Fast and Precise Token-Based Code Clone Detection

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2016

Hiroaki Murakami

Fast and Precise Token-Based Code Clone Detection

January 2016

Hiroaki Murakami

Abstract

Recently, code clones (hereafter, clones) have received much attention from many researchers in the field of software engineering. Clones are code fragments that are identical or similar to other code fragments in the source code, and they are generated for various reasons, such as copy-and-paste operations. It has been said that the presence of clones makes software maintenance more difficult. This is because, if developers modify a code fragment, they have to check each of its clones and verify whether they need to make the same modification to the clone. On the other hand, reusing code fragments by copy-and-paste operations has some advantages; for example, developers can easily implement functions that are similar to an existing function.

To detect clones automatically, various clone detectors have previously been developed. Because clone definitions are neither generic nor strict, researchers individually make their own definitions of clones and develop clone detectors based on their individual definitions. Some research studies compare the accuracies or performances of clone detectors. From experimental results, we found that existing clone detectors need improvements with respect to two problems: (1) the detection time is too long, and (2) the detection accuracies are not sufficient. The reason why the detection time is too long is the use of graph-based detection techniques. Graph-based detection techniques transform source code to graph representations, and their isomorphic sub-graphs are regarded as clones. Comparing isomorphic sub-graphs requires much time. Next, the detection accuracies need improvement for two reasons. The first reason is the use of module-based detection techniques. Module-based detection techniques regard similar modules (e.g., blocks or methods) as clones. However, these techniques cannot detect clones that are partially duplicated in modules. The second reason is the presence of repeated instructions. Repeated instructions include repeating `case` entries in a `switch` statement, repeating similar method invocations, and so on. Token-based detection techniques, which are also used, detect common sub-sequences of tokens in source code as clones. These techniques detect clones in a short time, but yield many uninteresting clones (clones that developers do not need to investigate) from the repeated

instructions. Many uninteresting clones are factors of decreasing detection accuracies.

In this dissertation, we propose two detection techniques that improve the existing problems. The first technique folds every repeated instruction in the source code, and then it detects clones by token-based techniques. The folding operation prevents many of the uninteresting clones detected by token-based techniques. The second one uses the Smith-Waterman algorithm for detecting clones. This detection technique resolves existing problems because it does not use graphs for detecting clones, and it detects statement-based clones that are more fine-grained than modules. We conducted experiments by using Bellon's benchmark and confirmed that the proposed detection techniques succeed in improving the existing problems.

In addition, we propose a technique for visualizing clones that are detected by the proposed detection techniques. Some clones have negative impacts on software maintenance. For example, if developers modify a code fragment, they have to check whether the clones need the same modification. In this case, developers often use tools that take a code fragment as input and take its clones as output. However, when developers use such existing tools, they have to open a number of source files and move the scroll bar up or down to browse all detected clones. To reduce the cost of browsing detected clones, we propose a visualization technique so that developers can analyze the detected clones in a single view without moving the scroll bar. Moreover, we combine the proposed clone detection techniques and the visualization technique. We conduct experiments with student participants to investigate the effectiveness of the combined techniques. In the experiment, we compared the proposed tool with the existing tool from the perspective of time for checking clones and usability. As a result, we confirmed that the proposed tool was superior to the existing tool in both measures.

List of Publications

[1-1] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto, “Folding Repeated Instructions for Improving Token-Based Code Clone Detection”, in Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), pp.64 –73, September, 2012.

[1-2] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto, “Implementation and Evaluation of Code Clone Detection Method Designed for Information of Repetition in Source Code”, IPSJ Journal, Vol.54, No.2, pp.845–856, February, 2013.

[1-3] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto, “Gapped Code Clone Detection with Lightweight Source Code Analysis”, in Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013), pp.93–102, May, 2013.

[1-4] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto, “Gapped Code Clone Detection Using the Smith-Waterman Algorithm”, IPSJ Journal, Vol.55, No.2, pp.981–993, February, 2014.

[1-5] Hiroaki Murakami, Yoshiki Higo and Shinji Kusumoto, “A Dataset of Clone References with Gaps”, in Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014), pp.412–415, May, 2014.

[1-6] Hiroaki Murakami, Yoshiki Higo and Shinji Kusumoto, “Making Correct Clone Set with Locations of Gapped Lines”, IEICE Journal, Vol.J97-D, No.9, pp.1537–1540, September, 2014.

[1-7] Hiroaki Murakami, Yoshiki Higo and Shinji Kusumoto, “ClonePacker: A Tool for Clone Set Visualization”, in Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp.474-478, March, 2015.

Acknowledgements

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 for his continuous supports, encouragement and guidance of the work. I also thank him for providing me an opportunity to do this work. Without his supports, this work would not have been possible.

I would like to thank Proessors Katsuro Inoue and Kenichi Hagihara, for their insightful comments and encouragement regarding this dissertation. I would also like to acknowledge the guidance of Professors Toshimitsu Masuzawa and Yasushi Yagi.

I thank greatly professors and staffs in my laboratory. I am grateful to Associate Professor Yoshiki Higo and Keisuke Hotta at Fujitsu Ltd. for their adequate guidances, valuable suggestions and discussions throughout this work. I appreciate Associate Professors Kozo Okano at Shinshu University and Hiroshi Igaki at Osaka Institute of Technology. When they were members of my laboratory, they provided me helpful comments and suggestions. I would also like to thank office workers in my laboratory, Tomoko Kamiya, Kaori Fujino, Ritsuko Hama and Yumi Nakano for continual supports and heartful kindnesses.

I also thank greatly student members in my laboratory. I truly feel grateful to seniors, Kazuki Kobayashi, Yuko Muto, Yui Sasaki, Yoshihiro Nagase, Kentaro Hanada and Kazuki Yoshioka. I had a very pleasurable time with them. I am deeply grateful to my friends, Tomoya Ishihara, Shuhei Kimura, Yukihiro Sasaki, Hiroaki Shimba and Jiachen Yang for their constant encouragements. Their supports and cares helped me to overcome many difficulties. I thank greatly juniors, Ayaka Imazato, Takafumi Ohta, Hiroyuki Kirinuki, Noa Kusunoki, Yuya Fujita, Takamasa Mizoro, Shota Egawa, Akio Ohtani, Ryotaro Kou, Naoto Ogura, Yusuke Sabi, Sohichi Sumi, Yuki Huruta, Yusuke Yuki, Haruki Yokoyama, Kento Shimonaka, Hiroki Nakajima, Yuto Yamada and Masahiro Yamamoto.

I express my warm thanks to professors and members in Inoue laboratory. I thank greatly Associate Professor Makoto Matsushita and Assistant Professor

Takashi Ishio. Their technical advices and comments enriched my dissertation. I also thank greatly Eunjong Choi, Yu Kashima, Tetsuya Kanda, Umekawa Kohichi, Yuya Onizuka, Akira Goto, Yuki Yamanaka and Hiroki Wakisaka. When I visit Inoue laboratory, I feel happy to talk with them.

I would like to express our sincere thanks towards researchers on software engineering. Associate Professor Yasutaka Kamei at Kyushu University and Assistant Professor Shinpei Hayashi at Tokyo Institute of Technology and Associate Professor Norihiro Yoshida at Nagoya University and Assistant Professor Hideki Hata at Nara Institute of Science and Technology provided me valuable comments and technical advices.

Finally, I would like to thank my family and all of my friends in the Department of Computer Science at Osaka University. I could complete this work because I am encouraged by them at all times.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Overview of Research . . . 3
- 1.3 Overview of Dissertation . . . 5

2 Preliminaries **7**
- 2.1 Definition . . . 7
- 2.2 Causes of Creation . . . 7
- 2.3 Types of Clones . . . 8
- 2.4 Clone Detection Techniques . . . 10
- 2.5 Bellon's Benchmark for Comparing Clone Detectors . . . 15
- 2.6 Clone Visualization Techniques . . . 18

3 Pre-Processing of Clone Detection for Reducing Uninteresting Clones **21**
- 3.1 Background . . . 21
- 3.2 Research Motivation . . . 23
- 3.3 Clone Detection with Folding Repeated Instructions . . . 25
- 3.4 Implementation . . . 29
- 3.5 Overview of Investigation . . . 30
- 3.6 Experiment A . . . 31
- 3.7 Experiment B . . . 34
- 3.8 Conclusion . . . 39

4 Clone Detection Using Smith-Waterman Algorithm **41**
- 4.1 Background . . . 41
- 4.2 Smith-Waterman Algorithm . . . 43
- 4.3 Proposed Technique . . . 46
- 4.4 Experimental Design . . . 51
- 4.5 Preliminary Experiment . . . 52

4.6 Experiment A 52
4.7 Experiment B 55
4.8 Threats to Validity 58
4.8.1 Clone References 58
4.8.2 Code Normalization 59
4.8.3 Three Parameters in Smith-Waterman Algorithm 59
4.9 Discussion 59
4.10 Conclusion 60

5 Clone Visualization Using Circle Packing **61**
5.1 Introduction 61
5.2 Circle Packing 62
5.3 Proposed Technique 64
5.3.1 Step-1: Detecting Clones 64
5.3.2 Step-2: Visualizing Clone Set 65
5.4 Tool: ClonePacker 65
5.4.1 Implementation 65
5.4.2 How to Use ClonePacker 65
5.4.3 Example of Supporting Scenario 66
5.5 Experiment 67
5.5.1 Evaluation for Clone Analysis Time 67
5.5.2 Evaluation for Usability 69
5.6 Threats to Validity 70
5.6.1 Configurations of Clone Detection 70
5.6.2 Target Software System 71
5.6.3 Participants 72
5.6.4 Experimental Methodology 72
5.7 Conclusions 72

6 Conclusion **73**
6.1 Contribution 73
6.2 Future Work 75

List of Figures

2.1 Example of *Clone Pair* and *Clone Set* 8
2.2 Examples of Type-1 Clones 9
2.3 Examples of Type-2 Clones 9
2.4 Examples of Type-3 Clones 10
2.5 Examples of Type-4 Clones 10
2.6 Example of Transformation from Source Code into AST 12
2.7 Example of Transformation from Source Code into PDG 13
2.8 Example of Reordered Clones 14
2.9 Examples of *Clone References* and *Clone Candidates* for Calculating *ok* and *good* Values 18
2.10 Example of Scatter Plot in Gemini [83] 19
2.11 Example of Edge Bundle [22] 20
2.12 Example of Tree View in VisCad [3] 20

3.1 Motivating Example, Which Shows That Many Uninteresting Clones Are Detected from Repeated Instructions 23
3.2 Transformed Source Code in Figure 3.1after Folding Operation . . 24
3.3 Motivating Example, Which Shows that Human Regards Whole Repeated Instructions as Clones 24
3.4 Overview of Proposed Technique 26
3.5 How Input Source Code is Transformed into Folded Hash Sequence 28
3.6 Example of Transformation with Heuristics 30
3.7 # of *Clone Candidates*, *Precision* and *Recall* in Experiment A . . 32
3.8 *Clone Reference* Newly Detected by Using Folding Operation . . 33
3.9 $Precision$ and $Recall$ of All Detectors for All Target Software Systems 36
3.9 $Precision$ and $Recall$ of All Detectors for All Target Software Systems 37
3.9 $Precision$ and $Recall$ of All Detectors for All Target Software Systems 38

3.10 *Clone Reference* Located in Repeated Instructions 39

4.1 Smith-Waterman Algorithm Applied to Two Base Sequences, "GACGACAACT" and "TACACACTCC" 44
4.2 Overview of Proposed Technique 45
4.3 Example of Detection Process Using Proposed Technique 47
4.3 Example of Detection Process Using Proposed Technique 48
4.3 Example of Detection Process Using Proposed Technique 49
4.4 $Recall$, $Precision$ and $F\text{-}measure$ on 3-Tuple of Parameters ($match$, $mismatch$, gap). 51
4.5 Example of Enhanced *Clone Reference* (*Clone Reference* No. 1101). 53
4.6 $Recall$, $Precision$ and $F\text{-}measure$ of CDSW Using Both *Clone References* 54
4.7 $Recall$, $Precision$ and $F\text{-}measure$ for Type-3 *Clone References* 56
4.8 $Recall$, $Precision$ and $F\text{-}measure$ for Type-1, Type-2 and Type-3 *Clone References* 57
4.9 Exection Time of DECKARD, NiCad and CDSW for Target Software Systems 58

5.1 Example of Circle Packing 62
5.2 Overview of Proposed Technique 63
5.3 Screenshot of ClonePacker 66
5.4 Modifications in JFreeChart 67
5.5 Results of Task Completion Time 68

List of Tables

2.1 Target Software Systems 16
2.2 Target Clone Detectors 16

3.1 Rate of Self-Overlapping Clones That Became Undetected by Using Folding Operations 34
3.2 # of *Clone Candidates*. Every "-" Means That Detector Could not Finish Clone Detection Because of Scalability Issue. 35

5.1 Details of Experimental Tasks 67
5.2 Results of System Usability Scale 71

Chapter 1

Introduction

This chapter provides an introduction to this dissertation.

1.1 Background

Software development consists of the following five phases [12].

- Requirement Phase
- Design Phase
- Implementation Phase
- Testing Phase
- Maintenance Phase

It has been said that the maintenance phase is the most expensive phase [7, 64]. For example, according to [2], approximately 80% of the total cost is spent on software maintenance. Maintenance of software systems is defined as the modification of a software product after its delivery in order to correct faults, to improve the performance or other attributes, or to adapt the software to a modified environment [26,54]. To modify software products, modifying the source code is unavoidable. It has been said that the presence of *code clones* makes software maintenance more difficult [35].

Code clones (hereafter, clones) are code fragments that are identical or similar to other code fragments in the source code. Clones are generated for various reasons, such as copy-and-paste operations. The reason why clones make software maintenance more difficult is that, if developers modify a code fragment, they

have to check its clones and verify whether the fragments need the same modifications [48]. On the other hand, reusing code fragments by copy-and-paste operations has some advantages; for example, developers can implement functions easily that are similar to an existing function.

To detect clones automatically, various clone detectors have previously been developed. Because clone definitions are neither generic nor strict, researchers individually make their own definitions of clones and develop clone detectors based on their individual definitions. Some research has compare the accuracies or performances of clone detectors [9, 68], and from the reported experimental results, we conclude that the existing clone detectors need improvements with respect to the following two problems.

Detection time is too long

Clone detection techniques are categorized into various types (described in Chapter 2). The program dependence graph (PDG)-based techniques constitute one category [24, 43]. These techniques transform source code to graph representations and then regard isomorphic sub-graphs as clones. The existing PDG-based techniques can detect clones that other techniques cannot detect, but comparing isomorphic sub-graphs requires much time.

Detection accuracies are not sufficient

Gapped clones are clones with modifications of some statements (e.g., added or modified statements) after a copy-and-paste operation. To detect gapped clones, some researchers have proposed techniques that regard similar modules (e.g., blocks or methods) as clones [53, 65]. However, these techniques cannot detect clones that are partially duplicated in modules.

Moreover, it is said that the presence of repeated instructions (e.g., repeating `case` entries in a `switch` statement or repeating similar method invocations) is a large factor for decreased accuracies of token-based detection techniques [25]. Token-based detection techniques constitute another detection category. These techniques detect common sub-sequences of tokens in the source code as clones. Although they detects clone in a short time, but they yield many uninteresting clones (clones that developers do not need to investigate) from the repeated instructions. Many uninteresting clones are factors of decreased detection accuracies.

In this dissertation, we propose two fast and precise token-based clone detection techniques that improve existing techniques with respect to these problems. In addition, we propose a technique for visualizing clones that are detected by our proposed detection techniques.

Some clones have negative impacts on software maintenance. For example, if developers modify a code fragment, they have to check whether its clones need the same modifications. In this case, developers often use tools that take a code fragment as input and take its clones as output. However, when developers use such existing tools, they have to open a number of source files and move the scroll bar up or down to browse all detected clones [27]. To reduce the cost of browsing the detected clones, we propose a visualization technique so that developers can analyze the detected clones in a single view without moving the scroll bar. Moreover, we combine the proposed clone detection techniques and the visualization technique, and then we conduct experiments with student participants to investigate the performance of the combined techniques.

1.2 Overview of Research

This dissertation describes the results of three studies. The first study is on the pre-processing of clone detection for reducing uninteresting clones. The second one proposes a clone detection technique for decreasing detection time and increasing detection accuracies. The third one proposes a technique to visualize clones for reducing the cost of browsing detected clones.

Pre-Processing of Clone Detection for Reducing Uninteresting Clones

As described before, the presence of repeated instructions in the source code is a large factor for detecting uninteresting clones. In this research, we propose a technique that folds every repeated instruction, and then it detects clones by token-based technique. The folding operation prevents many uninteresting clones detected by token-based techniques. We implemented the proposed technique as a tool, named FRISC. Then, we conducted a quantitative evaluation of FRISC by using Bellon's benchmark (described in Chapter 2). In his benchmark, Bellon made references of clones (called *clone references*), and compared accuracies among the clone detectors.

From the results of the evaluation, we confirmed the following:

- The folding operation reduced many uninteresting clones.
- Some clones were newly detected by the folding operation.
- FRISC detected more *clone references* than any other detectors in most cases.

Clone Detection Using the Smith-Waterman Algorithm

To detect gapped clones (clones with some gapped lines, described in Chapter 2), abstract syntax tree (AST)-based techniques, PDG-based techniques, metric-based techniques, and text-based techniques that use the longest common subsequence (LCS) algorithm have been proposed. However, each of these techniques has limitations. For example, the existing AST-based techniques and PDG-based techniques require much time for detecting clones. The existing metric-based techniques and text-based techniques using the LCS algorithm cannot detect clones if modules are partially duplicated, because these techniques calculate the similarity between two modules. To resolve these limitations, we propose a technique for detecting gapped clones by using the Smith-Waterman algorithm. We developed the proposed technique as a tool, named CDSW. CDSW resolved the existing problems because CDSW does not use AST or PDG for detecting clones, and CDSW detects statement-based clones that are more fine-grained than modules.

Moreover, we improved Bellon's benchmark. Bellon's *clone references* do not have location information of gapped lines in gapped clones. Thus, Bellon's benchmark does not evaluate some gapped clones correctly. To resolve the issue, we added location information of gapped lines to the *clone references*. We report an experiment that compares Bellon's *clone references* and our *clone references*. Finally, we compare accuracies between CDSW and existing clone detectors by using the enhanced *clone references*.

From the results of experiments, we confirmed the following:

- Our *clone references* can evaluate gapped clones more correctly than Bellon's *clone references*.
- CDSW detected clones in a large software system in a short time.
- CDSW was the best of all clone detectors from the viewpoint of *F-measure*.

Clone Visualization Using Circle Packing

As described before, developers often use tools that take a code fragment as input and take its clones as output. However, when developers use such existing tools, they have to open a number of source files and move the scroll bar up or down to browse all detected clones. To reduce the cost of browsing the detected clones, we propose a technique for browsing detected clones by using a single view without moving the scroll bar. The proposed technique was developed as a tool, named ClonePacker, which uses the circle packing technique for visualization. We conducted experiments with student participants, who compared ClonePacker with the existing tool. In the experiments, we evaluated the time taken to report the

locations of clones and the usability of the tools. As a result of the experiments, we confirmed that ClonePacker is better than the existing tools in both location detection and usability.

1.3 Overview of Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 is an overview of clones. This chapter describes definitions of clones, some clone detection/visualization techniques, and some related work on clones.

Chapters 3, 4, and 5 discuss the above three studies, i.e., the pre-processing of clone detection for reducing uninteresting clones, the clone detection technique for decreasing detection time and increasing detection accuracies, and the technique to visualize clones for reducing the cost of browsing detected clones.

Chapter 3 describes the pre-processing of clone detection for reducing uninteresting clones. In addition, we explain the results of the experiments using Bellon's benchmark.

Chapter 4 presents a new clone detection technique using the Smith-Waterman algorithm. In this chapter, we explain the Smith-Waterman algorithm and how to find clones from the source code by using the algorithm. Moreover, we describe the enhanced *clone references* and how to use them. Finally, this chapter shows a comparison of the accuracies between CDSW and the existing clone detectors.

