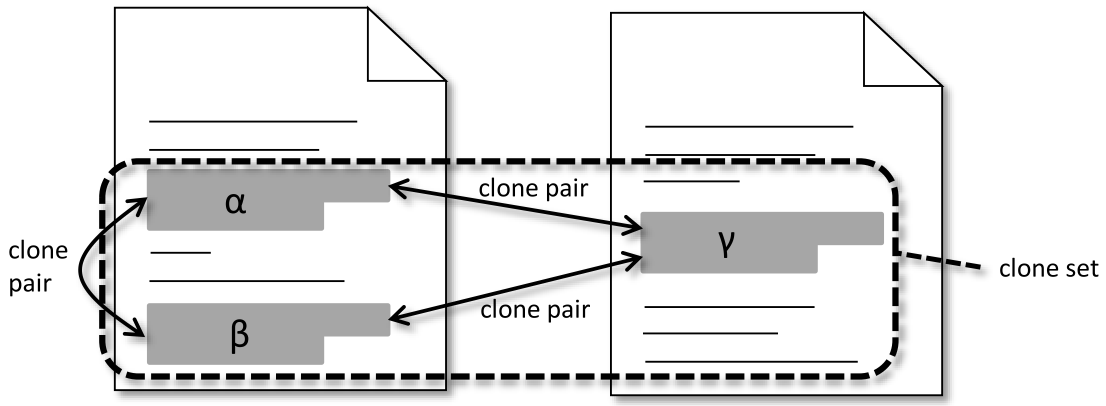


Figure 1: An Example of Clone Pairs and Clone Sets

2 Preliminaries

2.1 Code Clone

2.1.1 Definition

Code clone is defined as identical or similar code fragments in source code. As shown in Figure 1, we call a pair of code fragments α and β as a clone pair if α and β are similar. In addition, we call a set of code fragments S as a clone set if any pair of code fragments in S are clone pairs [4]. Note that there is neither a generic nor strict definition of code clone, therefore each clone detection method or tool has its own definition of code clone.

2.1.2 Causes of Creations

Code clones can be created or introduced by the following factors.

Copy-and-paste Operations

This is the most popular situation that code clones are created. The code reuse by copy-and-paste operations is a common practice in software development, because it is quite easy, and it enables us to make software development faster.

Stylized Processing

Processing used frequently (e.g. calculations of the income tax, insertions in queues, or access to data structures) may cause code duplication.

Lack of Suitable Functions

Programmers may have to write similar processes with similar algorithms if they use programming languages that do not have abstract data types or local variables.

Performance Improvement

Programmers can introduce code duplication intentionally to improve the performance of software systems in the case that the in-line expansion is not supported.

Automatically Generated Code

Code generation tools automatically create code based on stylized code. As a result, if we use code generation tools to handle similar processes, it may generate similar code fragments.

To Handle Multiple Platforms

Software systems that can handle multiple operation systems or CPUs tend to include many code clones in the processes handling each platform.

Accident

Different developers may write similar code accidentally. However, it is rare that the amount of similar code generated accidentally becomes high.

2.2 Code Clone Detection Methods

There are many methods that detect code clones automatically, and there are also many code clones detectors implementing these methods. Code clones detectors can be loosely categorized into the following categories by their detection units [5, 1].

2.2.1 Text-based Techniques

Text-based detection techniques detect code clones by comparing every line of code as a string. They detect multiple consecutive lines that match in specified threshold or more lines as code clones. The biggest advantage of this technique is that it can detect code clones quickly compared with other detection techniques. This technique requires no pre-processing on source code, which enables the fast detection. However, we cannot detect code clones including differences of coding styles (e.g. whether long lines are divided into multiple lines or not) with this technique.

The method proposed by Johnson [6] and the method proposed by Ducasse et al. [7] are instances of line-based clone detectors. In these methods, every line of code is compared after white space and tabs are removed. These methods are language-independent because they compare lines of code textually.

2.2.2 Token-based Techniques

In a token-based approach, source code is lexed/parsed/transformed to a sequence of tokens. This technique detects common subsequences of tokens as code clones. Compared to text-based approaches, a token-based approach is usually robuster against code changes such as formatting and spacing. Detection speed is inferior as compared with text-based techniques, meanwhile superior as Tree- or PDG-based approaches. This is because, in token-based approach, source code has to be transformed into intermediate representations such as AST and PDG.

CCFinder, a clone detector developed by Kamiya et al. [8], is one of the token-based detectors. CCFinder replaces user-defined identifiers by special tokens. By this pre-processing, it can detect code clones with different identifiers. In addition, it can handle multiple widely-used programming languages such as C/C++, Java, COBOL, and FORTRAN. Moreover, there is a major version up of CCFinder named CCFinderX [9]. In this version up, the detection algorithm is changed, and the detection speed is improved by multithreading.

CP-Miner is also a token-based detector. CP-Miner is developed by Li et al. [10]. Firstly, lexical and syntax analyses are performed on source code. User-defined identifiers are replaced by special tokens as well as CCFinder. The major difference between CP-Miner and CCFinder is in detection algorithms. In CP-Miner, hash values are calculated from every statement, and then a frequent pattern mining algorithm is applied to detect code clones. Frequent patterns do not have to be consecutive, which means that CP-Miner can detect Type-3 clones.

2.2.3 Tree-based Techniques

In a Tree-based detection, a program is parsed to a parse tree or an abstract syntax tree (in short, AST) with a parser of the language in interest. An AST is one of the intermediate representations that capture the structure of source code. Figure 2 shows an example of ASTs. Common subtrees are regarded as code clones. This approach considers the structural information of source code, therefore tree-based detectors do not detect code clones ignoring the structure of source code such as code clones including a part of a method and a part of another method. However, a disadvantage of this approach compared with Text- and Token-based approaches is that it requires more detection costs because of the additional cost required to transform source code to parse trees or ASTs.