Chapter 5 presents a clone visualization technique using the circle packing technique. It describes circle packing, how to use ClonePacker, and the experiments with the student participants.

Finally, Chapter 6 concludes this dissertation and states future work.

Chapter 2

Preliminaries

This chapter provides preliminaries of code clones.

2.1 Definition

Code clones (hereafter, clones) are defined as fragments that are identical or similar to other code in the source code. This subsection uses Figure 2.1 to explain two terms used in this dissertation.

The first term is *clone pair*, which is a pair of code fragments that are identical or similar to each other. In Figure 2.1, the three code fragments α, β, and γ are clones. In this case, three pairs of code fragments, (α, β), (α, γ), and (β, γ), are clone pairs.

The second term is *clone set*, which is a set of code fragments that are identical or similar to each other. In Figure 2.1, the set of code fragments (α, β, and γ) is a clone set.

2.2 Causes of Creation

Clones can be created or introduced in the following situations.

Copy-and-Paste Operations: This is the most prevalent situation in which clones are created. Reusing code by copy-and-paste operations is a common practice in software development, because it is quite easy and enables us to develop software faster.

Stylized Processing: Processing that is used frequently (e.g., calculations of income tax, insertions in queues, and access to data structures) may cause code duplications.

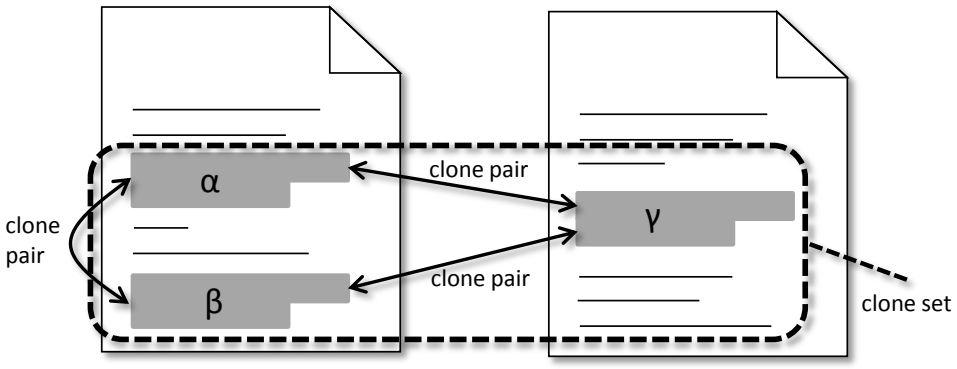


Figure 2.1: Example of *Clone Pair* and *Clone Set*

Lack of Suitable Functions: Developers 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: Developers can introduce code duplication intentionally to improve the performance of software systems in the case that 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, the tools may generate similar code fragments.

Handling Multiple Platforms: Software systems that can handle multiple operation systems or CPUs tend to include many clones in the process handling of each platform.

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

2.3 Types of Clones

Bellon et al. categorized clones into the following three types [9].

Type-1: Identical code fragments except for variations in whitespace, layout, and comments.

```
int sum(int data[], int n) {
  int result = 0;
  for(int i = 0; i < n; i++) {
    result += data[i];
  }
  return result;
}
```

(a) Code Fragment 1

```
int sum(int data[], int n)
{
  int result = 0;
  for(int i = 0; i < n; i++)
  {
    result += data[i];
  }
  return result;
}
```

(b) Code Fragment 2

Figure 2.2: Examples of Type-1 Clones

```
int sum(int data[], int n) {
  int result = 0;
  for(int i = 0; i < n; i++) {
    result += data[i];
  }
  return result;
}
```

(a) Code Fragment 1

```
int sum(int data[], int n) {
  int total = 0;
  for(int i = 0; i < n; i++) {
    total += data[i];
  }
  return total;
}
```

(b) Code Fragment 2

Figure 2.3: Examples of Type-2 Clones

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments.

Type-3: Copied fragments with further modifications, such as changed, added, or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout, and comments.

Moreover, some research groups have recently suggested an additional type of clones [40, 68].

Type-4: Code fragments that perform the same computation but are implemented by different syntactic variants.

Figures 2.2, 2.3, 2.4, and 2.5 give examples of each type of clones. In the figures, on the left are the original code fragments and one the right are the clones.

The right code fragment in Figure 2.2 is generated by changing formats after a copy-and-paste operation. Thus, the two code fragments in Figure 2.2 are Type-1

```
int sum(int data[], int n) {
  int result = 0;
  for(int i = 0; i < n; i++) {
    result += data[i];
  }
  return result;
}
```

(a) Code Fragment 1

```
int sum(int data[], int n) {
  int result = 0;
  for(int i = 0; i < n; i++) {
    result = result + data[i];
  }
  return result;
}
```

(b) Code Fragment 2

Figure 2.4: Examples of Type-3 Clones

```
int sum(int data[], int n) {
  int result = 0;
  for(int i = 0; i < n; i++) {
    result += data[i];
  }
  return result;
}
```

(a) Code Fragment 1

```
int sum(int data[], int n) {
  if(n == 1){
    return data[n-1];
  } else {
    return data[n-1]+sum(data, n-1);
  }
}
```

(b) Code Fragment 2

Figure 2.5: Examples of Type-4 Clones

clones. The differences between the code fragments in Figure 2.3 are the variable names. This means that the two code fragments in Figure 2.3 are Type-2 clones. In Figure 2.4, the addition operation is changed. Since a gap appears in the code fragment, the two code fragments in Figure 2.4 are Type-3 clones. Finally, in Figure 2.5, the right code fragments are not similar to the left ones. However, these two code fragments perform the same computation. Therefore, the two code fragments in Figure 2.5 are Type-4 clones.

2.4 Clone Detection Techniques

Clone detection is an important topic in the research of clones. Many techniques, including those listed below, detect clones automatically. Clone detection techniques can be categorized into the following categories.

Line-based Techniques

Line-based techniques detect clones by comparing every line of code fragments as a string. They regard multiple consecutive lines that exceed a specified threshold as clones. Line-based techniques can detect clones quickly as

compared with other detection techniques, because they do not require any pre-processing of the source code. However, they cannot detect clones that have different coding styles.

The techniques of Johnson [34] and Ducasse [18] are well-known line-based techniques. Their techniques compare every line of code after removing whitespaces, tabs, and line breaks. Thus, they detect clones that have different coding styles and are language-independent.

Simian is one of the most famous and commonly used line-based clone detectors [74]. Simian can handle about 20 programming languages (e.g., Java, C, and C++) and detect clones very quickly.

Token-based Techniques

First, token-based techniques transform source code into a token sequence. Then, they detect common sub-sequences of the tokens as clones. Compared to line-based techniques, token-based techniques are robust for code formatting. The detection speed of token-based techniques is inferior to that of line-based techniques, but superior to the tree-based or graph-based techniques discussed below.

Kamiya developed a token-based clone detector, named CCFinder [37], that is well known and has been widely used by many developers. It replaces user-defined identifiers (e.g., method names or variable names) with specific tokens. By pre-processing, CCFinder can detect Type-2 clones.

A major version of CCFinder is named CCFinderX [13]. Also, Livieri developed a distributed version of CCFinder, named D-CCFinder [51], that was implemented in a system with 80 PC workstations. In [51], a huge collection of open source software with about 400 million lines was analyzed with D-CCFinder in about 2 days.

Li developed a token-based clone detector, named CP-Miner [50]. First, lexical and syntax analyses are performed on the source code. User-defined identifiers are replaced with specific tokens, as in CCFinder. The major difference between CP-Miner and CCFinder is the detection algorithms. In CP-Miner, hash values are calculated from every statement, and then a frequent pattern mining algorithm [1] is applied for detecting clones. In the frequent pattern mining algorithm, the hash values do not have to be consecutive. Thus, CP-Miner can detect Type-3 clones.

Tree-based Techniques

In tree-based techniques, source code is transformed into a tree representation. An abstract syntax tree (AST) is one of the well-known tree repre-

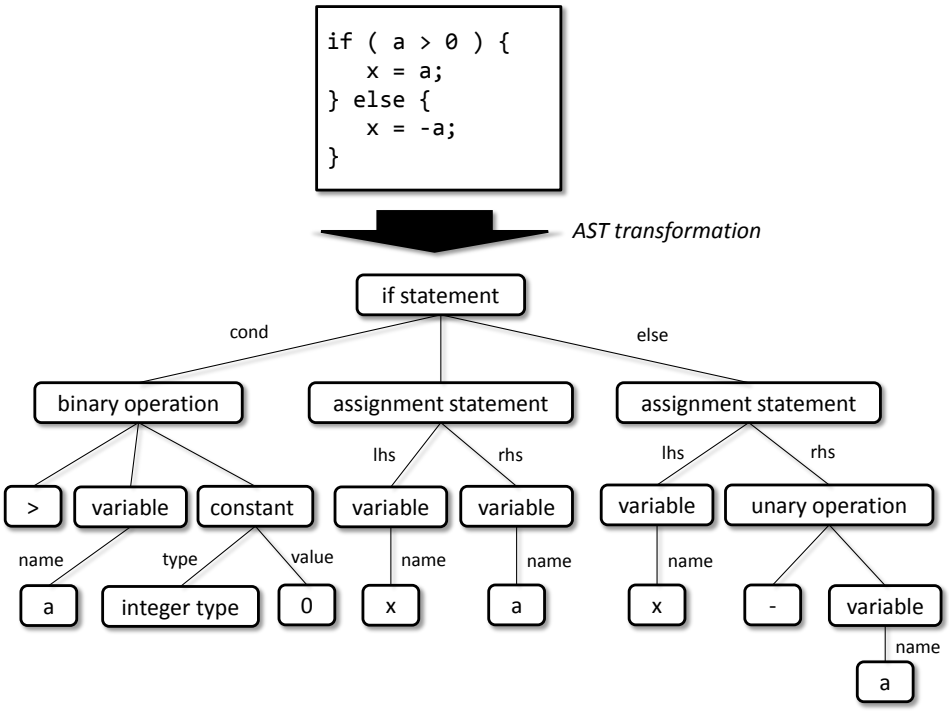


Figure 2.6: Example of Transformation from Source Code into AST

sentations. Figure 2.6 shows an example of an AST transformation. Tree-based techniques regard common sub-trees as clones, and these techniques are therefore also robust for code formatting. However, they have the disadvantage of requiring more time for detecting clones than do text-based and token-based techniques.

Baxter developed a tree-based clone detector, named CloneDR [8, 15], that calculates various metrics based on ASTs, then detects clones by comparing the metrics. Thus, CloneDR detects clones quickly in large software systems. CloneDR can also handle a lot of programming languages.

Koschke's technique [45] and Jiang's technique [33] are also tree-based approaches. In Koschke's technique, ASTs are compared with a suffix tree algorithm to increase the detection speed. Jiang's detector, named DECKARD, uses a locality-sensitive hashing algorithm [63] to detect clones. With the algorithm, DECKARD can detect Type-3 clones.

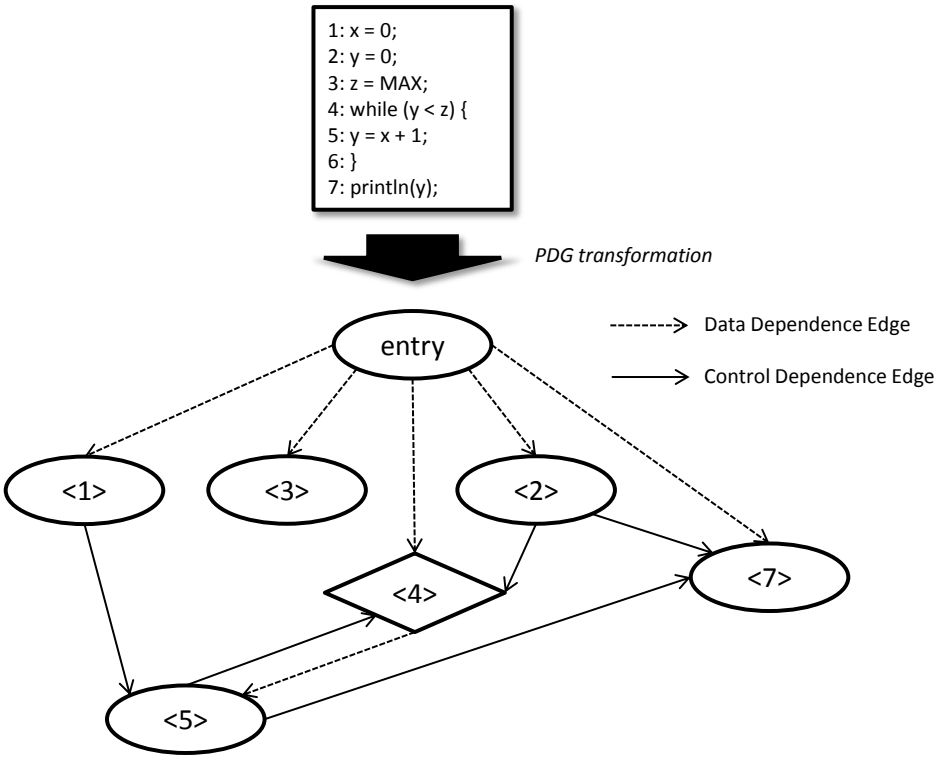


Figure 2.7: Example of Transformation from Source Code into PDG

Graph-based Techniques

In graph-based techniques, source code is transformed into a graph representation. A program dependence graph (PDG), which is one of the well-known graph representations, has data dependence edges and control dependence edges for each element of source code. Figure 2.7 shows an example of the transformation from source code into a PDG. Graph-based techniques regard isomorphic sub-graphs as clones. Because PDGs require a semantic analysis for their creation, these approaches require much more cost than do other detection techniques. However, these approaches can detect clones with some differences that have no impact on the program behavior. Figure 2.8 shows such a clone, which is reordered. Other techniques cannot detect these clones because they have reordered statements.

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

(a) Code Fragment 1

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

(b) Code Fragment 1

Figure 2.8: Example of Reordered Clones

Komondoor proposed the initial graph-based technique [43]. Komondoor's technique uses program slicing [85] to find isomorphic sub-graphs.

Krinke's technique [46] and Higo's technique [24, 72] are also classified as graph-based techniques. Both of these techniques are designed to reduce detection cost. Krinke's technique sets the limit of the search range for finding isomorphic sub-graphs. Higo proposed a technique that aggregates nodes in PDGs under some conditions. Moreover, he introduced a new dependence edge named "execution dependence edge" for PDGs [24]. By introducing the execution dependence edge, Higo's technique successfully detected clones that other graph-based techniques could not detect.

Other Detection Techniques

First, metric-based techniques are categorized into this category. Metric-based techniques calculate various metrics for every program module (e.g., method, function, and class), then detect clones by comparing the similarity of the modules based on the metrics. Mayland developed the initial metric-

based detector, named CLAN [53], that creates ASTs from source code, then calculates the metrics obtained from the ASTs. Lanubile's technique [49] and Kodhai's technique [42] are also categorized as metric-based techniques.

Roy developed a clone detector, named NiCad [17,59,65,67], that uses TXL programming language [16] for clone detection. NiCad has been extended to other uses, such as the analysis environment VisCad [3], clone genealogy creator gCad [69, 70], and SimCad [80, 81], which uses *simhash* [14] for clone detection.

Some researchers have proposed file-based detection techniques. Ossher's technique [61] and Sasaki's technique [86] are categorized into this category. File-based detection techniques can detect clones in a short time, because they compare every file instead of lines or tokens. A disadvantage of these techniques is that they cannot detect clones that are partially duplicated in files. Method-based detection techniques have also been studied [30, 32].

Recently, some researchers have proposed techniques that detect clones by using information other than the source code. Kim proposed a clone detection technique by comparing the memory states in each method and developed the proposed technique as a tool, named MeCC [40]. This technique can detect clones by using the fragment similarities missed by other detectors. Clone detections using Java byte-code have also been studied [38, 71]

Moreover, incremental detection techniques have been proposed [21, 28, 31, 58]. Incremental detections, which are clone detection results in previous code revisions, are registered, then used for the next clone detection. By reusing the detection results in previous revisions, the detection costs on the current revision are reduced.

2.5 Bellon's Benchmark for Comparing Clone Detectors

Some researchers have conducted comparative experiments of clone detectors [9, 66, 68, 76–78]. Bellon conducted the largest scale experiment [9]. In this subsection, we describe Bellon's benchmark.

Bellon's benchmark is one of the most famous benchmarks in the clone community. He compared six clone detectors from the perspective of accuracy and performance. He conducted the comparison in the following steps.

Step-1: Bellon selected eight software systems as target systems, and six clone detectors as target detectors. Tables 2.1 and 2.2 show the details of the target systems and the target detectors, respectively.

Step-2: He asked the developers of the clone detectors to detect clones from the target systems. Then the developers sent the location information of the detected clones to Bellon.

Step-3: 2% of the clones sent from the developers were randomly selected, then he checked each of them manually to determine whether they were actually clones.

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

Clone candidates: clones found by clone detectors.

Clone references: clones judged by Bellon manually.

In Bellon's benchmark, ok and $good$ values are defined. These two values decide whether each *clone candidate* matches any *clone references*. Assume that C is a *clone candidate*, R is a *clone reference*, F_1 is one code fragment, and F_2 is the other code fragment; $lines(F)$ means a set of code lines in F. The definition of the ok value is shown in Eqs. 2.1 and 2.2.

Table 2.1: Target Software Systems

Name	Language	Lines of Code	# of *Clone 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

Table 2.2: Target Clone Detectors

Developer	Clone Detector	Detection Technique
Baker	Dup [5]	token-based technique
Baxter	CloneDR [8]	AST-based technique
Kamiya	CCFinder [37]	token-based technique
Merlo	CLAN [53]	metrics-based technique
Rieger	Duploc [18]	line-based technique
Krinke	Duplix [46]	PDG-based technique

$$
\begin{aligned}
ok(C,R) = min(\quad & max(\quad contain(C.F_1, R.F_1), \\
& \qquad\quad contain(R.F_1, C.F_1), \\
& \quad max(\quad contain(C.F_2, R.F_2), \\
& \qquad\quad contain(R.F_2, C.F_2))).
\end{aligned} \tag{2.1}
$$

$$
contain(F_1, F_2) = \frac{|lines(F_1) \cap lines(F_2)|}{|lines(F_1)|}. \tag{2.2}
$$

The ok value intuitively means the overlapping ratio of a *clone candidate* and a *clone reference*. The more the ok value increases, the more the overlapping part of the *clone candidate* and the *clone reference* increases. Meanwhile, the $good$ value is much more restrictive for a candidate-reference match. The definition of a $good$ value is shown in Eqs. 2.3 and 2.4.

$$
\begin{aligned}
good(C,R) = min(\quad & overlap(C.F_1, R.F_1), \\
& overlap(C.F_2, R.F_2)).
\end{aligned} \tag{2.3}
$$

$$
overlap(F_1, F_2) = \frac{|lines(F_1) \cap lines(F_2)|}{|lines(F_1) \cup lines(F_2)|}. \tag{2.4}
$$

Clone references and *clone candidates* are matched when ok or $good$ values are equal to or greater than a threshold. In Bellon's benchmark, 0.7 is used for the threshold.

We show examples of calculating the ok value and the $good$ value in Figure 2.9. In the figure, the 80th-86th lines in the left code and the 267th-275th lines in the right code are *clone references*. Moreover, the 82nd-85th lines in the left code and the 269th-272nd lines in the right code are *clone candidates*. In this case, ok and $good$ values are calculated as follows.

$$
\begin{aligned}
ok(C,R) &= min(max(\frac{4}{4}, \frac{4}{7}), max(\frac{4}{4}, \frac{4}{9})) \\
&= min(\frac{4}{4}, \frac{4}{4}) \\
&= 1.0 \\
good(C,R) &= min(\frac{4}{7}, \frac{4}{9}) \\
&= 0.44
\end{aligned} \tag{2.5}
$$

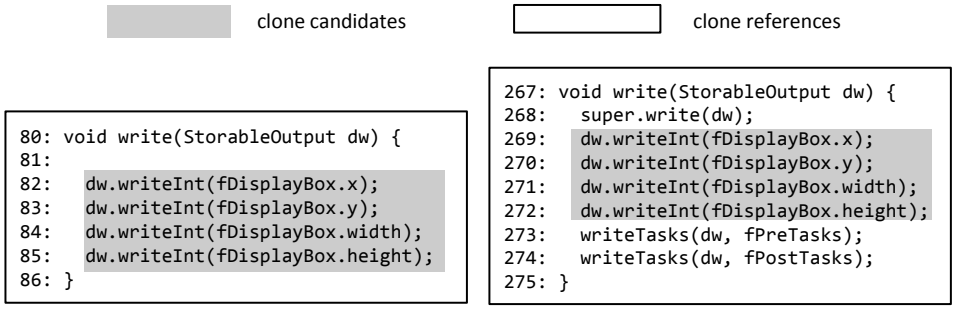


Figure 2.9: Examples of *Clone References* and *Clone Candidates* for Calculating *ok* and *good* Values

Then, $recall$ and $precision$ are calculated. Assume that $Refs$ is a set of *clone references*, $Cands$ is a set of *clone candidates*, and $DetectedRefs$ is a set of *clone references* whose ok or $good$ values are greater than or equal to the threshold. Therefore, the definitions of $recall$ and $precision$ are as shown in Eqs. 2.6 and 2.7.

$$Recall = \frac{|DetectedRefs|}{|Refs|}. \tag{2.6}$$

$$Precision = \frac{|DetectedRefs|}{|Cands|}. \tag{2.7}$$

Then, Bellon compared $recall$ and $precision$ between the target detectors. The results in Bellon's benchmark are summarized as follows.

- Line-based and token-based detection techniques have high $recall$ and low $precision$. This means that these techniques detect many *clone references*, but yield many false positives.
- Metric-based detection techniques have low $recall$ and high $precision$. This means that many of clones detected by these techniques are in *clone references*, even though these techniques miss many *clone references*.

2.6 Clone Visualization Techniques

Many clone detectors report the location information of detected clones, such as file names, start lines, and end lines. However, it is sometimes difficult for developers to analyze clones with only location information. Thus, some researchers have proposed clone visualization techniques.

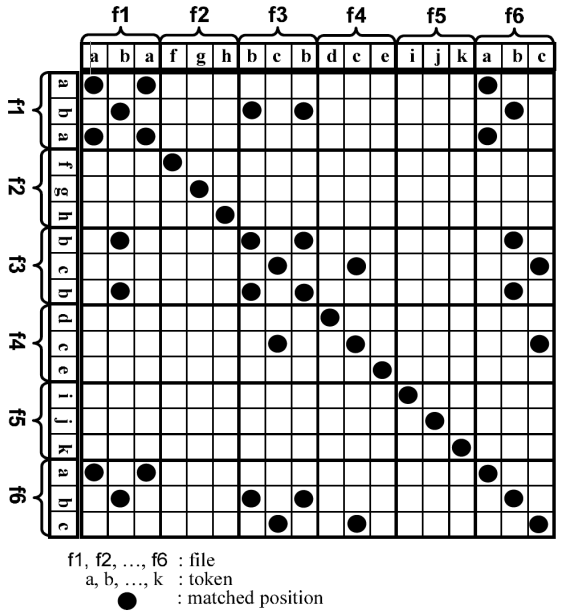


Figure 2.10: Example of Scatter Plot in Gemini [83]

Ueda developed a clone analysis environment, named Gemini [82, 83], that uses scatter plots for visualizing clones. Figure 2.10 shows an example of a scatter plot. Both the vertical and horizontal axes represent tokens of source code. Each black dot means that the corresponding tokens on the vertical and the horizontal axes are the same.

Hauptmann proposed a technique that shows clone detection results by using edge bundles [22]. Figure 2.11 shows an example of a clone detection result by using edge bundles. The outermost rectangles show directories, and pins pointing to a rectangle indicate files that exist in the directory. The blue lines connecting two pins indicate that the two files are clones. The advantage of this technique is that it associates the clone detection results with a file hierarchy.

Asaduzzaman developed a clone analysis support tool, named VisCad [3], that uses scatter plots and tree views for visualizing clones. Figure 2.12 shows an example of a tree view. The tree view shows the cloning status of directories and files by using rectangles. Developers can find the modules that have many clones from all the modules.

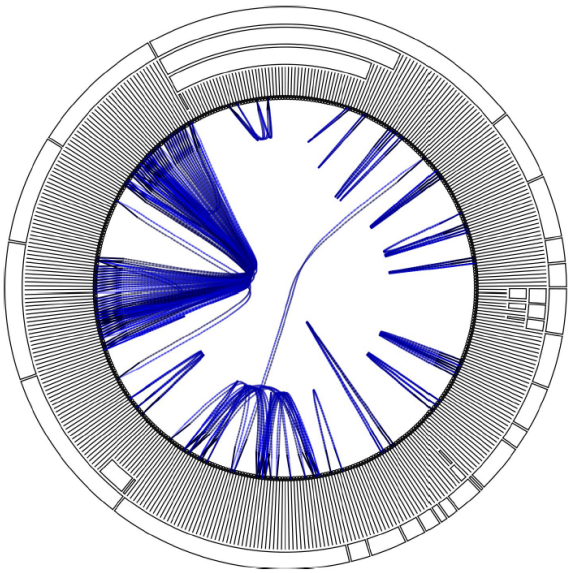


Figure 2.11: Example of Edge Bundle [22]

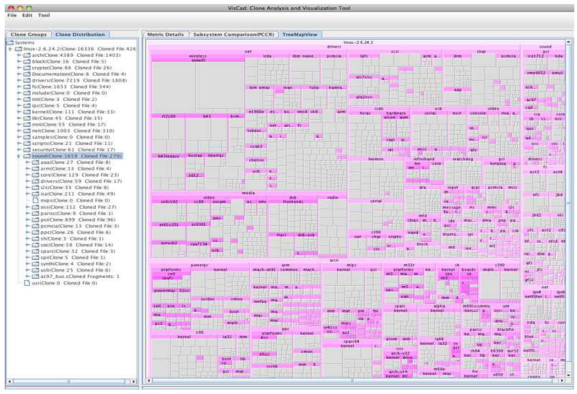


Figure 2.12: Example of Tree View in VisCad [3]

Chapter 3

Pre-Processing of Clone Detection for Reducing Uninteresting Clones

3.1 Background

Many software systems have many clones. In order to detect clones in the source code automatically, a variety of detection techniques has been proposed in the past [68]. At present, line-based and token-based detection techniques are often used because of the following reasons:

- line/token-based detections have high scalability because they neither require deep source code analysis nor construct complex intermediate structures for the detection. Consequently, they are used in various contexts of software developments. Also, they are used for detecting clones from large-scale software [37, 50], a number of software [44, 51, 73], a number of consecutive revisions of software [20, 29, 47, 52];
- implementing line/token-based detection techniques for multiple programming languages is easier than other detection techniques like PDG-based ones. Popular line/token based tools, CCFinder [37] and Simian [74], can handle widely-used languages such as C/C++, Java, COBOL.

On the other hand, automatic clone detections by tools inherently produce uninteresting clones. Every detection technique has its own unique definition of clones, and it detects clones based on the definition. However, developers do not need all clones detected by tools.

Bellon, et al. compared *recall* and *precision* of six detection tools by using oracle, which is a reference set of clones [9]. As a result, they revealed the followings:

- high *recall* tools detect many clones which implies that the detection results of those tools include many uninteresting clones [4, 37];
- high *precision* tools have low *recall* [8, 65]. Detecting a small number of uninteresting clones is their advantage but they miss many *clone references*.

To summarize the above points, line/token-based detections have high scalability, and they can be applied to various contexts of software development. However, they yield many uninteresting clones. We think that the presence of repeated instructions in source code is a large factor of uninteresting clones detection based on our experiences of clone related research [23]. For example, if we detect clones from the following example using a token-based approach with a suffix tree or suffix array algorithm, we will obtain a clone pair: one consists of the 1-2 lines code fragment; the other consists of the 2-3 lines code fragment. Both the code fragments in the clone pair are overlapped with each other. Detecting such a clone pair is meaningless.

```
1: unsigned char division mask;
2: unsigned int division offset;
3: unsigned int division size;
```

The above example is a repetition of consecutive variable declarations. If we tailor detection to ignore repeated instructions, the clone pair becomes undetected. We have revealed that there are various kinds of repeated instructions in the source code, and many clones are detected in them with a token-based approach [23]. Consequently, ignoring repeated instructions prevents many uninteresting clones from being detected. This chapter proposes a new clone detection technique focusing on not detecting uninteresting clones in repeated instructions. The contributions of this chapter are as follows:

- this chapter proposes a pre-processing of clone detection producing less uninteresting clones, and it has been developed as a token-based clone detector, FRISC;
- we evaluated the proposed technique on multiple open source software systems, and confirmed that the usefulness of the proposed technique.

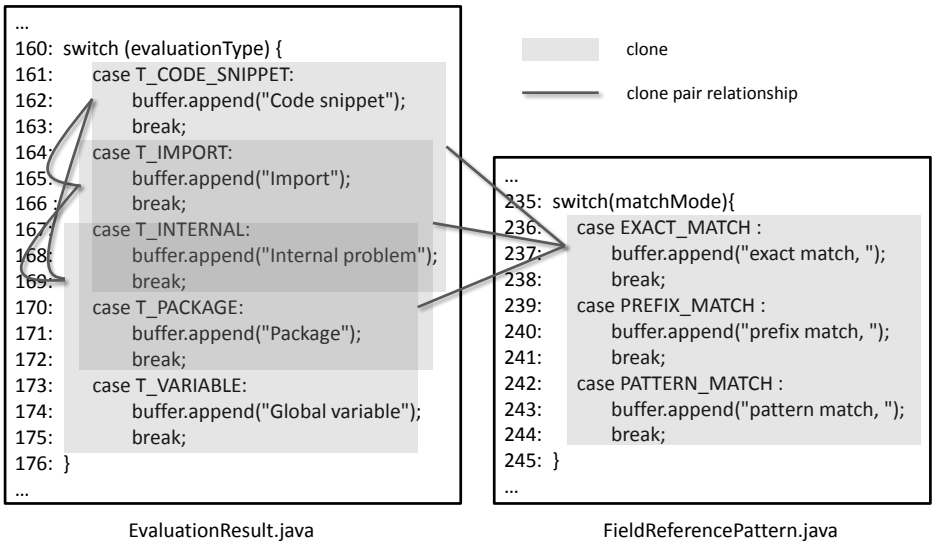


Figure 3.1: Motivating Example, Which Shows That Many Uninteresting Clones Are Detected from Repeated Instructions

3.2 Research Motivation

Figure 3.1 shows actual clones detected from repeated instructions. In the left-side source file, there are five `case` entries and three `case` entries exist in the right-side one. If we detect clones with using line/token-based detection tools, we will obtain six clone pairs. Every detected clone is a hatching part in Figure 3.1. As shown in this example, many clones are detected from repeated instructions.

Clones in repeated instructions have the following characteristics:

1. both the clones forming a clone pair are overlapped with each other. There is no reason to detect such a clone pair because both the clones forming it point almost the same locations of the source code;

2. both the clones forming a clone pair overlap with both the clones forming another clone pair. We need not both the clone pairs because they point almost the same locations of the source code.

Detecting all clones having the above characteristics enlarges detection results, so that we become unaware of clones in other parts of the target system.

The proposed technique can resolve the problem. Intuitively, the proposed technique firstly folds repeated instructions, and then it detects clones. By the

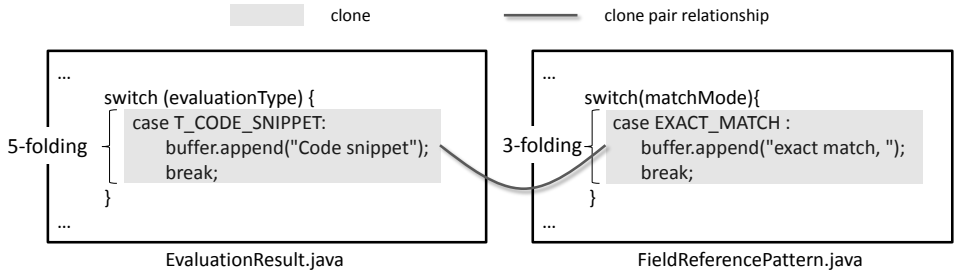


Figure 3.2: Transformed Source Code in Figure 3.1 after Folding Operation

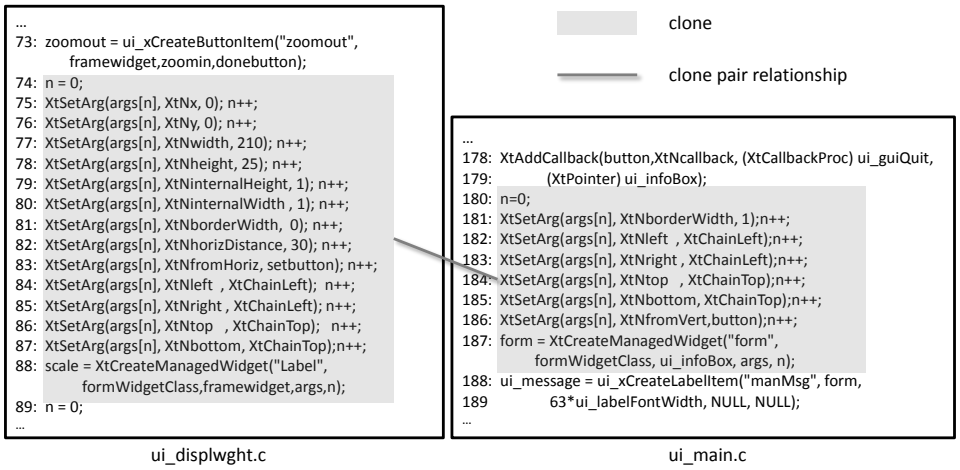


Figure 3.3: Motivating Example, Which Shows that Human Regards Whole Repeated Instructions as Clones

folding operation, the source code in Figure 3.1 is transformed to the source code in Figure 3.2. Consequently, the proposed technique detects only a single clone pair: one is a clone from the 161th line to the 175th line of the left-side source file; the other is a clone from the 236th line to the 244th line of the right-side source file. The proposed technique identifies the two `switch` statements as duplicated code without detecting clones having the characteristics 1 and 2.

Also, Figure 3.3 shows an example of *clone references* judged manually in the Bellon's experiment [9]. As shown in this figure, humans prefer clones covering a whole of the repeated instructions rather than ones covering a part of them. In other words, human does not care the differences of the number of repetitions between

the code fragments.

Herein, we define the following research question in order to confirm that the proposed technique detects fewer uninteresting clones, and it detects more preferable clones.

RQ1: Does folding repeated instructions improve *precision* and *recall* of detection results?

Currently, there is a variety of detection tools. In order to show the usefulness of the proposed technique by comparing them, we define the following research question.

RQ2: Does clone detection with folding repeated instructions have higher accuracy than existing tools?

3.3 Clone Detection with Folding Repeated Instructions

The proposed technique consists of the following five steps.

STEP1: Lexical Analysis and Normalization

STEP2: Generating Statement Hash

STEP3: Folding Repeated Instructions

STEP4: Detecting Identical Hash Sequences

STEP5: Mapping Identical Subsequences to Source Code

The proposed technique takes the followings as its inputs:

- source code;
- maximum elements length (the number of statements);
- minimum clone length (the number of tokens).

The proposed technique outputs a list of detected clone pairs. Figure 3.4 shows an overview of the proposed technique. The remainder of this section explains every of the steps in detail.

STEP1: Lexical Analysis and Normalization

In STEP1, all the target source files are transformed into token sequences. User-defined identifiers are replaced with special tokens to detect similar code fragments as clones even if they include different variables.

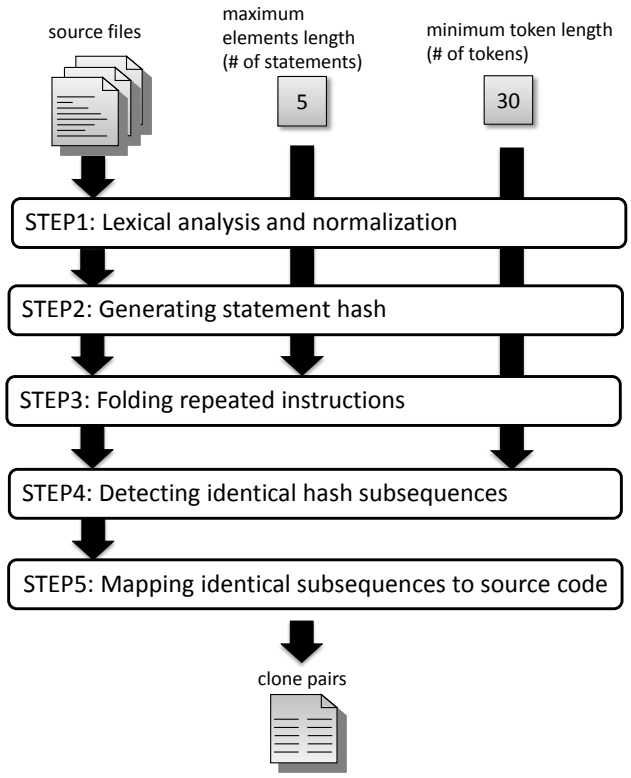


Figure 3.4: Overview of Proposed Technique

STEP2: Generating Statement Hash

In STEP2, 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 ("}"). STEP2 transforms token sequences into hash sequences. Note that every hash has a weight, which means the number of tokens included in its statement.