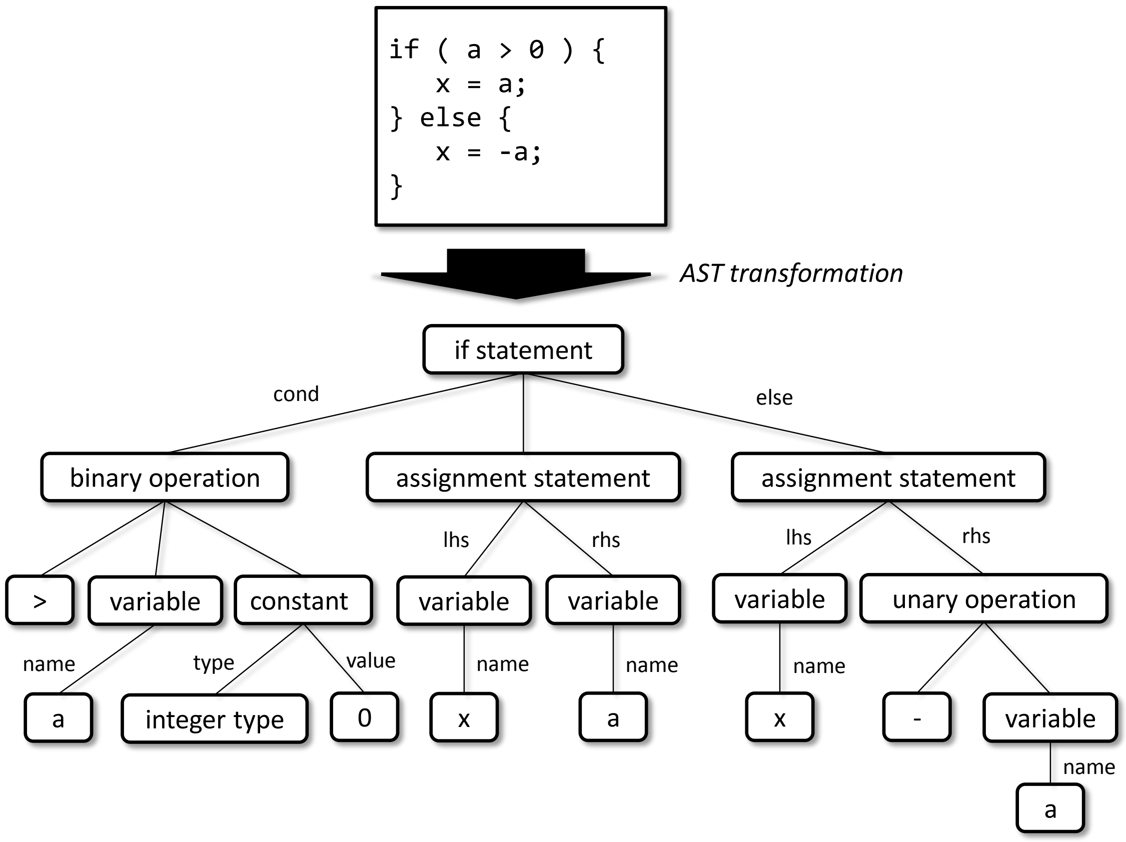


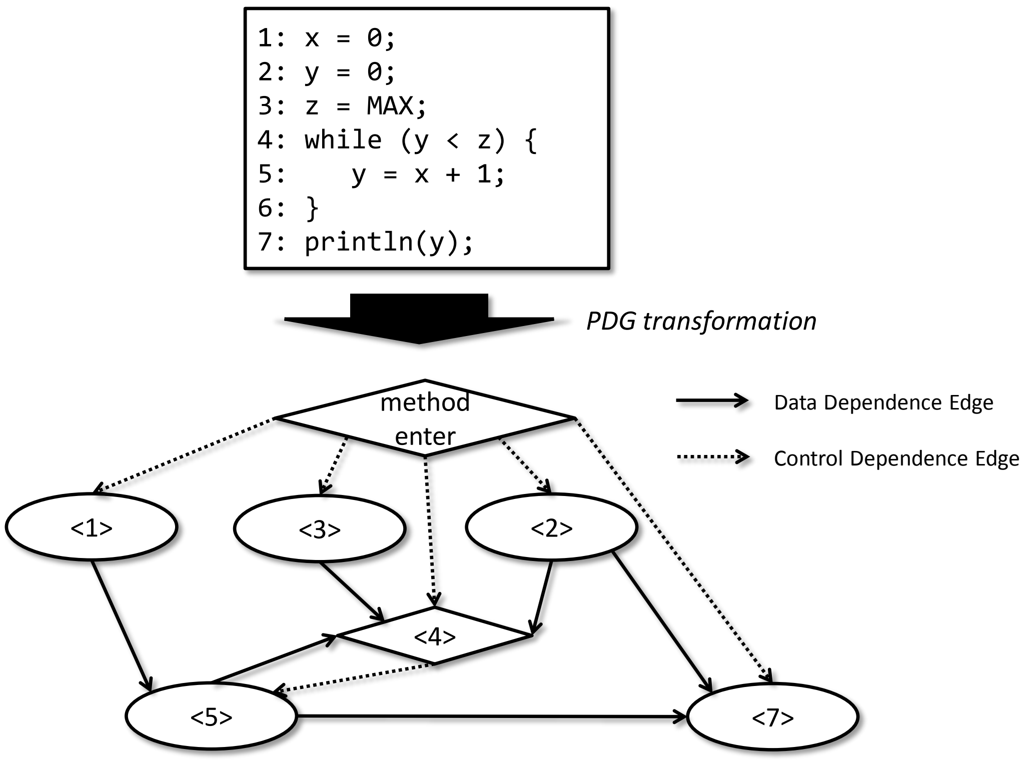
Figure 2: An Example of ASTs.

One of the pioneers of AST-based clone techniques is that of Baxter et al.'s CloneDR [11, 12]. CloneDR compares subtrees of ASTs by characterization metrics based on a hash function through tree matching, instead of comparing subtrees of ASTs directly. This processing allows CloneDR to detect code clones quickly from large software systems. It can handle a lot of programming languages. Moreover, it has a function to assist clone removal.

Koschke et al.'s method [13] and Jiang et al.'s method [14] are tree-based approaches as well as CloneDR. In Koschke et al.'s method, ASTs are compared with a suffix tree algorithm to have an increase of detection speed. On the other hand, Jiang et al. use a locality sensitive hashing algorithm to detect code clones. With the algorithm, Jiang et al.'s method can detect Type-3 code clones.

2.2.4 PDG-based Techniques

In a PDG-based approach, code clones are detected by comparing PDGs created from source code. Figure 3 shows an example of PDGs. Isomorphic subgraphs are regarded as code clones.



```
1: x = 0;
2: y = 0;
3: z = MAX;
4: while (y < z) {
5:    y = x + 1;
6: }
7: println(y);
```

Figure 3: An Example of PDGs.

PDGs require a semantic analysis for their creation, therefore this approach requires much cost than other detection techniques. However, this technique can detect code clones with additions/deletions/changes in statements or those with some differences that have no impact on the behavior of programs. This is because PDG-based techniques can consider the meanings of programs.

Figure 4 shows one of the code clones that include some differences that have no impact on the behavior of programs. Other techniques cannot detect these two code fragments as a code clone because there is a different order of statements.

One of the leading PDG-based clone detection approach is Komondoor and Horwitz's method [15]. Their method detects isomorphic subgraphs of PDGs with program slicing. They also propose an approach to group identified clones together while preserving the semantics of the original code for automatic procedure extraction to support software refactoring. Krinke's method [16], and Higo et al.'s method [17, 18] are also included in PDG-based techniques. Each detection method is optimized to reduce detection cost. Krinke sets a limit of the search range of PDGs with a threshold. By contrast, Higo et al. confine nodes to be base of subgraphs with some conditions. Moreover, Higo et al. introduce a new dependence named "execution dependence". That is, there

```
  fp = lookaheadset + tokensetsize;
  for (I = lookaheas(state) ; I < k ; i++) {
%     fp1 = LA + i * tokensetsize;
%     fp2 = lookaheadset;
%     while (fp2 < fp3)
%           *fp2++ |= fp1++;
  }
```

(a) Code Fragment 1

```
  fp3 = base + tokensetsize;
  …
  if (rp) {
      while ((j = *rp++) >= 0) {
          …
#             fp1 = lookaheadset;
#             fp2 = LA + j * tokensetsize;
#             while (fp1 < fp3)
#                   *fp1++ |= *fp2++;
      }
  }
```

(b) Code Fragment 2

Figure 4: A Code Clone with a Differnt Order of Statements

is an execution dependence from a node A to another node B if the program element represented by B may only be executed after the program element represented by A. By introducing this dependence, they succeeded to detect code clones that other PDG-based methods could not detect.

2.2.5 Other Detection Techniques

One of the detection techniques that can be categorized into this category is a metrics-based approach [19]. First, metrics-based detectors calculate metrics on every program module (such as files, classes, or methods), then detect code clones by comparing the coincidence or the similarity of these values.