STEP3: Folding Repeated Instructions

STEP3 is the core of the proposed clone detection technique. Firstly, repeated subsequences are identified. Every of the identified repeated subsequences is divided into the first repeated elements and its subsequent repeated elements. Then, the subsequent repeated elements are removed from the hash sequences. The weights of deleted elements are added to the weights of their first repeated ele-

Algorithm 3.1 Folding Repeated Hash Sequence

Require: $seq, max\_elmt\_length(\geq 1)$
Ensure: folded seq

```
 1: seq_len ← length(seq)
 2: for i = 0 to max_elmt_length do
 3:     left ← 0
 4:     loop
 5:         flg ← true; index ← left; tmpleft ← left; count ← 0;
 6:         while count ≤ i and index < seq_len do
 7:             if isStatementEnd(seq[index]) then
 8:                 if flg then
 9:                     k ← index + 1; flg ← false
10:                 end if
11:                 count ← count + 1
12:             end if
13:             index ← index + 1
14:         end while
15:         if index > seq_len then
16:             break
17:         end if
18:         tmp ← seq[left..index − 1]
19:         count ← 0; left ← index
20:         while count ≤ i and index < seq_len do
21:             if isStatementEnd(seq[index]) then
22:                 count ← count + 1
23:             end if
24:             index ← index + 1
25:         end while
26:         if index > seq_len then
27:             break
28:         end if
29:         tmp2 ← seq[left..index − 1]
30:         if tmp = tmp2 then
31:             seq ← seq[0..left − 1] + seq[index..seq_len]
32:             seq_len ← length(seq); left ← tmpleft
33:         else
34:             left ← k
35:         end if
36:     end loop
37: end for
38: return seq
```

ments. Algorithm 3.1 shows the algorithm used for folding repeated subsequences. In the algorithm, *seq* is a hash sequence, and *max_elmt_length* is a maximum elements length. As a result of the algorithm application, all the repeated subsequences whose elements length is equal to or less than the threshold (the max-

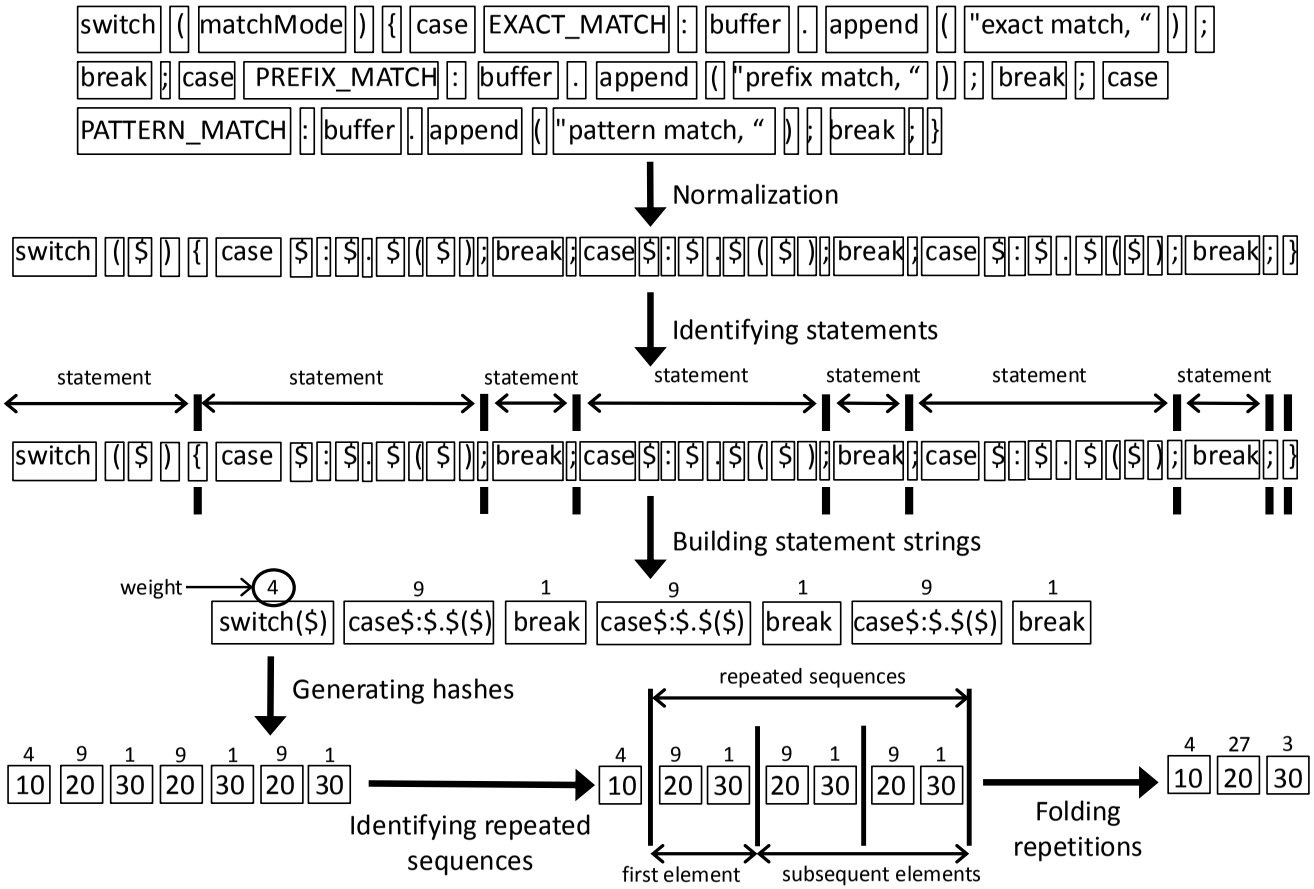


Figure 3.5: How Input Source Code is Transformed into Folded Hash Sequence

imum elements length) are folded. Figure 3.5 shows how input source code is transformed into folded hash sequences. Why we use the threshold is that, if elements of repetitions are large, users might not want to treat them as repetitions. Using the threshold realizes more configurable clone detections.

STEP4: Detecting Identical Hash Subsequences

Identical subsequences are detected from the folded hash sequences. If the sum of weights in an identical subsequence is smaller than the minimum token length, it is discarded.

STEP5: Mapping Identical Subsequences to Source Code

Identified subsequences detected in STEP4 are converted to location information in the source code (file name, start line, end line), which are clone pairs.

3.4 Implementation

We have developed a software tool, FRISC (Folding Repeated Instructions in Source Code), based on the proposed technique. Currently, FRISC can handle Java and C. However, FRISC performs only lexical analysis as a language-dependent processing, so that it is not difficult to extend FRISC to other programming languages.

FRISC supports multi-thread processings. All the steps of the proposed technique except STEP5 are processed in parallel. In STEP1, 2, and 3, every thread takes a source file and outputs its hash sequence one-by-one. This processing is performed for all the target source files. In STEP4, every thread detects identical hash subsequences from a different pair of hash sequences generated in STEP3. Of course, identical hash subsequences within a hash sequence are also detected. Current implementation does not perform STEP5 in parallel because it is relatively a lightweight processing. Hence, the detection speed of FRISC can be shortened with multi-thread processing drastically. FRISC accepts the number of threads as its command line option. FRISC uses some heuristics for identifying more significant clones. Currently, they are as follows:

Shrinking user-defined identifiers connected with ".": By shrinking those identifiers, we can detect clones even if the number of them are different. Figure 3.6 shows a transformation how such identifiers are shrunken.

Removing import and package statements: We do not think that clones in `import` and `package` statements are useful, and so they are removed in STEP1.

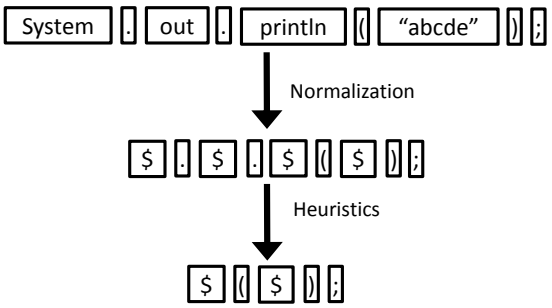


Figure 3.6: Example of Transformation with Heuristics

3.5 Overview of Investigation

We have conducted an investigation to answer the two research questions described in Section 3.2. The investigation consists of two experiments.

Experiment A: clones are detected by FRISC with two settings: one is with the folding operation; the other is without it. Then, *recall* and *precision* of the two detections are calculated and compared.

Experiment B: clones are detected by FRISC and multiple other tools. Then, *recall* and *precision* of all the detection results are calculated and compared.

In order to calculate *recall* and *precision*, we need correct clones. Herein we use Bellon's *clone references* in [60] as a reference set (a set of clones to be detected). Bellon's experiment is described in Chapter 2 in detail.

This experiment has a limitation on *recall* and *precision*. The *clone references* used in the experiments are not all the real clones included in the target systems. Consequently, 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 special 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* not matching any of the *clone references*; however, nobody knows whether they are truly false positives or not.

The remainder of this section summarizes the two experiments for investigating the RQs. The details of each experiment are described in Section 3.6 and 3.7, respectively.

Summary of Experiment A

The *precision* with folding repeated instructions is averagely higher than the one without it by 29.8%. On the other hand, the folding averagely decreased *recall* by 2.9%. The degree of *precision* increasing is about 10 times of the degree of *recall* decreasing.

Summary of Experiment B

FRISC detected more *clone references* than any of the comparison tools in most cases. Especially, for five out of the eight systems, both the *precision* and *recall* of FRISC are greater than those of CCFinder, which is one of the most widely-used detection tools. Still, the *precision* of FRISC is lower than those of CloneDR [8] and CLAN [53] for all the target systems.

3.6 Experiment A

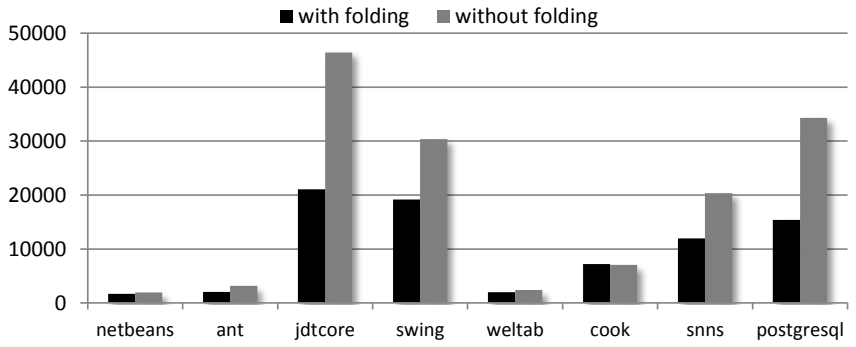
The purpose of Experiment A is to reveal how the number of *clone candidates*, *precision*, and *recall* are changed by folding repeated instructions. In this experiment, we used the following thresholds.

Maximum elements length: 5 is for detection with the folding operation, and 0 is for detection without it.

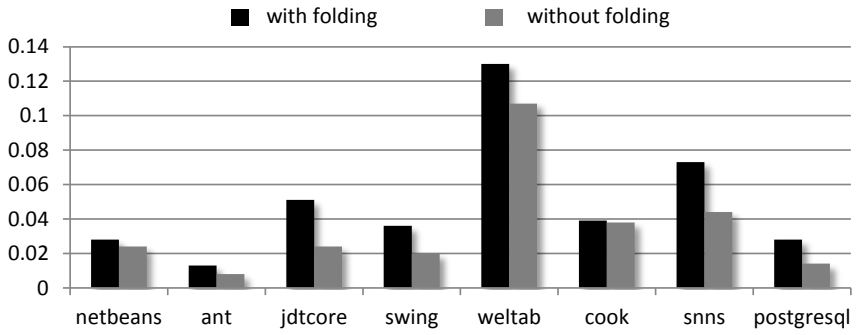
Minimum clone length: 30

Figure 3.7(a) shows the number of *clone candidates*. The folding decreases the number of *clone candidates* for almost all the target software. Especially, for `jdtcore`, which is the software where most *clone candidates* were detected, the number of *clone candidates* dropped by about 54%. We browsed the source code of `jdtcore`, and found that it includes a large number of consecutive `if-else` statements, consecutive `case` entries in `switch` statements, and consecutive `catch` statements. The folding prevented clones from being detected from those repetitions in the source code.

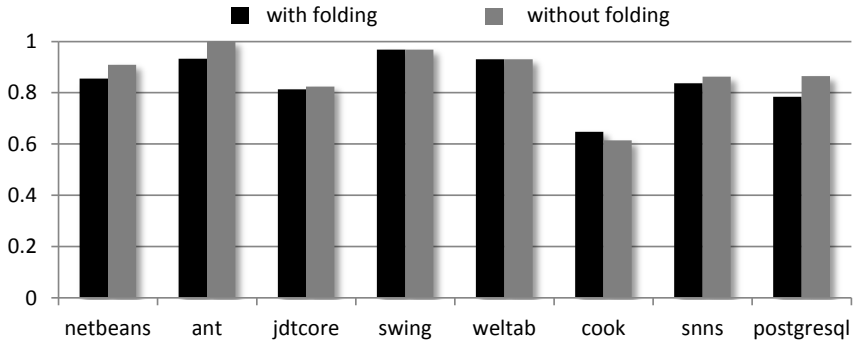
Average decreasing rate of *clone candidates* was about 32%, and we found that, in the case of `cook`, the number of *clone candidates* was slightly increased. Figure 3.8 shows a clone reference newly detected by using the folding operation. If we do not use the folding operation, the code fragment from the 28th line to the 31st line of the left-side source file is a clone of the code fragment from the 118th line to the 121st line of the right-side source file. However, the length of the code fragment is 26 tokens, which is less than the minimum token length, 30. Hence, the



(a) # of Clone Candidates



(b) *Precision*



(c) *Recall*

Figure 3.7: # of *Clone Candidates*, *Precision* and *Recall* in Experiment A

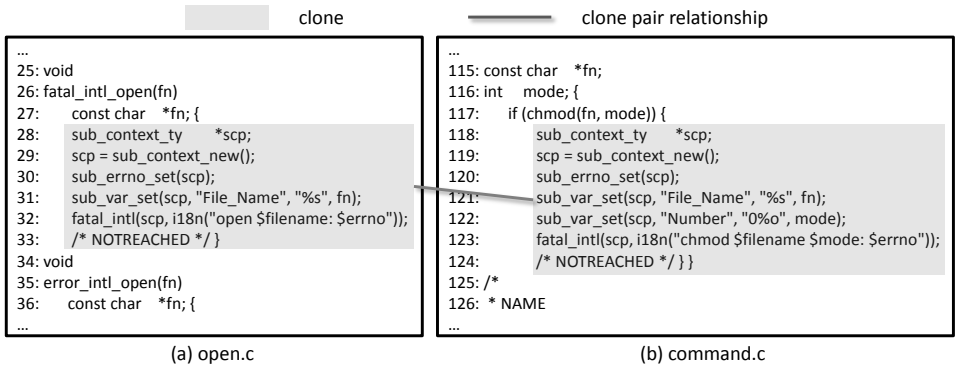


Figure 3.8: *Clone Reference* Newly Detected by Using Folding Operation

clone pair was discarded. On the other hand, when we used the folding operation, the code fragment from the 28th line to the 32nd line of the left-side source file was a clone of the code fragment from the 118th line to the 123rd line of the right-side source file. The code fragments are greater than the minimum token length, so that the clone pair was output. In `cook`, there are those clone pairs, so that the number of *clone references* with the folding operation is greater than the number of *clone references* without it. We expect that there are such clone pairs in the other target systems too.

Figure 3.7(b) shows the *precision* of the detections with and without the folding. For all of the software, *precision* was improved. We confirmed the followings:

- in the best case, *precision* increased by 53.8%,
- even in the worst case, *precision* increased by 2.6%,
- averagely, *precision* increased by 29.8%.

Figure 3.7(c) shows *recall* of the detections with and without the folding. The changes of *recall* varied from the target software, unlike *precision*. For system `cook`, *recall* was improved by the folding. More *clone references* were detected with the folding operation. Also, for two systems, `swing` and `weltab`, *recall* remained unchanged. However, for the other five systems, *recall* was decreased. Averagely, *recall* dropped by 2.9%.

We investigated clones that were detected without the folding but not detected with the folding to know why they became undetected. Table 3.1 shows the proportion of self-overlapping clones that became undetected. It is easy to remove

self-overlapping clones even if the folding operation is not applied. For example, after detecting clones, checking whether locations of code fragments of a clone pair is overlapped or not is a simple way. The proportions of self-overlapping clones are very different from software systems, which means that checking the self-overlapping as a post processing of clone detection is not enough to reduce uninteresting clones.

Consequently we answer RQ1 as follows: using the folding operation decreased the number of *clone candidates* by about 32% averagely. The decreasing caused the improvement of *precision*, averagely 29.8%. However, it also caused missing some *clone references*. The average of decreasing *recall* is 2.9%. We can conclude that the folding is a useful approach to prevent uninteresting clones from being detected meanwhile it misses some *clone references*.

3.7 Experiment B

The purpose of Experiment B is to reveal whether FRISC detects clones more precisely than existing tools or not. In this experiment, we chose the additional clone detector NiCad to Bellon's experiment.

Table 3.2 shows the number of *clone candidates* detected by the tools. The number of *clone candidates* considerably varies from tool to tool. Also, we can see that line/token-based tools found many more *clone candidates* than the other tools.

Figure 3.9 shows *precision* and *recall* of all the tools on all the target systems. The *recall* of FRISC is the best in all the tools for five out of the eight systems. FRISC could detect most *clone references* for the systems. For two of the remaining systems, `ant` and `snns`, FRISC placed the second position. In the worst case, `cook`, FRISC placed the third position.

Table 3.1: Rate of Self-Overlapping Clones That Became Undetected by Using Folding Operations

Software Name	Self-Overlapping
`netbeans`	75.2%
`ant`	15.1%
`jdtcore`	43.8%
`swing`	37.8%
`weltab`	52.7%
`cook`	89.1%
`snns`	75.5%
`postgresql`	79.5%

In order to reveal what kinds of *clone references* detected by the comparison tools were not detected by FRISC, we extracted all of those *clone references* from all the target systems. Then, we randomly selected 100 instances from them, and we browsed their source code. As a result, they were categorized as follows (the numbers in parentheses mean the number of clone pairs falling into the category):

A(71): *clone references* including some gaps;

B(17): *clone references* being less than 30 tokens;

C(11): *clone references* locating in repeated instructions;

D(1): *clone references* including unmatched modifiers.

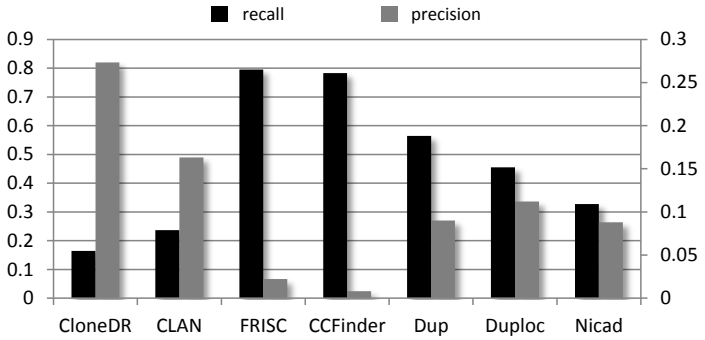
In token-based detections, identical subsequences are detected as clones. Gapped (Type-3) clones are not detected by naive token-based detections. Consequently, it is quite natural that 71 *clone references* falling into category A were not detected by FRISC. However, if we adopt some techniques like Roy et al. [65] or Juergens et al. [35] to detect such *clone references*, FRISC may detect some of those *clone references*.

Clone references falling into category B are smaller than 30 tokens. In Bellon's experiment, the minimum threshold of *clone references* is six lines, which is not a token-based threshold but a line-based one. However, FRISC takes a token-based threshold. In this experiment, FRISC took 30 tokens as the minimum clone length. Consequently, some *clone references* were not detected by FRISC.

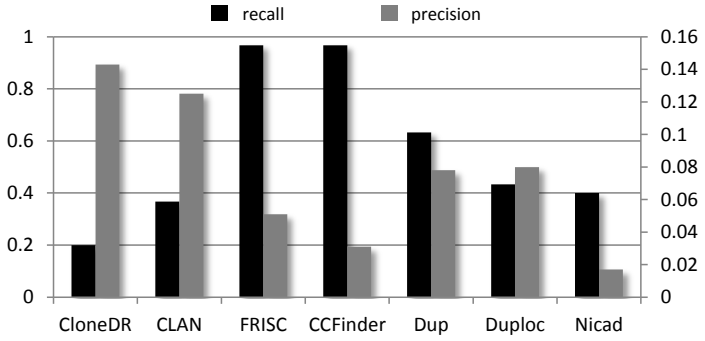
Figure 3.10 shows an example of a *clone reference* falling into category C. There are six `case` entries in a `switch` statement. The former three entries form a clone of the latter three entries. The proposed technique folds the six `case` entries into a single entry, so that no clone pair is detected. However, it is possible

Table 3.2: # of *Clone Candidates*. Every "-" Means That Detector Could not Finish Clone Detection Because of Scalability Issue.