Beside this, there are some file-based detection methods [20, 21]. This detection technique detects code clones by comparing every file instead of statements or tokens, which let it quick detections. However, this technique cannot find code clones that exist in a part of a file.

Moreover, incremental detection techniques are under intense studies [22, 23, 17]. In incremental detections, code clone detection results or their intermediate products persist by using databases, and it is used in the next code clone detection. By reusing previous revisions' analysis, it can reduce detection cost on the current revision substantially.

2.3 The Smith-Waterman Algorithm

The Smith-Waterman algorithm [3] is an algorithm for identifying similar alignments between two base sequences. This algorithm has an advantage that it identifies similar alignments even if they include some gaps. Figure 5 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 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 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 table): scores of all the remaining cells are calculated by using the following formula.

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

$$s(a_i, b_j) = \begin{cases} 1 & (a_i = b_j), \\ -1 & (a_i \neq b_j). \end{cases} \tag{2}$$

$$g = -1. \tag{3}$$

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; g is a gap-penalty parameter; and $s(a_i, b_j)$ is a score parameter 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.

While calculating 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 5, $v_{9,11}(= 5)$ is calculated by adding $v_{8,10}(= 4)$ and $s(v_{0,11}, v_{9,0})(= 1)$. In this case, the pointer from $c_{8,10}$ to $c_{9,11}$ is created.

Step D (traceback table): traceback means 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 Step D, the hatched cells with numbers represent the traceback path. The array elements pointed by the traceback path are similar local alignments(“ACGACAACT” and “ACACACT”).

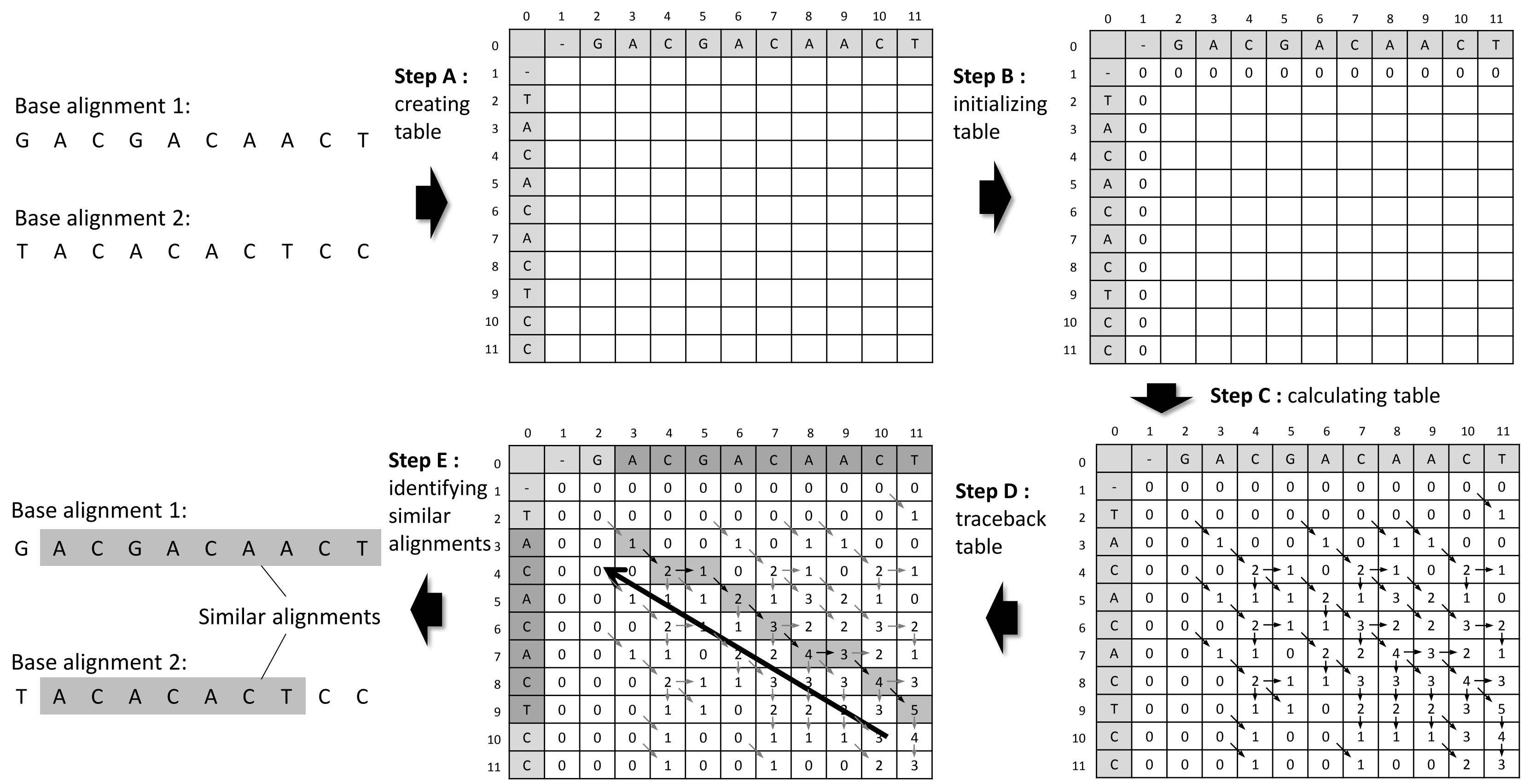


Figure 5: The Smith-Waterman Algorithm Applied to Two Base Sequences, “GACGACAACT” and “TACACACTCC”.

3 Proposed Method

The proposed method takes the followings as its inputs:

- **source files**,
- **minimal clone length** (number of tokens),
- **maximal gap rate** (ratio of gapped tokens in the detected tokens).

The proposed method outputs a list of detected clone pairs. Scoring parameter $s(a_i, b_j)$ and gap-penalty parameter g use the same equations (2), (3) described in Section 2.3. The proposed method consists of the following four steps.

Step 1: performing lexical analysis and normalization

Step 2: generating statement hash

Step 3: identifying similar hash sequences

Step 4: mapping identical subsequences to source code

Figure 4 shows an overview of the proposed method. Figure 7 shows an example of detection process using proposed method. The remainder of this section explains every of the steps in detail.

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.

Step 2: generating statement hash

A hash value is generated from 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.

Step 3: identifying similar hash sequences

Similar hash sequences are identified from hash sequences generated in Step 2 by using the Smith-Waterman algorithm. Herein, we make following changes to Step D described in section 2.3 for detection of code clones.

- Traceback begins at multiple cells in order to detect two or more 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 following characteristics are selected as start cells of traceback.