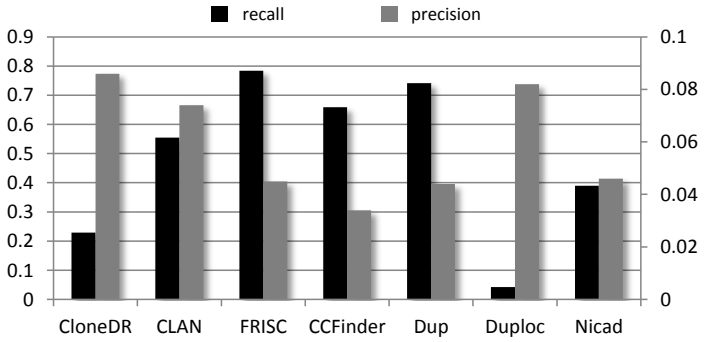
Software Name	FRISC	Dup	CloneDR	CCFinder	CLAN	Duploc	Duplix	Nicad
`netbeans`	1,696	344	33	5,552	80	223	-	24
`ant`	2,106	245	42	950	88	162	-	19
`jdtcore`	21,494	22,589	3,593	26,049	10,111	710	-	1,142
`swing`	20,606	7,220	3,766	21,421	2,809	-	-	1,804
`weltab`	1,969	2,742	186	3,898	101	1,754	1,201	160
`cook`	7,222	8,593	1,438	2,964	449	8,706	2,135	159
`snns`	11,940	8,978	1,434	18,961	318	5,212	12,181	352
`postgresql`	15,362	12,965	1,452	21,383	930	-	-	352



(a) `netbeans`

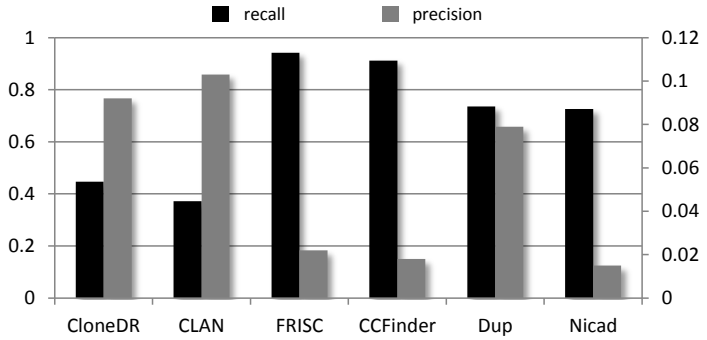


(b) `ant`

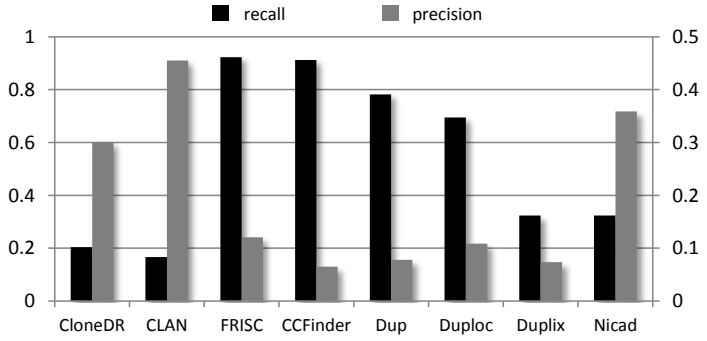


(c) `jdtcore`

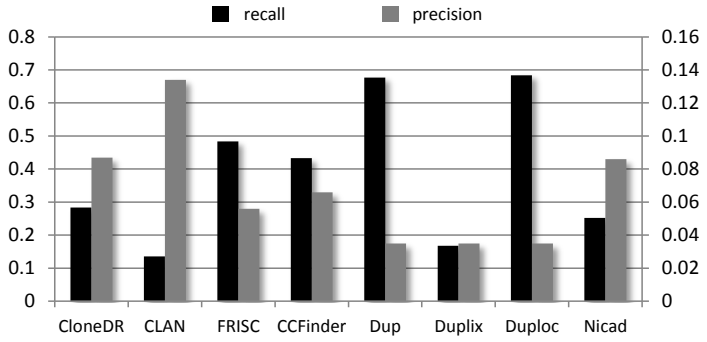
Figure 3.9: *Precision* and *Recall* of All Detectors for All Target Software Systems



(d) `swing`

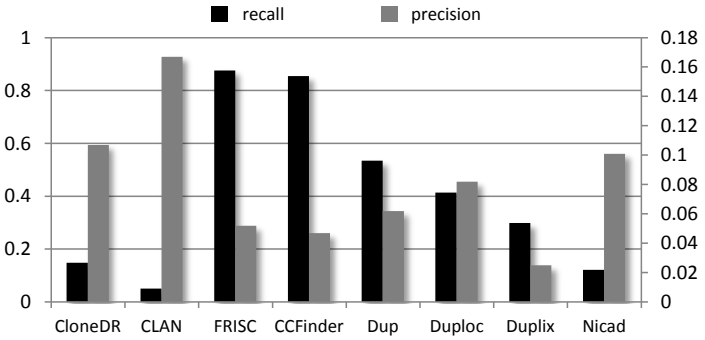


(e) `weltab`

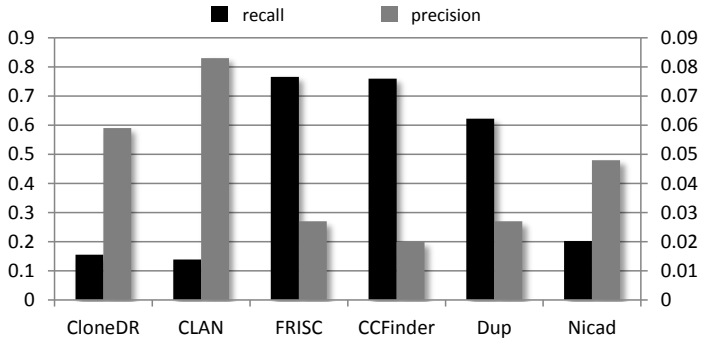


(f) `cook`

Figure 3.9: *Precision* and *Recall* of All Detectors for All Target Software Systems



(g) `snns`



(h) `postgresql`

Figure 3.9: $Precision$ and $Recall$ of All Detectors for All Target Software Systems

to detect such a clone pair with the proposed technique. If the sum of weights of a folded sequence is more than twice of the minimum token length, there is a clone pair in it.

A *clone reference* was not detected because there is an unmatched modifier in it (category D): a clone has "`final`" modifier in a method declaration; the correspondent does not have. Currently, FRISC does not normalize modifiers. However, it is not difficult to remove modifiers so as to detect such a *clone reference*.

The result shows that *precision* of the proposed technique is not so high as the other token-based tools. However, we must notice that *clone references* used in this experiment is not all the real clones in the systems. A low *precision* using the *clone references* does not directly mean that its detection result includes many false positives.

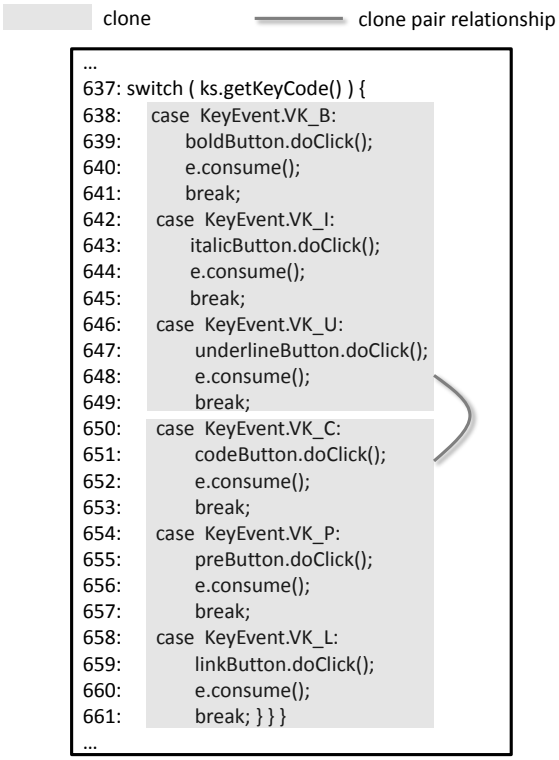


Figure 3.10: *Clone Reference* Located in Repeated Instructions

We answer RQ2 as follows: the proposed technique could detect more *clone references* than any of the other detection tools used for comparison in most cases. However, it detects many *clone candidates* as well as other token-based detection tools. Therefore, it may detects many uninteresting clones.

3.8 Conclusion

In this chapter, we proposed a new token-based clone detection technique. The proposed technique folds repeated instructions for preventing uninteresting clones from being detected. There are some techniques that perform filtering after detecting clones for removing uninteresting clones [23, 44]. On the other hand, the proposed technique performs folding operations for the same purpose. The proposed technique has two advantages. One is that the proposed technique does not have cost for detecting uninteresting clones. The other is that the proposed technique can find clones that the traditional token-based techniques cannot detect. The

proposed technique was developed as an actual tool, FRISC. We applied FRISC to eight open source software systems, and we confirmed the followings:

- the folding operation reduces uninteresting clones;
- there are some *clone references* newly detected by the folding operation;
- FRISC detects more *clone references* than any other comparison tools used in Bellon's benchmark in most cases; and still,
- FRISC detects uninteresting clones as well as other token-based clone detectors.

In the future, we are going to investigate where tools detect uninteresting clones. We expect that most uninteresting clones are detected from limited kinds of code patterns. If we can ignore clones detected from those code patterns, uninteresting clones are drastically decreased.

Chapter 4

Clone Detection Using Smith-Waterman Algorithm

4.1 Background

Recently, clones have received much attention. It has been said that the presence of clones makes software maintenance more difficult [19]. This is because if developers modify a code fragment, it is necessary to check its correspondents and verify whether they need the same modifications simultaneously or not. On the other hand, reusing code fragments by copy-and-paste operations has some advantages. One of them is that programmers can implement functions easily that is similar to existing one.

Moreover, recently many studies have investigated the premise of "the presence of 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 clones. However, to summarise this point then it is said "not all clones make software maintenance more difficult" [36]. Consequently, removing or not generating all clones are inappropriate from the perspective of efficient software development or maintenance. It is important to minimize the risk of clones with low cost.

From a viewpoint of program comprehension, analysis of clones plays an important role. Refactoring and removing some clones can improve readability, maintainability and manageability of software systems [19].

Developers often make some changes to code fragments after cloning to adjust the code fragments to the destination of the cloning [41]. Moreover, cloned fragments often evolve differently from the original fragments [20]. These facts indicate that there often exists some gaps between the original code fragments and

pasted fragments. In order to detect clones appropriately, it is necessary to detect clones even if they include some gaps. In other words, detecting Type-3 clones is required for better understanding of clones and software systems.

As described in Chapter 2, a number of techniques detecting clones have been proposed before now. In those detection techniques, text-based techniques using the LCS algorithm, AST-based techniques, PDG-based techniques and metric-based technique can detect Type-3 clones. However, each of them has limitations as described previously. In order to resolve those limitations, we propose a clone detection technique using the Smith-Waterman algorithm [75]. The proposed technique detects not only Type-1 and Type-2 but also Type-3 clones in a shorter time frame than the AST-based or PDG-based techniques. The reason is that the proposed technique does not use any intermediate representations such as ASTs or PDGs. Furthermore, the proposed technique detects 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 technique performs a fine-grained detection that identifies statement-based clones.

We implemented the proposed technique and evaluated it by using Bellon's benchmark [9]. However, Bellon's benchmark has a limitation that the Type-3 *clone references* does not have the information about where gaps are. Bellon's *clone references* represent clones with only the information about where they start and where they end. We do not consider that gapped parts of clones should be regarded as clones. Thus, Bellon's benchmark is likely to evaluate Type-3 clones incorrectly when it is used as-is. Therefore, we remade the *clone references* with the information about where gaps are. Moreover, we compared the result by using Bellon's *clone references* with that by using the enhanced *clone references*. Finally, the proposed technique was compared with the existing techniques by using the enhanced *clone references*.

Consequently, the contributions of this chapter are as following:

- We tailored the Smith-Waterman algorithm to 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 clones with consideration for the size of them or 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 clones start and they end.

- We confirmed that the proposed technique had higher $F\text{-}measure$ than the existing techniques.

The rest of this chapter is organized as follows: Section 4.2 introduces the concept of the Smith-Waterman algorithm. We provide an overall summary of the proposed technique in Section 4.3. Section 4.4 describes the overview of investigation, then Section 4.5, Section 4.6 and Section 4.7 report the evaluations of the proposed technique in detail. Section 4.8 describes threats to validity. Section 4.9 discusses the experimental result or previous techniques. Section 4.10 summarizes this chapter and refers to the future work.

4.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm [75] 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 4.1 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 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.

$$v_{i,j}(2 \leq i, 2 \leq j) = max \begin{cases} v_{i-1,j-1} + s(a_i, b_j), \\ v_{i-1,j} + gap, \\ v_{i,j-1} + gap, \\ 0. \end{cases} \tag{4.1}$$

$$s(a_i, b_j) = \begin{cases} match & (a_i = b_j), \\ mismatch & (a_i \neq b_j). \end{cases} \tag{4.2}$$

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.

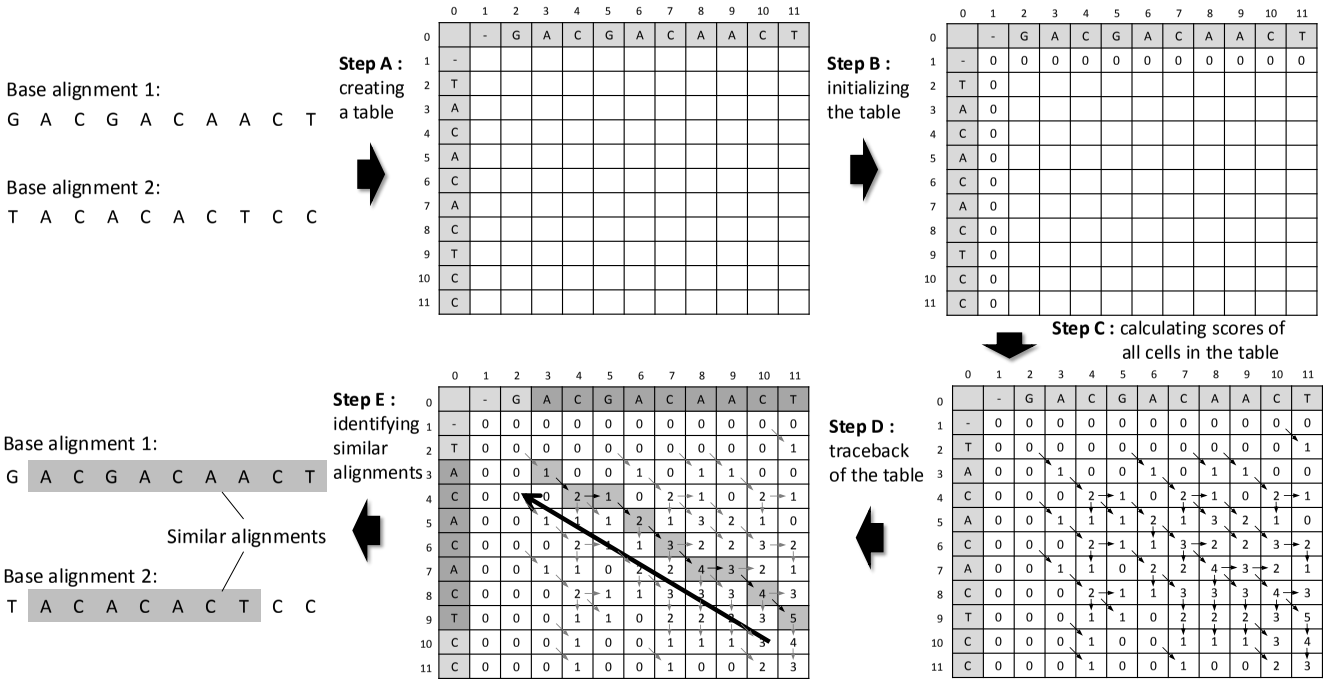


Figure 4.1: Smith-Waterman Algorithm Applied to Two Base Sequences, “GACGACAACT” and “TACACACTCC”

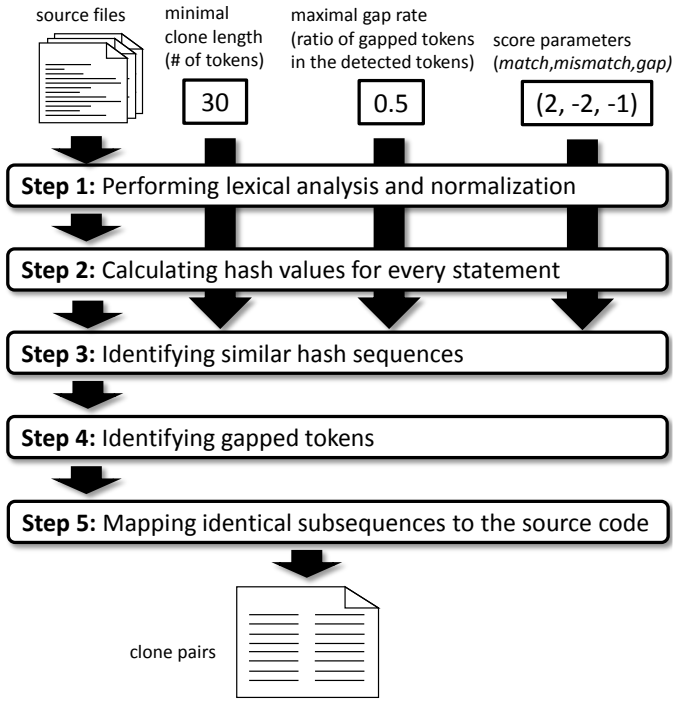


Figure 4.2: Overview of Proposed Technique

$Match$, $mismatch$ and gap can be set all kinds of values freely. In Figure 4.1, 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 4.1, $v_{9,11}(= 5)$ 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 4.1, 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.

4.3 Proposed Technique

The proposed technique 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 chapter, score parameters ($match$, $mismatch$ and gap) were decided by a preliminary experiment. Section 4.5 reports how to decide these parameters.

The proposed technique outputs a list of detected clone pairs. The proposed technique 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 Source Code

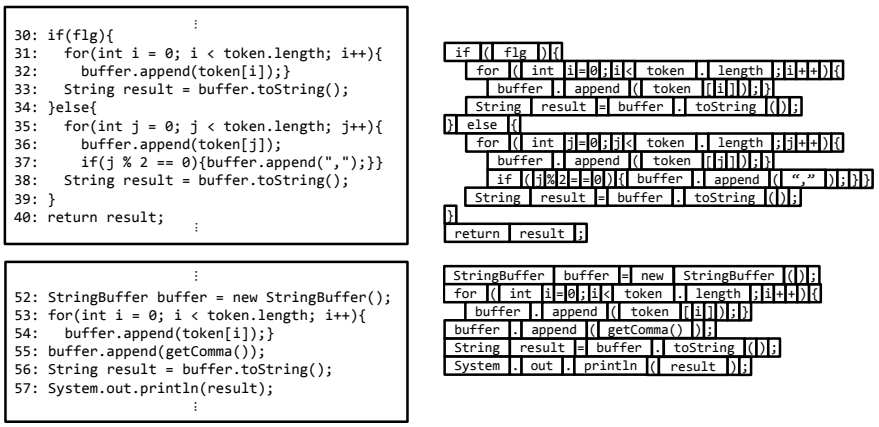
Figure 4.2 shows an overview of the proposed technique. Figure 4.3 shows an example of detection process using the proposed technique. The remainder of this section explains each step in detail.

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 clones even if they include different variables. All modifiers are deleted for the same reason.

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.



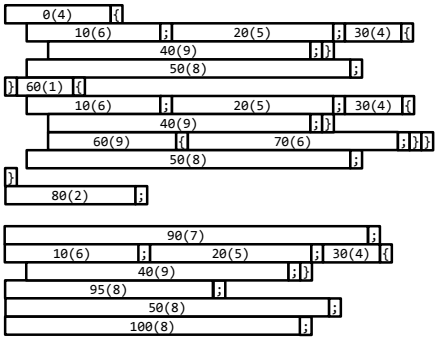
(a) Sample Source Files

(b) Lexical Analysis

(c) Normalization

(d) Identifying Statements

Figure 4.3: Example of Detection Process Using Proposed Technique



(e) Calculating Hash Values

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	95	50	100

# of tokens sequence2

7 6 5 4 9 8 8 8

(f) Generating Hash Sequences

	-	90	10	20	30	40	95	50	100
-	0	0	0	0	0	0	0	0	0
0	0								
10	0								
20	0								
30	0								
40	0								
50	0								
60	0								
10	0								
20	0								
30	0								
40	0								
60	0								
70	0								
50	0								
80	0								

(g) Creating Table

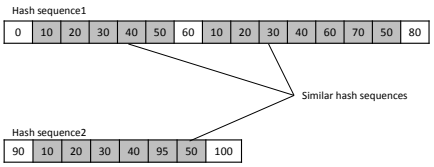
	-	90	10	20	30	40	95	50	100
-	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
10	0	0	1	0	0	0	0	0	0
20	0	0	0	2	1	0	0	0	0
30	0	0	0	1	3	2	1	0	0
40	0	0	0	0	2	4	3	2	1
50	0	0	0	0	1	3	3	4	3
60	0	0	0	0	0	2	2	3	2
10	0	0	1	0	0	1	1	2	1
20	0	0	0	2	0	0	0	1	0
30	0	0	0	1	3	2	1	0	0
40	0	0	0	0	2	4	3	2	1
60	0	0	0	0	1	3	3	2	1
70	0	0	0	0	0	2	2	1	0
50	0	0	0	0	0	1	1	3	0
80	0	0	0	0	0	0	0	0	0

(h) Calculating Scores

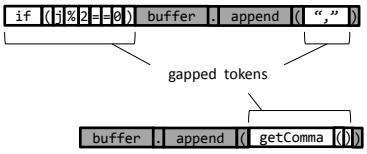
Figure 4.3: Example of Detection Process Using Proposed Technique

	-	90	10	20	30	40	95	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	3	2	1
70	0	0	0	0	0	2	2	1	0
50	0	0	0	0	0	1	1	3	0
80	0	0	0	0	0	0	0	0	0

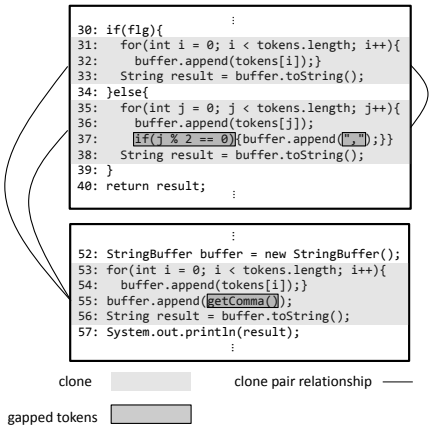
(i) Traceback



(j) Identifying Similar Hash Sequences



(k) Identifying Gapped Tokens



(l) Detected Clones

Figure 4.3: Example of Detection Process Using Proposed Technique

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 4.2 to tailor the algorithm for clone detection.

- 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 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.

  $$S = \{c_{m,n} \mid k \leq m \leq i \wedge l \leq n \leq j\} \tag{4.3}$$

  The purpose of reducing the scope of traceback is in order not to detect redundant clones.

- The number of tokens and gaps are counted during traceback in order to detect 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.

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.

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.

Step 5: Mapping Identical Subsequences to 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

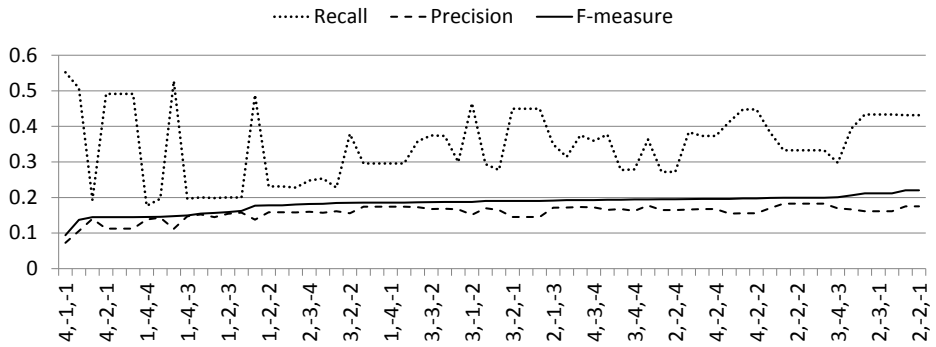


Figure 4.4: $Recall$, $Precision$ and $F\text{-}measure$ on 3-Tuple of Parameters ($match$, $mismatch$, gap).

pairs. Note that the gapped line represents the line that contains the gapped tokens. This representation can make sense because the location information of clones are represented by line numbers.

4.4 Experimental Design

We have developed a software tool, CDSW based on the proposed technique described in Section 4.3. 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 technique.

RQ 1: Does evaluating $recall$ and $precision$ with gap information have higher accuracy than without it?

RQ 2: Does the proposed technique have higher accuracy than existing techniques?

RQ 3: Can the proposed technique finish detecting clones from large-scale software systems?

Experiment A investigates RQ1 and Experiment B investigates RQ2 and RQ3, respectively. In order to calculate accuracy, reference clones are necessary. In this chapter, we used Bellon's dataset for the experiments as well as Chapter 3. The details of each Experiment are described in Section 4.5, Section 4.6 and 4.7, respectively.

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

$$match = \{x \subset \mathbb{Z} \mid 1 \leq x \leq 4\} \tag{4.4}$$
$$mismatch = \{y \subset \mathbb{Z} \mid -4 \leq y \leq -1\} \tag{4.5}$$
$$gap = \{z \subset \mathbb{Z} \mid -4 \leq z \leq -1\} \tag{4.6}$$

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 4.4 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 4.4, 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 references* are likely to be contained in *clone candidates*. Meanwhile, $precision$ 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 some *clone references* relatively.

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

4.6 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 [9], in order to determine whether a candidate matches a reference, ok value and $good$ value are used. However, these formulae do not consider the gapped fragments included in clones. Therefore, we remade the *clone references* with information of gapped lines and

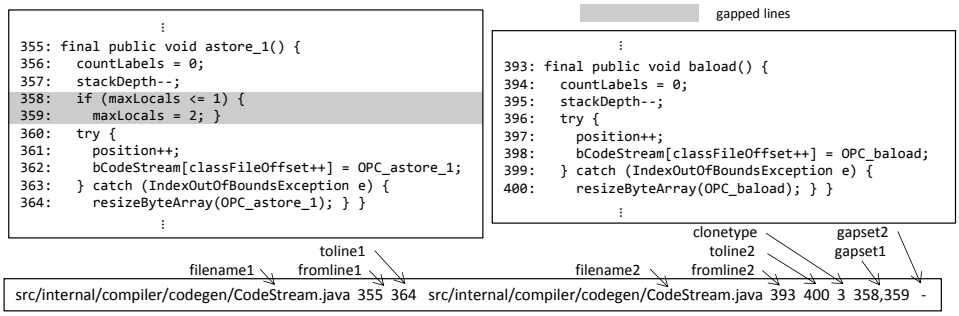


Figure 4.5: Example of Enhanced *Clone Reference* (*Clone Reference* No. 1101).

made it public on the website[1]. Furthermore, we put the file format of our *clone references* on the same website.

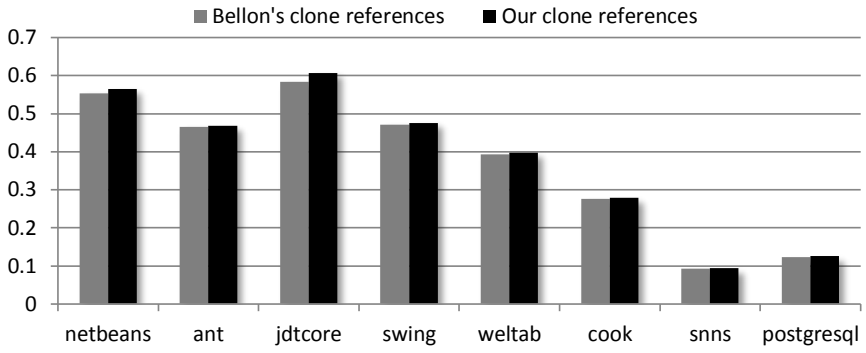
Figure 4.5 shows an example of our *clone references*. In Figure 4.5, the source file in the left has gapped lines 358-359. On the other hand, right one has no gapped lines.

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 Bellon's *clone references*, some Type-3 clones contain gapped lines because Bellon's *clone references* have only the information about where clones start and where they end. Meanwhile, in the case of our *clone references*, all the clones do not contain gapped lines. In other words, our *clone references* consist of true clones. Thus, evaluations using our *clone references* enable us to obtain true $recall$, $precision$ and $F\text{-}measure$.

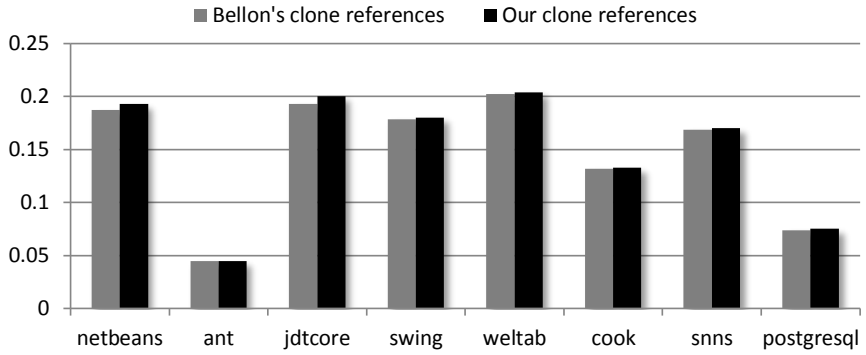
We calculated $recall$, $precision$ and $F\text{-}measure$ using Bellon's and our *clone references*. Figure 4.6(a), (b) and (c) shows $recall$, $precision$ and $F\text{-}measure$ of CDSW using both the *clone references*, respectively. For all of the software, $recall$, $precision$ and $F\text{-}measure$ 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 clones start and where they end but also the information about where the gaps are could evaluate clones more precisely.

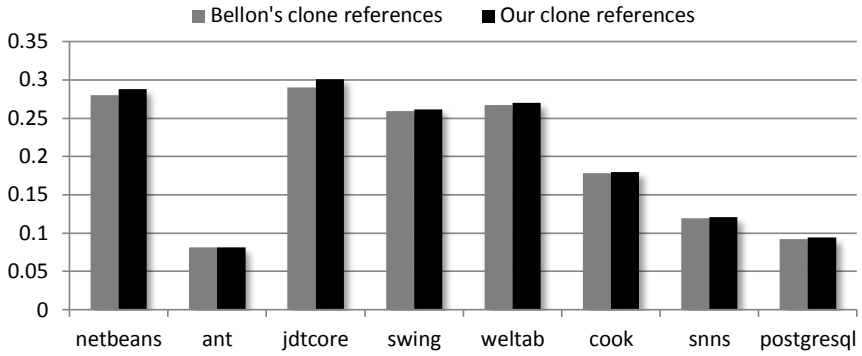
[1] `http://sdl.ist.osaka-u.ac.jp/~h-murakm/2014_clone_references_with_gaps/`



(a) *Recall*



(b) *Precision*



(c) *F-measure*

Figure 4.6: *Recall*, *Precision* and *F-measure* of CDSW Using Both *Clone References*

4.7 Experiment B

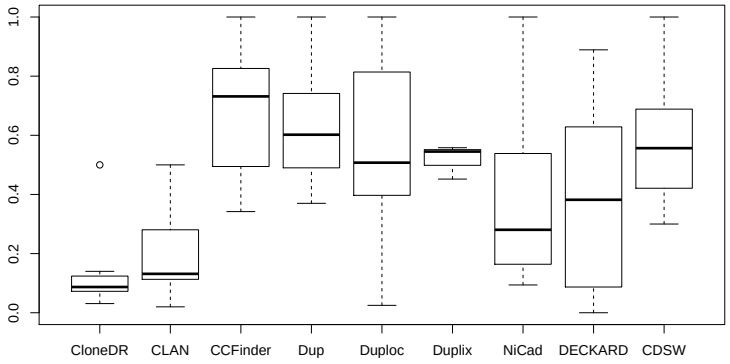
One purpose of Experiment B is to reveal whether CDSW detects clones more accurately than existing clone detectors or not. The other purpose is to reveal that CDSW detects clones in practical time. In this experiment, we chose the additional clone detector NiCad and DECKARD to Bellon's experiment. We calculated $recall$ and $precision$ of all the clone detectors by using our *clone reference* with information of gapped lines. In section 4.3, we described that CDSW outputs gapped lines in 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 clones. Therefore, we only use the information about where clones start and where they end.

Figure 4.7(a) shows $recall$ 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 4.7(b) shows the case of $precision$. CLAN gets the first position, and CDSW gets the second. Figure 4.7(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 a achieves good balance of $recall$ and $precision$ for the Type-3 *clone references*.

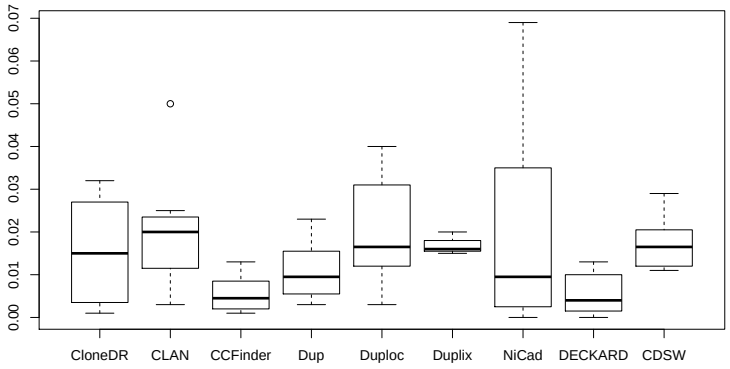
Figure 4.8(a) shows $recall$ of all the clone detectors for Type-1, Type-2 and Type-3 *clone references*. The median of CDSW is the fifth position behind CCFinder, Dup, Duploc and DECKARD. Figure 4.8(b) shows the case of $precision$. CDSW is the best, and that of CLAN is the next by a mere touch. Figure 4.8(c) shows the case of $F\text{-}measure$. 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 DECKARD, NiCad and CDSW. The reason why we selected these three clone detectors is that they can detect Type-3 clones and are available now. Figure 4.9 shows the execution time to detect clones in target software systems. CDSW detect clones in the shortest time in them. Moreover, we applied CDSW to the latest `PostgreSQL` (version 9.2.3, 839 files, 930,524 line of code). CDSW could detect clones from the software in 7 minutes 30 seconds. Thus, we confirmed that CDSW is fast clone detector.

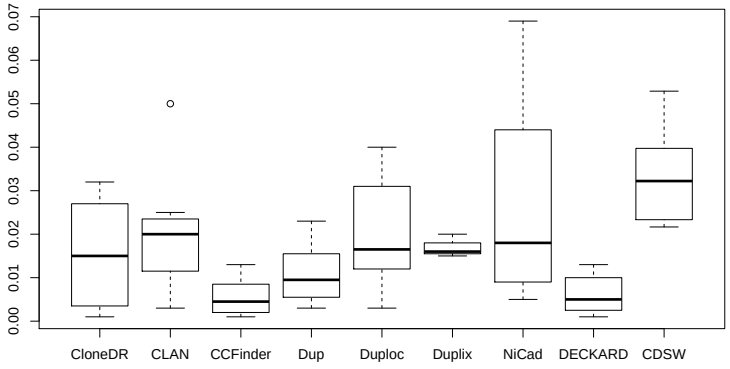
Consequently we answer RQ 2 as follows: CDSW is the best in all the clone detectors used in Bellon's benchmark in the case of $F\text{-}measure$. Since $F\text{-}measure$ is harmonic average of $recall$ and $precision$, it would be said that CDSW has higher accuracy than the existing techniques.



(a) $Recall$

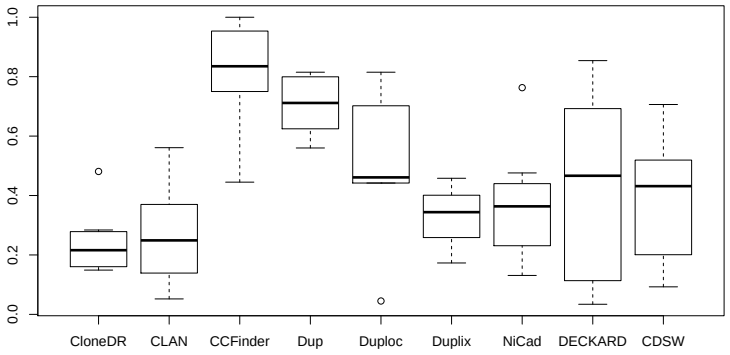


(b) $Precision$

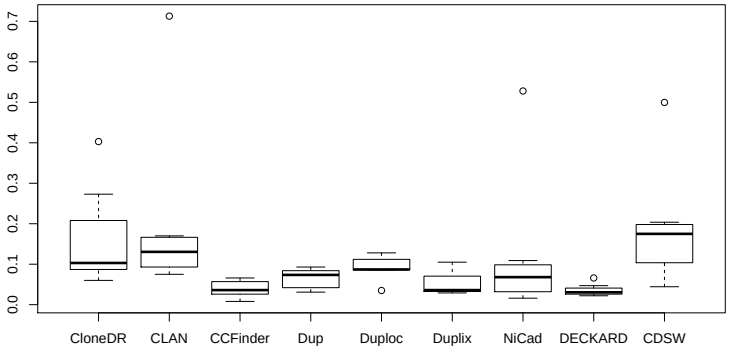


(c) $F\text{-}measure$

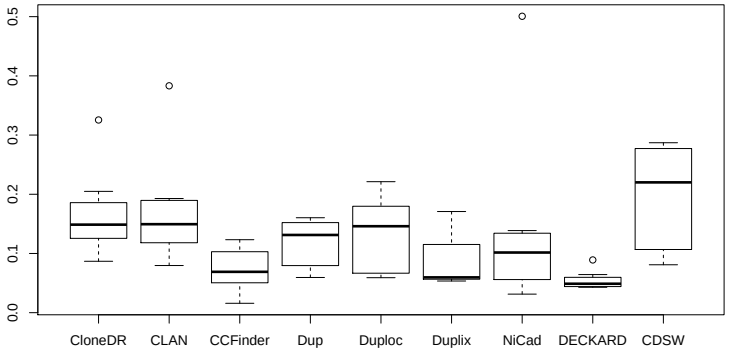
Figure 4.7: $Recall$, $Precision$ and $F\text{-}measure$ for Type-3 *Clone References*



(a) *Recall*



(b) *Precision*



(c) *F-measure*

Figure 4.8: *Recall*, *Precision* and *F-measure* for Type-1, Type-2 and Type-3 *Clone References*

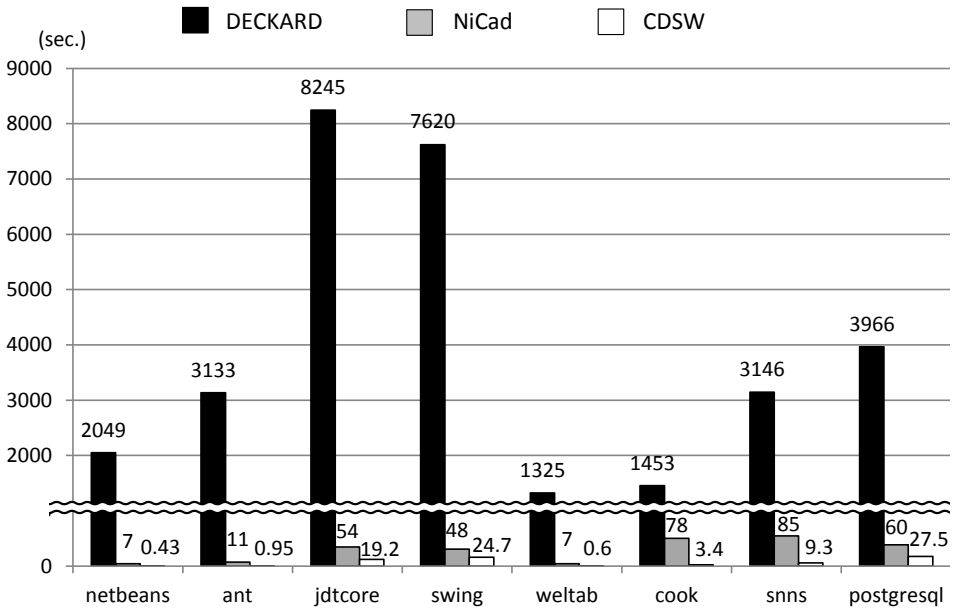


Figure 4.9: Exection Time of DECKARD, NiCad and CDSW for Target Software Systems

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

4.8 Threats to Validity

4.8.1 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 clones in target software systems. Therefore, if all the 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 the clones in target software systems.

4.8.2 Code Normalization

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

4.8.3 Three Parameters in Smith-Waterman Algorithm

In this chapter, we investigated appropriate parameters ($match$, $mismatch$, gap), then compared the accuracy of CDSW with that of existing clone detectors. If these three parameters were calculated by other ways, experimental results would be changed. For example, changing gap parameter constantly according to the length of code fragments might be possible.