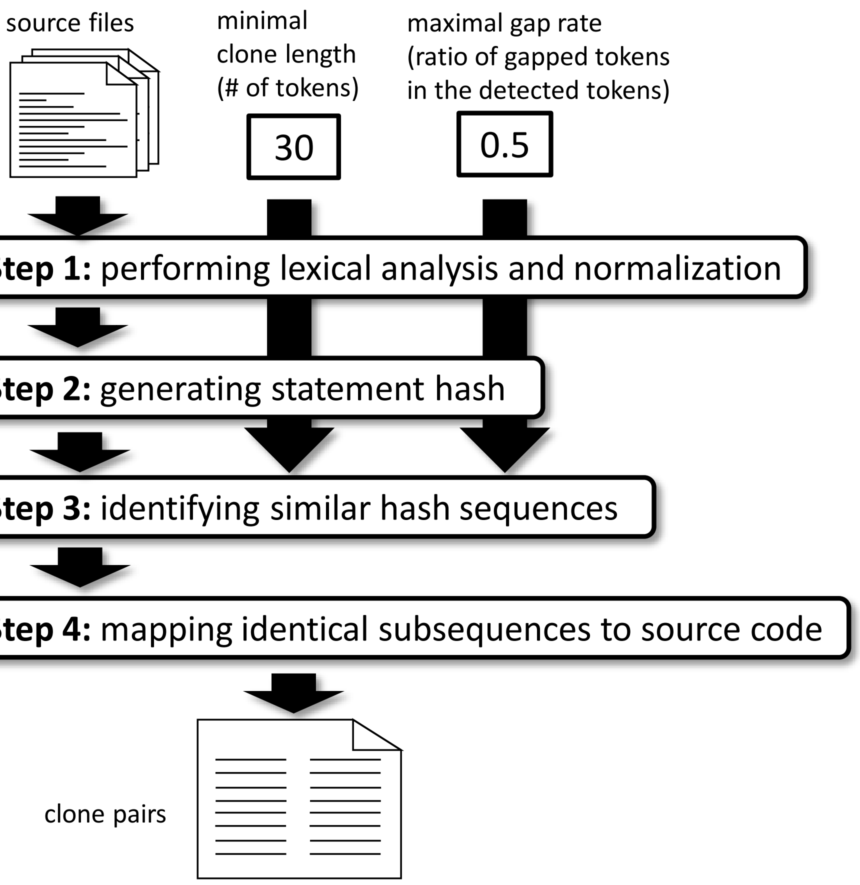

source files
minimal
clone length
(# of tokens)
maximal gap rate
(ratio of gapped tokens
in the detected tokens)
30
0.5
Step 1: performing lexical analysis and normalization
Step 2: generating statement hash
Step 3: identifying similar hash sequences
Step 4: mapping identical subsequences to source code
clone pairs

Figure 6: Overview of the Proposed Method.

- $v_{i,j} > 0$
- $v_{i,0} = v_{0,j}$

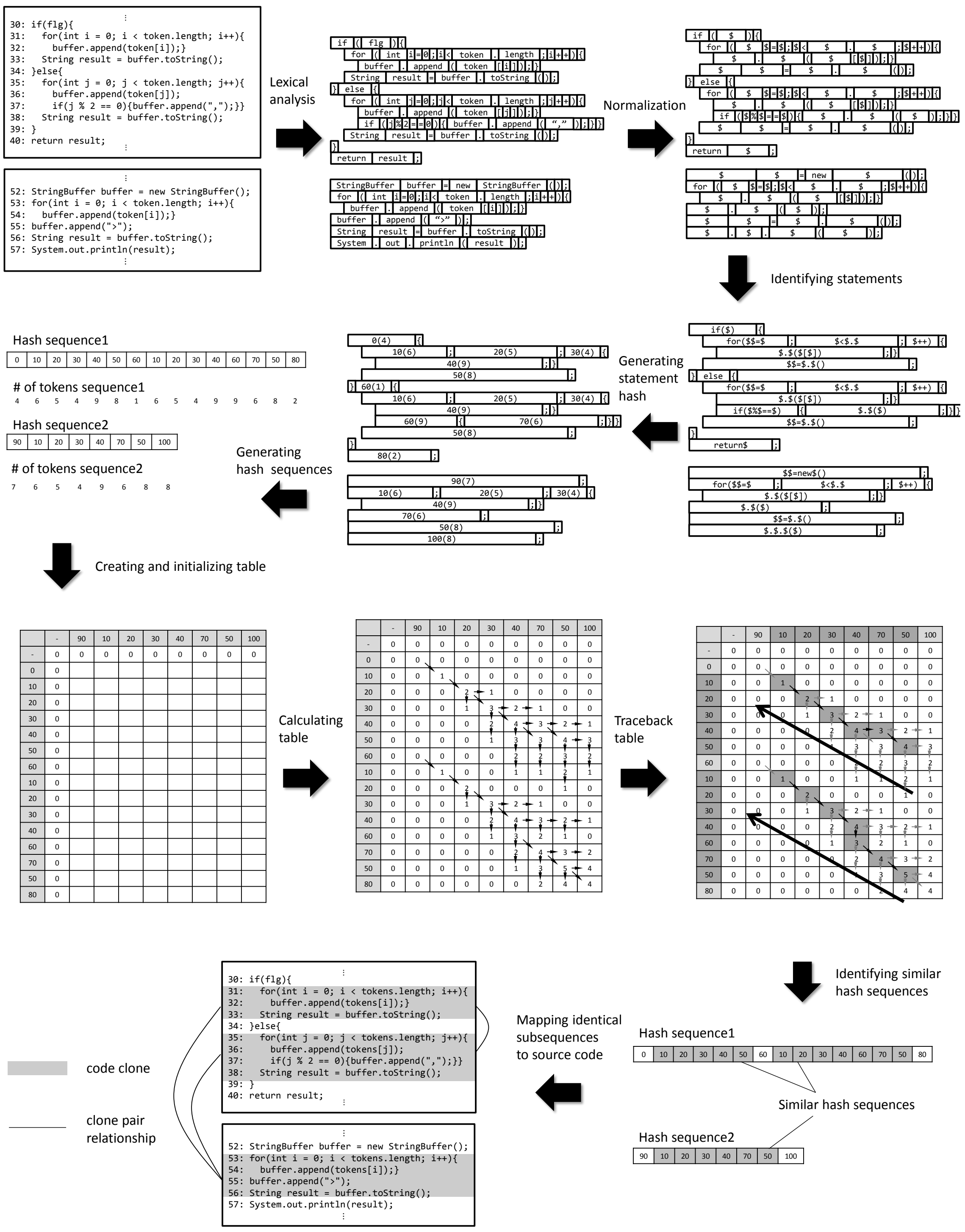
Moreover, if the start cell is $c_{i,j}$ and the end cell is $c_{k,l}$ ($k \leq i, l \leq j$), the cells included in the following set S will be out of scope from all the traceback following the current traceback in order not to detect redundant code clones.

$$S = \{c_{m,n} | k \leq m \leq i \land l \leq n \leq j\} \tag{4}$$

- The number of tokens and gaps are counted while tracebacking in order to detect code clones whose token length is greater than a minimal clone length and the ratio of gapped tokens in the detected tokens is less than a maximal gap rate.

Step 4: mapping identical subsequences to source code

Identified subsequences detected in Step 3 are converted to location information in the source code (file path, start line, end line and gapped lines), which are clone pairs.