4.9 Discussion

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 clones in consideration for the information of their gapped lines. Moreover, the proposed technique makes some changes to the Smith-Waterman algorithm as described in section 4.3. The changes enable the Smith-Waterman algorithm to detect one or more similar subsequences from two sequences. Therefore, the proposed technique 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 [10]. Moreover, implementations of the Smith-Waterman algorithm on graphics processing units (GPUs) were proposed [39]. If we use GPUs for implementation of CDSW, the detection time would be reduced.

4.10 Conclusion

This chapter proposed a new technique to detect not only Type-1 and Type-2 but also Type-3 clones by using the Smith-Waterman algorithm. The proposed technique was developed as a software tool, CDSW. We 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, we 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 we remade. The following items are confirmed.

- We tailored the Smith-Waterman algorithm for clone detection.
- The accuracy of clone detection results improved by using not only the information about where clones start and where they end but also the 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 good balance of true positives and false positives.
- CDSW detected clones in short time for large-scale software systems.

As described in section 4.6, in this experiment, the information of gapped lines that CDSW outputs was not used for accuracy comparison of clone detectors. In the future, we are going to conduct experiments using the information about where gaps are. If the information of gapped lines are used for evaluating clones, more accurate results would be obtained.

Chapter 5

Clone Visualization Using Circle Packing

5.1 Introduction

Recent research has revealed that some clones make software maintenances more difficult [11]. For example, if developers modify a code fragment for fixing a bug or adding a new function, developers have to check whether its clones need the same modification or not. In order to find clones of a given code fragment, some researchers have developed tools that took a code fragment as input and took its clones as output. Libra [27] is one of such tools. Libra receives a code fragment from a user, and uses CCFinder [37] to detect clones of the input code fragment. Libra has two views when the user browses the detected clones. One is a tree view representing all files that are targeted for the clone detection, and the other is a source code view representing source code that is selected by the user. The detected clones are highlighted in the source code view. However, we consider that Libra has an issue. Developers using Libra cannot browse detected clones efficiently because developers have to open a number of source code and move up/down a scroll bar for browsing all detected clones. We consider that the source code of detected clones should be viewable easily. It is necessary to understand detected clones to a certain extent (e.g. which types of clones are detected?) without browsing source code.

In order to resolve this issue, we developed a clone set visualization tool, named ClonePacker. ClonePacker uses Circle Packing [62] for visualizing detected clones. We evaluated ClonePacker by comparing with Libra through an experiment with participants. In the experiment, the participants reported the locations of clones by using ClonePacker and Libra, then we compared the reporting time

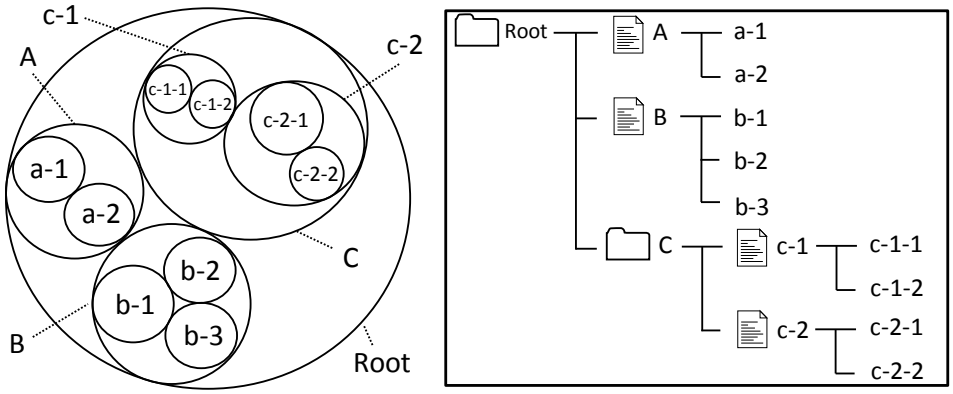


Figure 5.1: Example of Circle Packing

between the tools. Moreover, we compared ClonePacker with Libra from the perspective of their usability by using System Usability Scale [6]. Consequently, the contributions of this chapter are followings.

- We proposed a technique to visualize detected clones and developed the proposed technique as a tool, named ClonePacker. Programmers using ClonePacker can understand detected clones to a certain extent without browsing source code.
- We confirmed that developers using ClonePacker reported the locations of clones faster than Libra and the accuracy was unchanged.
- We confirmed that ClonePacker has higher usability than Libra.

The remainder of this chapter is organized as follows: Section 5.2 describes Circle Packing. Sections 5.3 and 5.4 show the proposed technique and details of ClonePacker. Section 5.5 reports the evaluations of ClonePacker by comparing with Libra. Sections 5.6 describe threats to validity. Finally, we conclude this chapter in Section 5.7.

5.2 Circle Packing

Circle Packing is one of the enclosure diagrams. Figure 5.1 shows an example of Circle Packing. In the figure, Circle Packing represents three categories A, B and C. Each of the categories has some elements. For example, category A has two elements a-1 and a-2 and category C has two sub-categories, c-1 and c-2. Circle

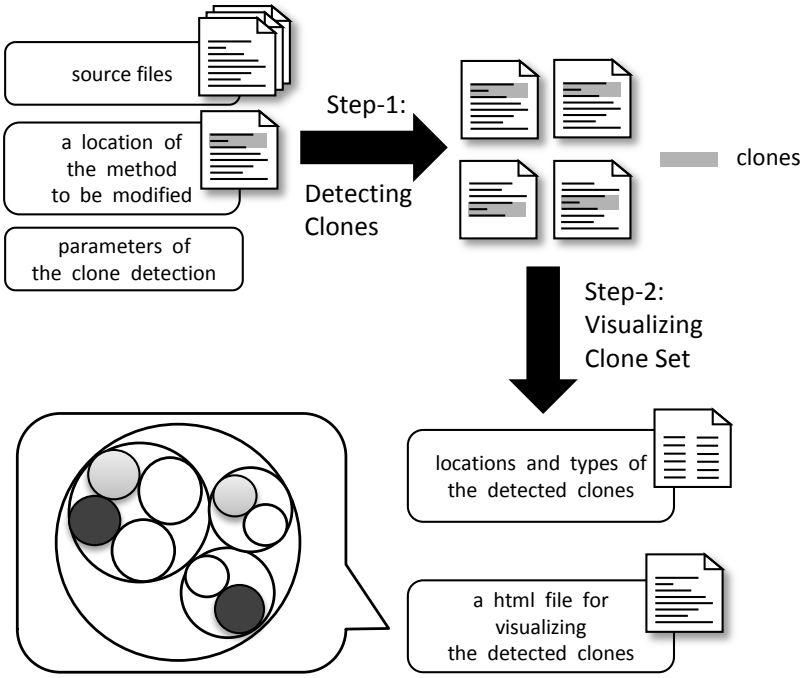


Figure 5.2: Overview of Proposed Technique

Packing is useful for representing hierarchical data structures. In Fig. 5.1, it is clear that elements a-1 and a-2 are in the same category. On the other hand, elements a-1 and b-1 are in different categories. Furthermore, both of elements c-1-1 and c-2-1 are in category C, however, they are in different sub-categories, c-1 and c-2.

We use Circle Packing for visualizing detected clones with file hierarchies. In this chapter, we assume that the innermost circles represent methods, and the circles covering the innermost circles represent files. Moreover, the circles covering file circles represent directories. For example, in Fig. 5.1, the outermost circle represents directory Root. The directory contains two files A and B, and one directory C. File A has two methods a-1 and a-2. Directory C has two files c-1 and c-2, and file c-1 has two methods c-1-1 and c-1-2.

There are two characteristics of Circle Packing [79]. One characteristic is that Circle Packing represents file hierarchies as previously mentioned. Combining clones with file hierarchies provides beneficial information for developers. For example, Hauptmann said that some industrial software systems accidentally contained the same files in different directories, and knowing such system parts helps developers reduce special cases of clones from the analysis [22]. The other char-

acteristic is that the area and color of each circle can be used to represent arbitrary information. In this study, the area is used for representing the size of the method, and the color is used for representing the type of the clones. Details are describes in sub-section 5.4.2. Knowing the sizes and types of clones is important in clone analysis. For example, Mondal said that Type-3 clones should be given a higher priority than Type-1 and Type-2 clones in clone analysis [55].

5.3 Proposed Technique

Figure 5.2 shows an overview of the proposed technique. The proposed technique consists of two steps.

Step-1: Detecting Clones

Step-2: Visualizing Clone Set

First, users prepare a set of source files that is targeted for the clone detection (in short, target source files). Second, users specify a method to be modified (in short, a target method). Then, the proposed technique detects clones of the target method from the target source files. Lastly, detected clones are visualized by using Circle Packing.

The inputs of the proposed technique are followings:

- target source files,
- a file name and a start line of a target method, and
- parameters of the clone detection (*minimum token length* and *the number of allowed gapped statements*).

The outputs are followings:

- file names, start lines, end lines and types of the detected clones, and
- a html file for visualizing the detected clones with Circle Packing.

In the rest of this section, we describe each step.

5.3.1 Step-1: Detecting Clones

The proposed technique detects clones of the target method from the target source files by considering the input *minimum token length* and *the number of allowed gapped statements*. In this step, the proposed technique uses a version of

customized our previous technique [57] to detect method clones[1]. The technique can detect all types of clones in a short time. The technique detects clones as a clone set.

5.3.2 Step-2: Visualizing Clone Set

The proposed technique visualizes the clone set obtained in Step-1. The clone set is visualized as Circle Packing. Furthermore, locations and types of the detected clones are also reported.

5.4 Tool: ClonePacker

5.4.1 Implementation

We have implemented the proposed technique as a tool, ClonePacker. ClonePacker has been developed as an Eclipse plugin. It is downloadable from our website[2]. We used JavaScript library D3[3] for visualizing the clone set. The proposed technique creates a html file representing the clone set. Then, the proposed technique visualizes the clone set by giving the html file to D3.

5.4.2 How to Use ClonePacker

Figure 5.3 shows a screenshot of ClonePacker. First, users select a target method by setting a caret position on the method. In Figure 5.3, the caret position exists at 114th line. In this case, method draw (109th - 127th lines) is selected as the target method. After the users push the button A, ClonePacker finds clones of the target method.

After ClonePacker finishes detecting clones, the users can see the detection results. In Figure 5.3, the right view B shows the detected clones with Circle Packing. The yellow circle represents the target method that the users selected. The red one is Type-1 clone, the blue one is Type-2 clone and the green one is Type-3 clone. In this case, one Type-1 clones, two Type-2 clones and one Type-3 clone were detected. The Type-1 clone and one of Type-2 clones locate in the same directory with the target method and the others locate in different directories. The size of each innermost circle represents LOC of the method. Location and type of each clone are showed in the bottom table by clicking each circle. The location of the clone is represented as a combination of its file path, its method name, its start

[1] Method clones are methods that have identical or similar methods in source code.
[2] http://sdl.ist.osaka-u.ac.jp/~h-murakm/clonepacker/
[3] http://d3js.org/

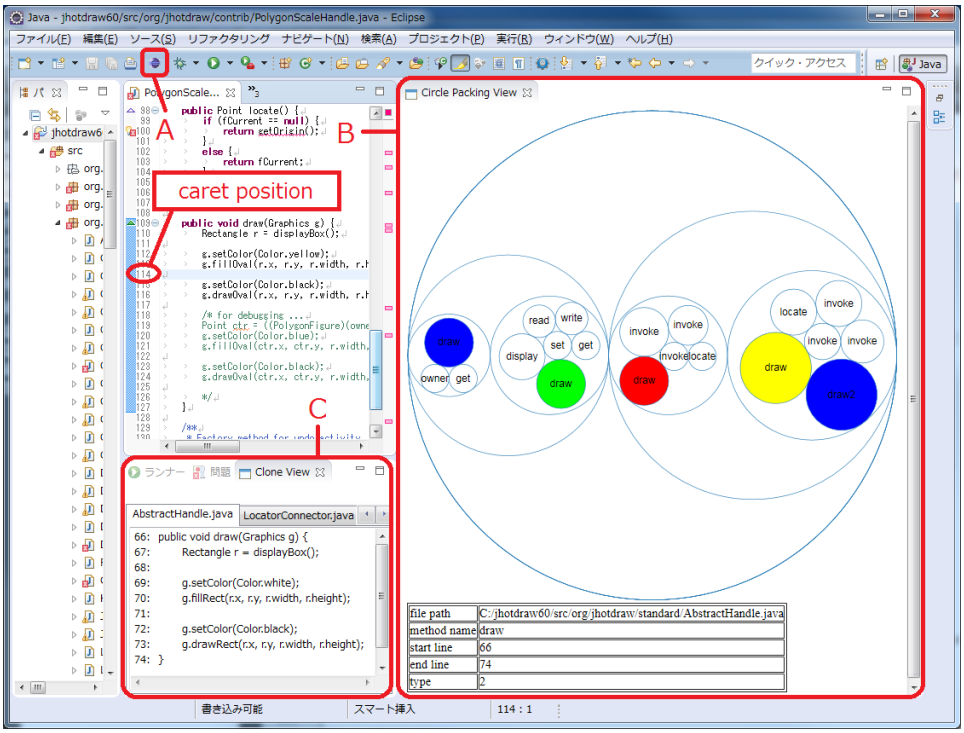


Figure 5.3: Screenshot of ClonePacker

line and its end line. The users can also browse the source code of the clones at the bottom view C.

5.4.3 Example of Supporting Scenario

Figure 5.4 shows two code fragments in JFreeChart. The 608th line of CombinedDomainXYPlot.java and the 469th line of CombinedDomainCategoryPlot.java include the same method invocations. One was modified in 04/Dec./2007 and the other was modified in 28/Mar./2008. From the commit log of 28/Mar./2008, the modification in CombinedDomainCategoryPlot.java was for bug fix. Thus, the two method invocations must have been modified simultaneously. However, the developers overlooked the modification in CombinedDomainCategoryPlot.java. In Fig. 5.4, the two code fragments are clones. By using ClonePacker in 04/Dec./2007, the developers would have understood that the two method invocations must have been modified simultaneously. ClonePacker is useful for preventing code fragments that should be modified simultaneously from being overlooked.

trunk/source/org/jfree/chart/plot/CombinedDomainXYPlot.java

```
  604: protected void setFixedRangeAxisSpaceForSubplots(AxisSpace space) {
  605:   Iterator iterator = this.subplots.iterator();
  606:   while (iterator.hasNext()) {
  607:      XYPlot plot = (XYPlot) iterator.next();
- 608:      plot.setFixedRangeAxisSpace(space);
+           plot.setFixedRangeAxisSpace(space, false);
  609:   }
  700: }
```

This modification was occurred in **04/Dec./2007**

trunk/source/org/jfree/chart/plot/CombinedDomainCategoryPlot.java

```
  465: protected void setFixedRangeAxisSpaceForSubplots(AxisSpace space) {
  466:   Iterator iterator = this.subplots.iterator();
  467:   while (iterator.hasNext()) {
  468:      CategoryPlot plot = (CategoryPlot) iterator.next();
- 469:      plot.setFixedRangeAxisSpace(space);
+           plot.setFixedRangeAxisSpace(space, false);
  470:   }
  471: }
```

This modification was occurred in **28/Mar./2008**

Figure 5.4: Modifications in JFreeChart

5.5 Experiment

5.5.1 Evaluation for Clone Analysis Time

In order to evaluate ClonePacker, we conducted an experiment with participants. The participants performed some tasks with ClonePacker and Libra. Then, we compared task completion time of ClonePacker and Libra. In this experiment, ten participants took part in the experiment. Eight participants were master's course students, and the other two participants were undergraduate students at Osaka University.

First, we divided the participants into two groups, called G_A and G_B. Since the number of the participants was ten, each group had five participants.

Table 5.1: Details of Experimental Tasks

Tasks	Target method	Locations of the target method (start line - end line)	# Type-1	# Type-2	# Type-3
Task-1	suite	test/samples/minimap/MinimapSuite.java (37 - 57)	0	2	0
Task-2	handles	figures/GroupFigure.java (67 - 74)	0	2	1
Task-3	draw	contrib/PolygonScaleHandle.java (111 - 129)	4	3	1
Task-4	store	util/SerializationStorageFormat.java (62 - 68)	0	1	0
Task-5	fillRoundRect	contrib/zoom/ScalingGraphics.java (212 - 217)	0	2	2
Task-6	handles	contrib/TextAreaFigure.java (299 - 303)	5	0	1

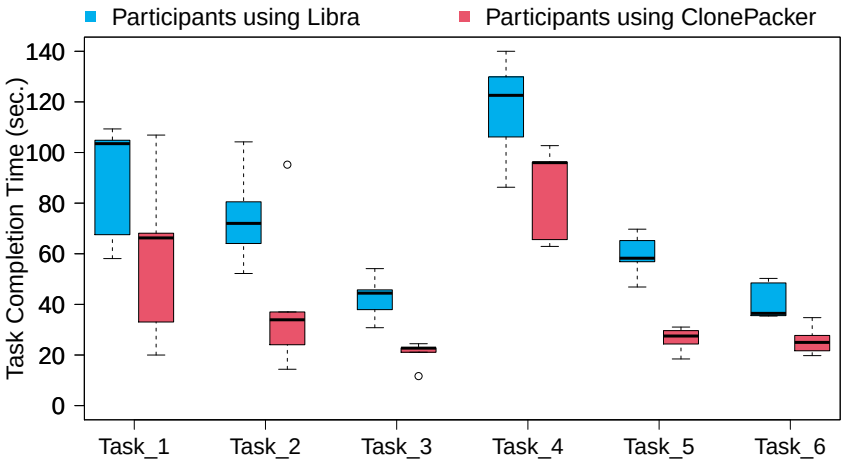


Figure 5.5: Results of Task Completion Time

Second, each group worked on the tasks. All of the tasks were very simple, *"Please report locations of all clones of the given method"*. In each task, the participants were given one target method, then they found its clones by using the tools and reported the locations of detected clones. In this experiment, we set *minimum token length* as 30 and *the number of allowed gapped statements* as 2. We made the participants measure their task completion time from the beginning to the end in each task. Table 5.1 shows details of the tasks. All of the target methods were found in JHotDraw 6.0 beta 1. For example, in Task-2, ClonePacker found three clones (two Type-2 clones and one Type-3 clone). However, Libra found only two Type-2 clones because Libra used CCFinder for detecting clones and CCFinder did not have a capability of detecting Type-3 clones.

Although ClonePacker reported types of the detected clone, the participants had to report only the locations of the detected clones. The reason is that Libra did not report types of detected clones and we would like to provide a fair comparison between ClonePacker and Libra. Furthermore, in order to achieve a fair comparison, both the groups changed the tools at the timing of finishing a half of the tasks. G_A used ClonePacker and G_B used Libra for working on Task-1, Task-2 and Task-3. Then, G_A used Libra and G_B used ClonePacker for working on Task-4, Task-5 and Task-6.

Figure 5.5 shows results of the task completion time. Its horizontal axis represents each task and the vertical axis represents task completion time. The blue box plots represent time for participants using Libra and red box plots represent time for participants using ClonePacker. For example, in Task-1, the fastest participant

using ClonePacker took about 20 seconds per clone to report locations of detected clones. From Figure 5.5, it was likely that the participants using ClonePacker reported the locations of clones faster than Libra.

In order to show that there was a significant difference between the completion time for ClonePacker and Libra, we introduced the following null and alternative hypotheses.

H_0**:** The null hypothesis is that there is no significant difference between the completion time for ClonePacker and Libra.

H_1**:** The alternative hypothesis is that there is a significant difference between the completion time for ClonePacker and Libra.

We confirmed that completion time for ClonePacker and Libra have equal variances and do not follow a normal distribution at 0.05 level of a significance by using F-test and Shapiro-Wilk test, respectively. Thus, we conducted Wilcoxon test. The p-value obtained from Wilcoxon test was 6.724e-05. Since p-value was less than 0.05, we rejected H_0 and adopted H_1. Therefore, there was a significant difference between the completion time for ClonePacker and Libra. From the result of Wilcoxon test and Figure5.5, we confirmed that the participants using ClonePacker reported the locations of clones faster than Libra.