Lexical analysis
Normalization
Identifying statements
Generating statement hash
Generating hash sequences
Creating and initializing table
Calculating table
Traceback table
Identifying similar hash sequences
Similar hash sequences
Mapping identical subsequences to source code
code clone
clone pair relationship

```
30: if(flg){
31:    for(int i = 0; i < token.length; i++){
32:      buffer.append(token[i]);}
33:    String result = buffer.toString();
34: }else{
35:    for(int j = 0; j < token.length; j++){
36:      buffer.append(token[j]);
37:      if(j % 2 == 0){buffer.append(",");}}
38:    String result = buffer.toString();
39: }
40: return result;
```

```
52: StringBuffer buffer = new StringBuffer();
53: for(int i = 0; i < token.length; i++){
54:    buffer.append(token[i]);}
55: buffer.append(">");
56: String result = buffer.toString();
57: System.out.println(result);
```

Hash sequence1

0	10	20	30	40	50	60	10	20	30	40	60	70	50	80

# of tokens sequence1

4 6 5 4 9 8 1 6 5 4 9 9 6 8 2

Hash sequence2

90	10	20	30	40	70	50	100

# of tokens sequence2

7 6 5 4 9 6 8 8

	-	90	10	20	30	40	70	50	100
-	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
10	0	0	1	0	0	0	0	0	0
20	0	0	0	2	1	0	0	0	0
30	0	0	0	1	3	2	1	0	0
40	0	0	0	0	2	4	3	2	1
50	0	0	0	0	1	3	3	4	3
60	0	0	0	0	0	2	2	3	2
10	0	0	1	0	0	1	1	2	1
20	0	0	0	2	0	0	0	1	0
30	0	0	0	1	3	2	1	0	0
40	0	0	0	0	2	4	3	2	1
60	0	0	0	0	1	3	2	1	0
70	0	0	0	0	0	2	4	3	2
50	0	0	0	0	0	1	3	5	4
80	0	0	0	0	0	0	2	4	4

```
30: if(flg){
31:    for(int i = 0; i < tokens.length; i++){
32:      buffer.append(tokens[i]);}
33:    String result = buffer.toString();
34: }else{
35:    for(int j = 0; j < tokens.length; j++){
36:      buffer.append(tokens[j]);
37:      if(j % 2 == 0){buffer.append(",");}}
38:    String result = buffer.toString();
39: }
40: return result;
```

```
52: StringBuffer buffer = new StringBuffer();
53: for(int i = 0; i < tokens.length; i++){
54:    buffer.append(tokens[i]);}
55: buffer.append(">");
56: String result = buffer.toString();
57: System.out.println(result);
```

Figure 7: An Example of Detection Process Using the Proposed Method.

4 Overview of Investigation

We have developed a software tool, CDSW based on the proposed method described in Section 3. We have conducted two investigations to answer the following research questions.

RQ 1: Does evaluating $precision$ and $recall$ by using the information where gaps are have higher accuracy than using only the information where code clones start and where they end?

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

RQ 3: Does the proposed method finish detecting code clones in practical time?

Experiment A investigates RQ 1. Experiment B investigates RQ 2 and RQ 3.

In order to calculate $recall$ and $precision$, we need the relevant code clones. Herein we use freely available code clone data in literature [24] as a reference set (a set of code clones to be detected). The reference set includes code clones information of eight software systems. Table 1 shows an overview of the target systems. In the remainder of this thesis, we use the following terms.

Clone candidates: code clones detected by clone detectors.

Clone references: code clones included in the reference.

We use the ok value to decide whether every clone candidates matches any of the clone references or not. In this investigation, We use 0.7 as the threshold, which is the same value used in literature [1]. We calculate $recall$ and $precision$ for evaluating the detection capability. Assume that R is a detection result, 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 is equal to or greater than the

Table 1: Target Software Systems

Name	Short name	Language	Lines of code	# of references
netbeans-javadoc [25]	netbeans	Java	14,360	55
eclipse-ant [26]	ant	Java	34,744	30
eclipse-jdtcore [26]	jdtcore	Java	147,634	1,345
j2sdk1.4.0-javax-swing [27]	swing	Java	204,037	777
weltab	weltab	C	11,460	275
cook [28]	cook	C	70,008	440
snns [29]	snns	C	93,867	1,036
postgresql [30]	postgresql	C	201,686	555

threshold in R. $Recall$ and $precision$ are defined as follows:

$$Recall = \frac{|S_R|}{|S_{refs}|}. \tag{5}$$

$$Precision = \frac{|S_R|}{|R|}. \tag{6}$$

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 of $precision$ does not directly indicate that the detection result includes many false positives. A low value means that there are many clone candidates are not matching any of the clone references; however, nobody knows whether they are truly false positives or not.

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

The details of each experiment are described in Section 5 and 6, respectively.

5 Experiment A

The purpose of Experiment A is to reveal how $precision$ and $recall$ are changed by our defined formula. In Bellon's benchmark [1], in order to determine whether a candidate matches a reference, $overlap(CF_1, CF_2)$, $contained(CF_1, CF_2)$, $good(CP_1, CP_2)$ and $ok(CP_1, CP_2)$ are used, where CP_1 and CP_2 are clone pairs, CF_1 and CF_2 are code fragments.

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.

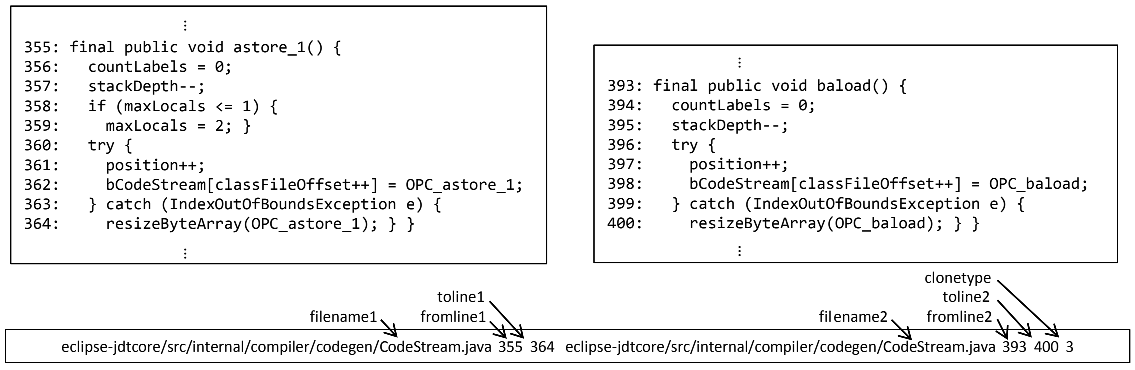
Figure 8 shows an example of Bellon's clone references, and Figure 9 shows an example of our clone references. In Figure 9, left source file has gapped lines 358-359. On the other hand, right one has no gapped lines.

If $recall$ and $precision$ are calculated by using the clone references with the information of gapped lines, these values probably would be more precise. In the case of Bellon's clone references, some Type-3 code clones contain gapped lines because Bellon's clone references have only the information of 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 $recall$ and $precision.$

We calculated $recall$ and $precision$ using Bellon's and our clone references. Figure 10 and Figure 11 show the $recall$ and $precision$ of the CDSW using Bellon's and our clone references. For all of the software, $precision$ and $recall$ were improved. In the best case, $recall$ increased by 4.6 % and $precision$ increased by 4.4 %. In the worst case, $recall$ increased by 0.24 % $precision$ increased by 0.21 %.

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

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

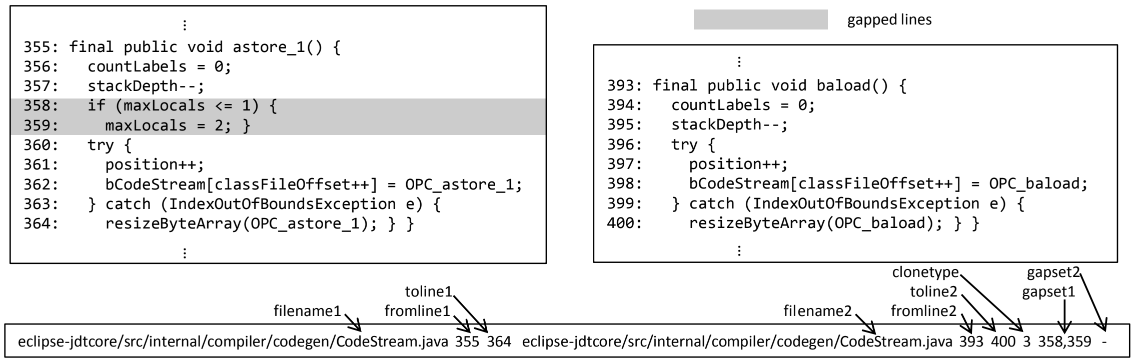


```
          ⋮
355: final public void astore_1() {
356:   countLabels = 0;
357:   stackDepth--;
358:   if (maxLocals <= 1) {
359:     maxLocals = 2; }
360:   try {
361:     position++;
362:     bCodeStream[classFileOffset++] = OPC_astore_1;
363:   } catch (IndexOutOfBoundsException e) {
364:     resizeByteArray(OPC_astore_1); } }
          ⋮
```

```
          ⋮
393: final public void baload() {
394:   countLabels = 0;
395:   stackDepth--;
396:   try {
397:     position++;
398:     bCodeStream[classFileOffset++] = OPC_baload;
399:   } catch (IndexOutOfBoundsException e) {
400:     resizeByteArray(OPC_baload); } }
          ⋮
```

```
eclipse-jdtcore/src/internal/compiler/codegen/CodeStream.java 355 364 eclipse-jdtcore/src/internal/compiler/codegen/CodeStream.java 393 400 3
```

Figure 8: An Example of Bellon's Clone Reference (Clone Reference No. 1101).



```
          ⋮
355: final public void astore_1() {
356:   countLabels = 0;
357:   stackDepth--;
358:   if (maxLocals <= 1) {
359:     maxLocals = 2; }
360:   try {
361:     position++;
362:     bCodeStream[classFileOffset++] = OPC_astore_1;
363:   } catch (IndexOutOfBoundsException e) {
364:     resizeByteArray(OPC_astore_1); } }
          ⋮
```

```
          ⋮
393: final public void baload() {
394:   countLabels = 0;
395:   stackDepth--;
396:   try {
397:     position++;
398:     bCodeStream[classFileOffset++] = OPC_baload;
399:   } catch (IndexOutOfBoundsException e) {
400:     resizeByteArray(OPC_baload); } }
          ⋮
```

```
eclipse-jdtcore/src/internal/compiler/codegen/CodeStream.java 355 364 eclipse-jdtcore/src/internal/compiler/codegen/CodeStream.java 393 400 3 358,359 -
```

Figure 9: An Example of the Clone Reference We Remade (Clone Reference No. 1101).

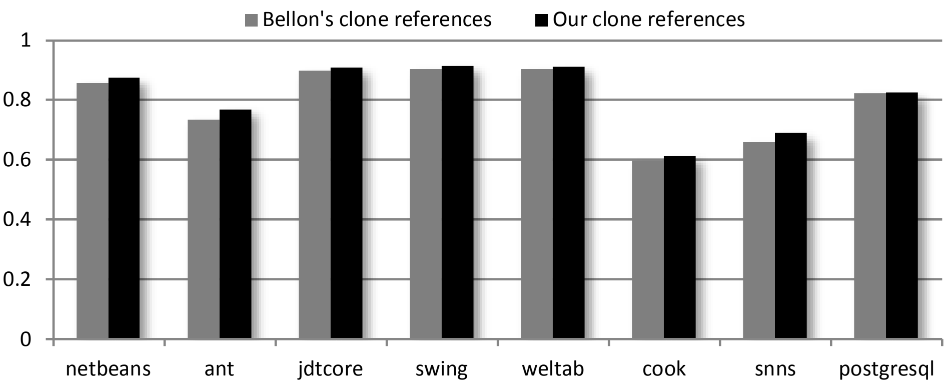


Figure 10: *Recall* of CDSW for the Target Software Systems Calculated by Bellon's and Our Clone References.

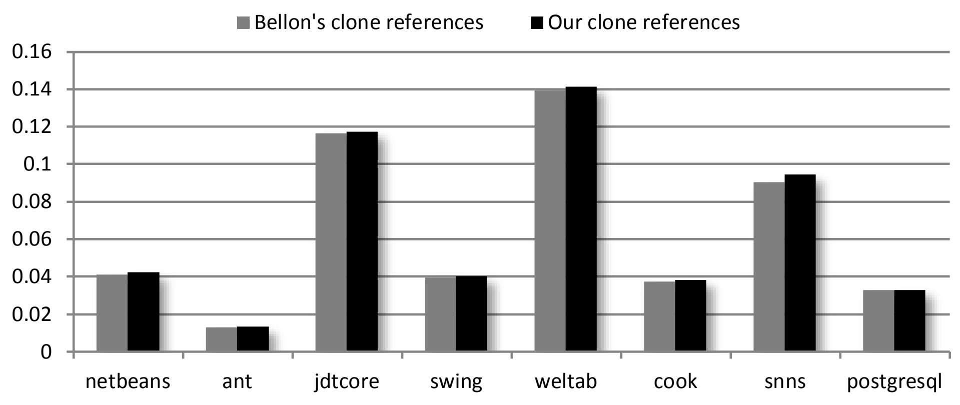


Figure 11: $Precision$ of CDSW for the Target Software Systems Calculated by Bellon's and Our Clone References.

6 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 2 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. In section 3, 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 the information where code clones start and where they end.

Figure 12 shows the $recall$ of all clone detectors for the Type-3 clone references. The median of CDSW is the best in all the clone detectors. Figure 13 shows the $precision$ of all clone detectors for the Type-3 clone references. The median of CLAN is the best, and that of CDSW is the middle position. Figure 14 shows the $recall$ of all clone detectors for the Type-1 and Type-2 clone references. The median of CCFinder is the best, and that of CDSW is the next. Figure 15 shows the $precision$ of all clone detectors for the Type-1 and Type-2 clone references. The median of CLAN is the best, and that of CloneDR is the next. The medians of the remaining detectors are almost the same.

We compared the execution time of the existing tool and CDSW. Herein, we selected DECKARD as a representative of the existing tools because DECKARD are widely used, easy to use and detects code clones using intermediate representation. Figure 16 shows the execution time of DECKARD and CDSW. DECKARD detect code clones from the target software systems in about 22 minutes to 2 hours and a half. Meanwhile, CDSW detect code clones from the target software

Table 2: Tools Used for Comparison

Developer	Tool	Detection method
Baker	Dup [31]	token-based
Baxter	CloneDR [11]	AST-based
Kamiya	CCFinder [8]	token-based
Merlo	CLAN [32]	metrics-based
Rieger	Duploc [33]	line-based
Krinke	Duplix [16]	PDG-based
Jiang	DECKARD [14]	AST-based
Roy	NiCad [34]	token-based using LCS

systems in a few seconds to a few minutes. Moreover, We applied CDSW to large-scale software JDK 7 java packages (1,631 files and 660,698 lines of codes). CDSW could detect code clones from JDK 7 in about 30 minutes.

Consequently we answer RQ 2 as follows: CDSW could detect as many clone references as token-based clone detectors. Especially, CDSW could detect the most Type-3 clone references. However, it detected many clone candidates as well as other clone detectors. Therefore, it might detect many false positives.

Besides, we answer RQ 3 as follows: CDSW could detect code clones in a practical time on software systems even if they have over six hundred thousand lines.