5.5.2 Evaluation for Usability

Next, we compared ClonePacker with Libra from the perspective of their usability by using System Usability Scale [6]. The purpose of this evaluation is to investigate whether ClonePacker is easy to use or not.

System Usability Scale consists of the following 10 item questionnaire with five response options for the participants; strongly agree (5), agree (4), neutral (3), disagree (2), Strongly disagree (1).

Q1: I think that I would like to use this system frequently.

Q2: I found the system unnecessarily complex.

Q3: I thought the system was easy to use.

Q4: I think that I would need the support of a technical person to be able to use this system.

Q5: I found the various functions in this system were well integrated.

Q6: I thought there was too much inconsistency in this system.

Q7: I would imagine that most people would learn to use this system very quickly.

Q8: I found the system very cumbersome to use.

Q9: I felt very confident using the system.

Q10: I needed to learn a lot of things before I could get going with this system.

The participants answered the questionnaire for ClonePacker and Libra, respectively. Moreover, we provided a free description column in order to collect their opinions and comments. Note that the questionnaires were filled out anonymously.

Next, the answers of the participants were normalized by the following steps.

- for odd questionnaire, subtract 1 from the answer of the participants.
- for even questionnaire, subtract the answer of the participants from 5.
- By these transformations, all values become from 0 to 4 (0 is the most negative answer, and 4 is the most positive answer).
- Each transformed values was multiplied by 2.5. After this multiplication, sum of the values become from 0 to 100 (100 is the most positive score).

Table 5.2 shows that the results of the evaluation. From Table 5.2, the scores of the participants for ClonePacker is higher than Libra in most cases. Therefore, we confirmed that ClonePacker has higher usability than Libra. However, in only two cases, Libra has higher usability than ClonePacker. The questionnaires are Q2 and Q6. This imply that some participants think that ClonePacker is unnecessarily complex or there is too much inconsistency in ClonePacker. For Q2, the participant answered "ClonePacker requires care for browsing two views" in the free description column. For Q6, the free description column was blank. In the future, we will improve ClonePacker based on the opinions.

5.6 Threats to Validity

5.6.1 Configurations of Clone Detection

In this experiment, we set *minimum token length* as 30 and *the number of allowed gapped statements* as 2. In general, configurations of a clone detection strongly affect the detection results. Wang et al. proposed a technique to find suitable configurations of a clone detection automatically [84]. If we use their technique for finding suitable configurations, we may obtain different results from this experiment.

Table 5.2: Results of System Usability Scale

Participant	Tool	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Score
A	ClonePacker	3	1	4	2	4	2	4	2	4	3	**72.5**
	Libra	3	4	2	2	3	2	2	4	3	4	42.5
	difference	0	3	2	0	1	0	2	2	1	1	30.0
B	ClonePacker	4	2	4	2	4	2	4	2	4	2	**75.0**
	Libra	3	2	2	2	3	2	4	4	3	2	57.5
	difference	1	0	2	0	1	0	0	2	1	0	17.5
C	ClonePacker	4	2	4	4	4	2	5	2	4	4	**67.5**
	Libra	4	3	2	4	2	2	2	4	3	4	40.0
	difference	0	1	2	0	2	0	3	2	1	0	27.5
D	ClonePacker	4	1	4	3	3	2	4	1	3	3	**70.0**
	Libra	2	4	1	4	3	3	2	3	2	3	32.5
	difference	2	3	3	1	0	1	2	2	1	0	37.5
E	ClonePacker	3	2	4	3	3	2	4	2	3	2	**65.0**
	Libra	3	2	3	4	3	2	3	2	3	3	55.0
	difference	0	0	1	1	0	0	1	0	0	1	10.0
F	ClonePacker	3	2	4	2	4	3	4	2	4	4	**65.0**
	Libra	2	3	2	3	3	2	3	3	2	4	42.5
	difference	1	1	2	1	1	-1	1	1	2	0	22.5
G	ClonePacker	4	4	4	2	2	2	4	4	2	2	**55.0**
	Libra	1	2	4	4	1	2	4	5	1	2	40.0
	difference	3	-2	0	2	1	0	0	1	1	0	15.0
H	ClonePacker	4	1	4	2	4	1	5	2	4	2	**82.5**
	Libra	2	3	1	4	2	2	1	5	2	4	25.0
	difference	2	2	3	2	2	1	4	3	2	2	57.5
I	ClonePacker	4	1	5	2	4	1	4	2	3	3	**77.5**
	Libra	2	3	3	4	2	2	3	4	3	4	40.0
	difference	2	2	2	2	2	1	1	2	0	1	37.5
J	ClonePacker	4	1	4	1	4	2	4	1	4	4	**77.5**
	Libra	2	1	4	2	3	2	4	3	4	4	62.5
	difference	2	0	0	1	1	0	0	2	0	0	15.0

5.6.2 Target Software System

We used only one target software system in this study. If we use other software systems, the results might be different. In order to minimize this threat, we should apply ClonePacker to many other systems. Furthermore, ClonePacker visualizes many circles for large software systems including many clones. In such a case, in order to visualize all detected clones, each circle is likely to be small. Thus, the user may not be able to browse detected clones efficiently. In the future, we are going to tackle this problem.

5.6.3 Participants

Ten participants used ClonePacker and Libra for conducting the given tasks. All of the participants had experiences of Java programming more than one year. If their programming skills are differed widely, the differences would affect their completion time of the given tasks. However, we tried our best to allocate the participants by considering their programming experiences. Moreover, the participants changed the tools at the timing of finishing a half of the tasks. Hence, we considered that we were able to minimize the differences of skills in G_A and G_B.

5.6.4 Experimental Methodology

Some people may think that the experimental methodology is unfair because the proposed visualization technique is more abstract representation than the existing technique. There are many situations that a code fragments is used for retrieval key and then detection results are analyzed in clone analysis or management. Moreover, some studies designed experiments on the assumption for such situations [56, 87]. We therefore consider that the experimental methodology is reasonable. Of course, more elaborate clone analysis is needed. For example, we compare time for fixing clone-related bugs actually between use cases of clone analysis tools. In this experiment, we compared time for reporting locations of detected clones between tools because knowing locations of clones is the most fundamental task in clone analysis.

5.7 Conclusions

In this chapter, we introduced our Eclipse plugin, named ClonePacker. It helps programmers when they modify a code fragment and check its clones. ClonePacker receives a set of source files and a method that is to be modified from programmers. Then, ClonePacker detects clones of the method from the source files. Finally, ClonePacker visualizes the detection results by using Circle Packing.

We conducted an experiment with participants to compare task completion time of ClonePacker and Libra. As a result, we confirmed that programmers using ClonePacker reported the locations of clones faster than Libra, and ClonePacker has higher usability than Libra. In the future, we are going to apply ClonePacker to many systems.

Chapter 6

Conclusion

This chapter provides the conclusions of this dissertation.

6.1 Contribution

In this dissertation, we proposed fast and precise clone detection. The proposed techniques are designed for improving existing techniques with respect to the following problems.

Problem 1: Detection time is too long

Problem 2: Detection accuracies are not sufficient

Problem 1 is caused by AST-based and PDG-based detection techniques. These detection techniques transform source code into a tree or graph representation, and then they detect common sub-trees or isomorphic sub-graphs as clones. Finding common sub-trees or isomorphic sub-graphs requires much time. Therefore, the detection time of these techniques is too long.

Problem 2 has two causes. The first is the use of module-based clone detection techniques. Some existing detection techniques for gapped clones regard similar modules as clones (e.g., blocks or methods). Thus, these techniques cannot find clones that are partially duplicated in modules. The second cause is the presence of repeated instructions in the source code. For example, repeating `case` entries in a `switch` statement or repeating similar method invocations are included in the repeated instructions. The existing line-based or token-based detection techniques find many redundant clones from the repeated instructions. Hence, these techniques have low $precision$.

To resolve these problems, we proposed two detection techniques.

First, we conducted a study of **clone detection for reducing false positives**. In that study, we proposed pre-processing that folds repeated instructions in the source code. This pre-processing prevented many false positives that are detected in line-based and token-based techniques. Then, we implemented token-based detection with processing using the suffix array algorithm. The name of the implemented tool is FRISC. To evaluate the performance of FRISC, we compared FRISC with the existing clone detectors. We used Bellon's benchmark for the comparison. From the experimental results, we confirmed that (1) the folding operation avoided many false positives, (2) some clones were newly detected by the folding operation, and (3) FRISC detected more *clone references* than any other detectors in most cases.

Second, we conducted a study of **clone detection using the Smith-Waterman algorithm**. In that study, we proposed the clone detection technique that resolves the existing problems. The proposed technique did not adopt tree-based or graph-based comparisons, and regarded consecutive similar statements as clones. We developed the proposed technique as a tool, named CDSW. The detection time of CDSW is rapid, and CDSW can find clones that are partially duplicated in modules. Moreover, we improved Bellon's benchmark by adding location information of gapped lines for *clone references*. We reported an experiment that compares Bellon's *clone references* and our *clone references*. Finally, we compared accuracies between CDSW and the existing clone detectors by using Bellon' benchmark and the enhanced *clone references*. From the experimental results, we confirmed that (1) our *clone references* evaluated gapped clones more correctly than Bellon's *clone references*, (2) CDSW detected clones in a short time from a large software system, and (3) CDSW had the best *F-measure* of all the clone detectors used with Bellon's Benchmark.

We revealed that FRISC and CDSW improved the existing problems. Many clone detectors including FRISC and CDSW provide the location information of detected clones, such as file names, start lines, and end lines. However, it is sometimes difficult for developers to analyze clones with only the location information of the clones.

Next, we conducted research on clone analyses. We conducted a study of **clone visualization using circle packing**. When developers analyze detected clones, they sometimes use clone visualization tools. However, when programmers use such existing tools, they have to open a number of source files and move the scroll bar up or down to browse the detected clones. To reduce the cost of browsing the detected clones, we proposed a clone visualization technique and developed the proposed technique as a tool, called ClonePacker. By using ClonePacker, developers can analyze clones with a single view. We conducted experiments with student participants, who compared ClonePacker with the existing tool Libra from the perspectives of time to report the clone locations and usability of the

tools. From the experimental results, we confirmed that (1) the developers using ClonePacker report the locations of clones faster than the developers using Libra and (2) ClonePacker has higher usability than Libra.

6.2 Future Work

Based on the results and knowledge provided by these studies, some issues and improvements will be starting points for future work.

Further improvement of detection accuracy: In this dissertation, we proposed technique for improving accuracy of clone detection. However, we consider that there is room for improvement on detection accuracy. Improving detection accuracy is an everlasting challenge in clone-related research. One solution is combining the techniques described in Chapter 3 and Chapter 4. In Chapter 3, we proposed pre-processing for reducing uninteresting clones. Moreover, in Chapter 4, we proposed the detection technique for improving the detection time and accuracy. Thus, the pre-processing and the detection techniques can be combined.

Applying the proposed techniques to various software systems: In this dissertation, we used open source software systems for the experiments. There are many studies that clones in commercial software systems are investigated. It is said that clones in commercial software systems have different characteristics from clones in open source software systems. We should apply the proposed techniques to various software systems (e.g., industrial systems, very large systems and systems currently in development) and investigate characteristics of detected clones in them. Then, we may obtain interesting results.

Bibliography

[1] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, 1995.

[2] G. Alkhatib. The Maintenance Problem of Application Software: An Empirical Analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104, 1992.

[3] M. Asaduzzaman, C. K. Roy, and K. A. Schneider. VisCad: Flexible Code Clone Analysis Support for NiCad. In *Proceedings of the 5th International Workshop on Software Clones*, pages 77–78, 2011.

[4] B. S. Baker. Finding Clones with Dup: Analysis of an Experiment. *Transactions on Software Engineering*, 33(9):608–621, 2007.

[5] B.S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[6] A. Bangora, P. T. Kortumb, and J. T. Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.

[7] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software Complexity and Maintenance Costs. *Communications of the ACM*, 36(11):81–94, 1993.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 368–377, 1998.

[9] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *Transactions on Software Engineering*, 31(10):804–818, 2007.

[10] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *Proceedings of the 7th International Symposium on String Processing Information Retrieval*, pages 39–48, 2000.

[11] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An Empirical Study on Inconsistent Changes to Code Clones at the Release Level. *Science of Computer Programming*, 77(6):760–776, 2012.

[12] R. N. Burns and A. R. Dennis. Selecting the Appropriate Application Development Methodology. *SIGMIS Database*, 17(1):19–23, 1985.

[13] CCFinderX. <http://www.ccfinder.net/ccfinderx-j.html>.

[14] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[15] CloneDR. <http://www.semdesigns.com/Products/Clone/>.

[16] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[17] J.R. Cordy and C.K. Roy. The NiCad Clone Detector. In *Proceedings of the 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[18] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 109–118, 1999.

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

[20] N. Göde and R. Koschke. Frequency and Risks of Changes to Clones. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 311–320, 2011.

[21] N. Göde and R. Kosheke. Incremental Clone Detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.

[22] B. Hauptmann, V. Bauer, and M. Junker. Using Edge Bundle Views for Clone Visualization. In *Proceedings of the 6th International Workshop on Software Clones*, pages 86–87, 2012.

[23] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology*, 49(9–10):985–998, 2007.

[24] Y. Higo and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, 2011.

[25] Y. Higo and S. Kusumoto. Repeated Instructions Removal Preprocessing for Lightweight Code Clone Detection. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering*, pages 2–4, 2011.

[26] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On Software Maintenance Process Improvement Based on Code Clone Analysis. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, pages 185–197, 2002.

[27] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support Based on Code Clone Analysis. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 262–269, 2007.

[28] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental Code Clone Detection: A PDG-Based Approach. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 3–12, 2011.

[29] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Emprical Study on Open Source Software. In *Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pages 73–82, 2010.

[30] K. Hotta, J. Yang, Y. Higo, and S. Kusumoto. How Accurate Is Coarse-grained Clone Detection?: Comparison with Fine-grained Detectors. In *Proceedings of the 8th International Workshop on Software Clones*, pages 1–18, 2014.

[31] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the 32th International Conference on Software Engineering*, pages 1–9, 2010.

[32] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 387–391, 2012.

[33] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[34] J.H. Johnson. Substring Matching for Clone Detection Tools. In *Proceedings of the 10th International Conference on Software Maintenance*, pages 120–126, 1994.

[35] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proceedings of the 31th International Conference on Software Engineering*, pages 1–9, 2009.

[36] T. Kamiya. Classifying code clones with configuration. In *Proceedings of the 4th International Workshop on Software Clones*, pages 75–76, 2010.

[37] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Transactions on Software Engineering*, 28(7):654–670, 2002.

[38] I. Keivanloo and G. Roussel amd J. Rilling. Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning. In *Proceedings of the 6th International Workshop on Software Clones*, pages 36–42, 2012.

[39] 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*, 229(11):4247–4258, 2010.

[40] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory Comparison-Based Clone Detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011.

[41] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[42] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya. Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics. In *Proceedings of the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, pages 241–243, 2010.

[43] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001.

[44] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, 2012.

[45] R. Koschke, R. Falke, and P. Frenzel. Clone Detection using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[46] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[47] J. Krinke. Is Cloned Code more stable than Non-Cloned Code? In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manupulation*, pages 57–66, 2008.

[48] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the 13th International Conference on Software Maintenance*, pages 314–321, 1997.

[49] F. Lanubile and T. Mallardo. Finding Function Clones in Web Applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 379–386, 2003.

[50] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *Transactions on Software Engineering*, 32(3):176–192, 2006.

[51] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Program Using Distributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering*, pages 106–115, 2007.

[52] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the Relation between Changeability Decay and the Characteristics of Clones and Methods. In *Proceedings of the 23rd International Conference on Automated Software Engineering,*, pages 100–109, 2008.

[53] J. Mayland, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, pages 244–253, 1996.

[54] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy J. Krinke, and K. A. Schneider. An Empirical Study of the Impacts of Clones in Software Maintenance. In *Proceedings of the 19th International Conference on Program Comprehension*, pages 242–245, 2011.

[55] M. Mondal, C. K. Roy, and K. A. Schneider. A Comparative Study on the Bug-Proneness of Different Types of Code Clones. In *Proceedings of the 31th International Conference on Software Maintenance and Evolution*, pages 91–100, 2015.

[56] S. Morisaki, N. Yoshida, Y. Higo, S. Kusumoto, K. Inoue, K. Sasaki, K. Murakami, and K Matsui. Empirical Evaluation of Similar Defect Detection by Code Clone Search. *IEICE Transactions on Information and Systems*, 91-D(10):2466–2477, 2008 (in Japanese).

[57] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped Code Clone Detection with Lightweight Source Code Analysis. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 93–102, 2013.

[58] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. ClemanX: Incremental Clone Detection Tool for Evoluving Software. In *Proceedings of the 31st International Conference on Software Engineering*, pages 437–438, 2009.

[59] NiCad3 Clone Detector. <http://www.txl.ca/nicaddownload.html>.

[60] Detection of Software Clones. <http://www.bauhaus-stuttgart.de/clones/>.

[61] J. Ossher, H. Sajnani, and C. Lopes. File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 283–292, 2011.

[62] Circle Packing. <http://bl.ocks.org/mbostock/4063530>.

[63] L. Paulevë, H. Jëgou, and L. Amsaleg. Locality Sensitive Hashing: A Comparison of Hash Function Types and Querying Mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, 2010.

[64] A. J. Rostkowycz, V. Rajlich, and A. Marcus. A Case Study on the Long-Term Effects of Software Redocumentation. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 92–101, 2004.

[65] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 172–181, 2008.

[66] C. K. Roy and J. R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 153–162, 2008.

[67] C. K. Roy and J. R. Cordy. Near-Miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[68] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Rools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[69] R. K. Saha, C. K. Roy, and K. A. Schneider. gCad: A Near-Miss Clone Genealogy Extractor to Support Clone Evolution Analysis. In *Proceedings of the 29th International Conference on Software Maintenance*, pages 488–491, 2013.

[70] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the Evolution of Type-3 Clones: An Exploratory Study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 139–148, 2013.

[71] A. Santone. Clone Detection through Process Algebras and Java Bytecode. In *Proceedings of the 5th International Workshop on Software Clones*, pages 73–74, 2011.

[72] Scorpio. <https://github.com/YoshikiHigo/TinyPDG>.

[73] W. Shang, B. Adams, and A. E. Hassan. An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce. In *Proceedings*

of the 25th International Conference on Automated Software Engineering,, pages 275–284, 2010.

[74] Simian. <http://www.redhillconsulting.com.au/products/simian/>.

[75] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[76] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling Classical Clone Detection Tools for Ultra-Large Datasets: An Exploratory Study. In *Proceedings of the 7th International Workshop on Software Clones*, pages 16–22, 2013.

[77] J. Svajlenko and C. K. Roy. Evaluating Modern Clone Detection Tools. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, pages 321–330, 2014.

[78] J. Svajlenko and C. K. Roy. Evaluating Clone Detection Tools with BigCloneBench. In *Proceedings of the 31th International Conference on Software Maintenance and Evolution*, pages 131–140, 2015.

[79] S. Tonidandel, E. King, and J. Cortina. *Big Data at Work: The Data Science Revolution and Organizational Psychology*. Routledge, 2015.

[80] S. Uddin, C. K. Roy, and K. Schneider. SimCad : An Extensible and Faster Clone Detection Tool for Large Scale Software Systems. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 236–238, 2013.

[81] 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 *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 13–22, 2011.

[82] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code Clone Analysis Tool. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, pages 31–32, 2002.

[83] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *Proceedings of the 8th International Symposium on Software Metrics*, pages 67–76, 2002.

[84] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 9th*

joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 455–465, 2013.

[85] M. Weiser. Program Slicing. *Transactions on Software Engineering*, 10(4):352–357, 1984.

[86] Y. Sasaki and T. Yamamotoy and Y. Hayase and K. Inoue. Finding File Clones in FreeBSD Ports Collection. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 102–105, 2010.

[87] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iida. SHINOBI: A Real-time Code Clone Detection Tool for Software Maintenance. Technical Report NAIST-IS-TR2007011, Graduate School of Information Science, Nara Institute of Science and Technology, 2010.