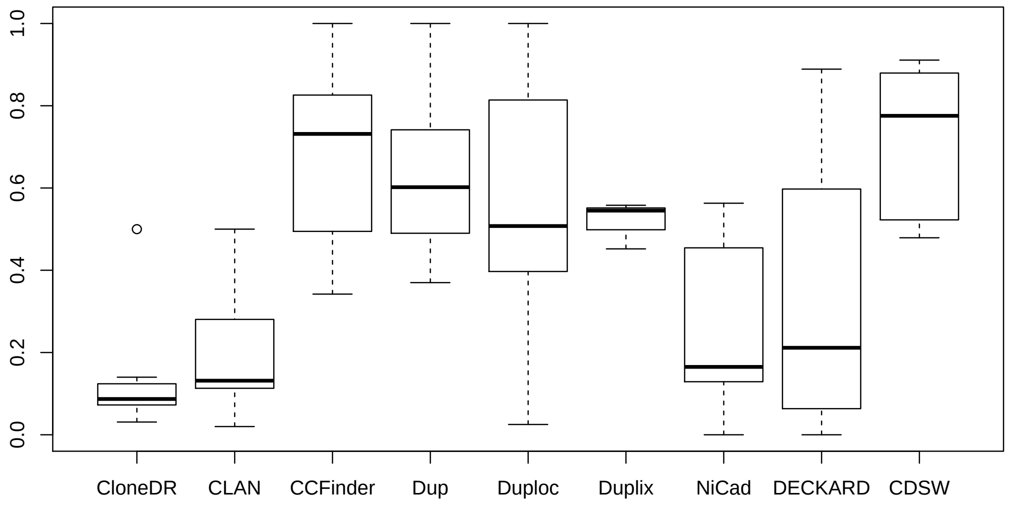


Figure 12: $Recall$ of All Clone Detectors for the Type-3 Clone References.

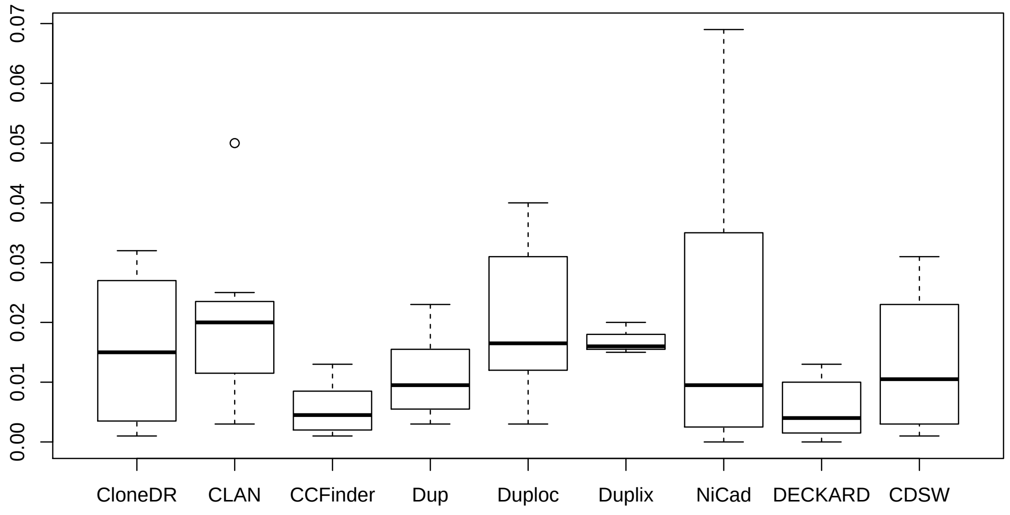


Figure 13: $Precision$ of All Clone Detectors for the Type-3 Clone References.

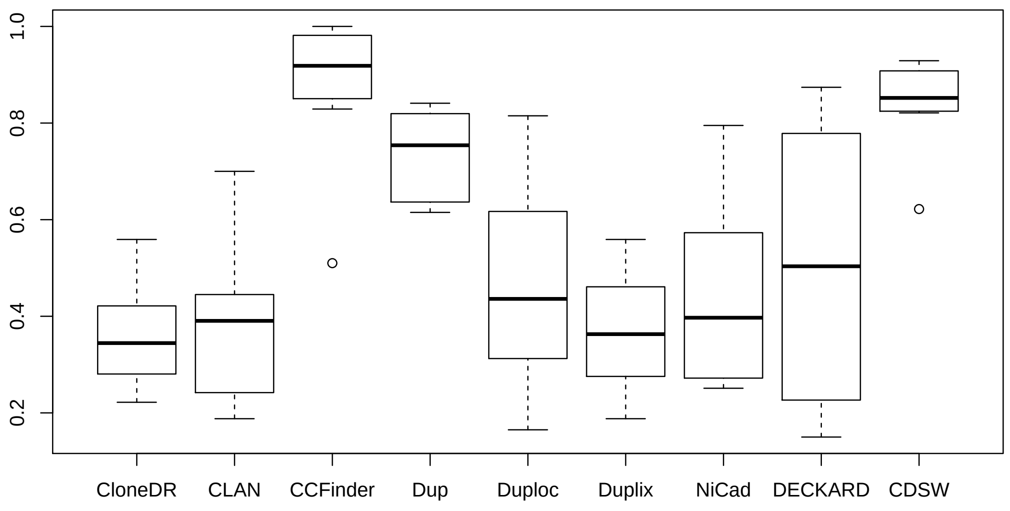


Figure 14: *Recall* of All Clone Detectors for the Type-1 and Type-2 Clone References.

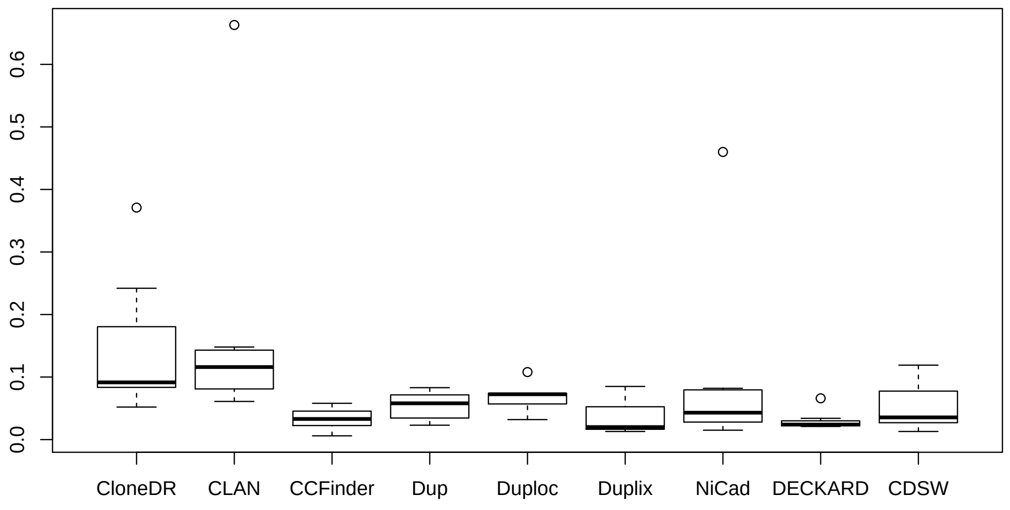


Figure 15: *Precision* of All Clone Detectors for the Type-1 and Type-2 Clone References.

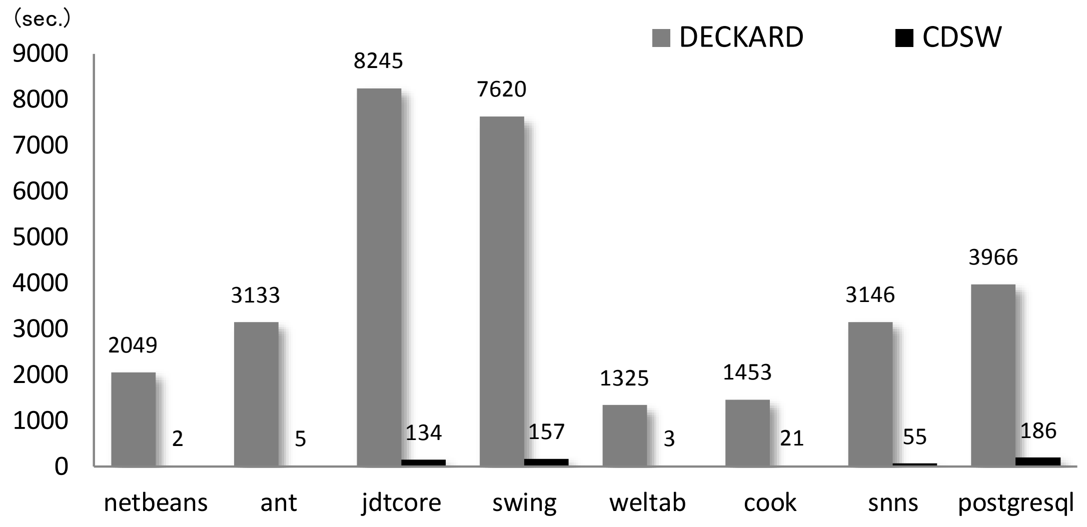


Figure 16: Execution Time of DECKARD and CDSW.

7 Threats to Validity

7.1 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 target software systems. Therefore, if all the code clones in target software systems are used as clone references, we might obtain different results. However, it is almost impossible to make clone references from all code clones in target software systems.

7.2 Code Normalization

The proposed method replaces every of variables and literals with a specific token as a normalization. This means that the normalization ignores the types of them. 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. Meanwhile, if the proposed method does not normalize source code, it cannot detect code clones that have differences of variable names or literals.

8 Conclusion

In this thesis, the authors proposed a new method to detect not only Type-1 and Type-2 but also Type-3 code clones by using the Smith-Waterman algorithm. The proposed method was developed as a software tool, CDSW. Furthermore, the authors remade the clone references used in Bellon's benchmark by adding information of gapped lines. The authors applied CDSW to eight open source software systems and calculated $precision$ and $recall$ by using the clone references remade by the authors. The authors confirmed the followings.

- The accuracy of evaluation of clone detection results improved by using not only the information where code clones start and where they end but also the information where the gaps are.
- CDSW detected as many relevant code clones as any other clone detectors used in Bellon ' s benchmark. Especially, CDSW detected the most relevant Type-3 code clones in them.
- CDSW might detect as many false positives as the other token-based clone detectors might detect.
- CDSW detected code clones in practical time for even large-scale software that has over six hundred thousand lines.

As described in section 6, in this experiment, The authors did not use the gapped lines that CDSW outputs for accuracy comparison of clone detectors. In the future, the authors are going to conduct experiments using the information where gaps are. If we use the information where gaps are for evaluating code clones, more accurate results would be obtained.

9 Acknowledgement

During this work, I have been fortunate to have received assistance from many people. This work could not have been possible without their valuable contributions.

First, I wish to express my deepest gratitude to my supervisor Professor Shinji Kusumoto at the Osaka University, for his continuous support, encouragement and guidance of the work. I also thank him for providing me an opportunity to do this work.

I am grateful to Kozo Okano, Associate Professor at the Osaka University for his helpful comments.

I am also deeply grateful to Hiroshi Igaki, Associate Professor at the Osaka University for his encouragement, practical advice, valuable suggestions, helpful comments and discussions throughout this work.

I would like to express my sincere gratitude to Yoshiki Higo, Assistant Professor at the Osaka University for his adequate guidance, valuable suggestions and discussions throughout this work. Without his supports, this work would never have reached completion.

I am deeply grateful to Keisuke Hotta, a doctoral course student at the Osaka University, for his valuable helps and discussions throughout this work. I am also thankful to him for reading my thesis, commenting on my views and helping me understand and enrich my ideas.

I thank greatly, Tomoya Ishihara, Shuhei Kimura, Yukihiro Sasaki, Hiroaki Shimba and Ji-achen Yang, a master course student at the Osaka University for their constant encouragements. Their support and care helped me to overcome many difficulties.

Finally, I would like to thank all of my friends in the Department of Computer Science at the Osaka University, especially the members in Kusumoto Laboratory, for their helpful advices, suggestions and assistance.

References

[1] 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, Oct. 2007.

[2] Clone Detection Literature. `http://www.cis.uab.edu/tairasr/clones/literature/`.

[3] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195–197, Mar. 1981.

[4] K. Inoue, T. Kamiya, and S. Kusumoto. Code-clone detection methods. *Computer Software*, Vol. 18, No. 5, pp. 529–536, Sep. 2001. (in Japanese).

[5] Y. Higo, S. Kusumoto, and K. Inoue. A survey of code clone detection and its related techniques. *IEICE Trans. on Information and Systems*, Vol. 91-D, No. 6, pp. 1465–1481, June 2008. (in Japanese).

[6] J.H. Johnson. Substring matching for clone detection tools. In *Proc. of the 10th International Conference on Software Maintenance*, pp. 120–126, Sep. 1994.

[7] 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*, pp. 109–118, Aug. 1999.

[8] 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, July 2002.

[9] CCFinderX. `http://www.ccfinder.net/ccfinderx.html`.

[10] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. on Software Engineering*, Vol. 32, No. 3, pp. 176–192, Mar. 2006.

[11] I.D. 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*, pp. 368–377, Mar. 1998.

[12] CloneDR. `http://www.semdesigns.com/Products/Clone/`.

[13] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 253–262, Oct. 2006.

[14] 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*, pp. 96–105, May 2007.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, July 2001.

[16] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pp. 301–309, Oct. 2001.

[17] Y. Higo and S. Kusumoto. Code clone detection on specialized pdgs with heuristics. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 75–84, Mar. 2011.

[18] Scorpio. `http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio`.

[19] J. Mayrand, 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*, pp. 244–253, Nov. 1996.

[20] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 283–292, Sep. 2011.

[21] Y. Sasaki, T. Yamaoto, Y. Hayase, and K. Inoue. File clone detection for a large scale software system. *IEICE Trans. on Information and Systems*, Vol. J94-D, No. 8, pp. 1423–1433, Aug. 2011. (in Japanese).

[22] N. Göde and R. Kosheke. Incremental clone detection. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering*, pp. 219–228, Mar. 2009.

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

[24] Detection of Software Clones. `http://www.bauhaus-stuttgart.de/clones/`.

[25] Javadoc. `http://javadoc.netbeans.org`.

[26] Eclipse. `http://www.eclipse.org`.

[27] Java 2 SDK. `http://java.sun.com`.

[28] Cook. `http://miller.emu.id.au/pmiller/software/cook/`.

[29] The Stuttgart Neuronal Network Simulator. `http://www-ra.informatik.uni-tuebingen.de`.

[30] PostgreSQL. `http://www.postgresql.org`.

[31] B.S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1343–1362, Oct. 1997.

[32] 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*, pp. 244–253, Nov. 1996.

[33] 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*, pp. 109–118, Aug. 1999.

[34] 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*, pp. 172–181, June 2008.