A Token-based Code Clone Detection Tool - CCFinder and Its Empirical Evaluation

Toshihiro Kamiya[†], Shinji Kusumoto[†], and Katsuro Inoue[†‡]

{kamiya, kusumoto, inoue}@ics.es.osaka-u.ac.jp

Abstract

A code clone is a code portion in source files that is identical or similar to another. Since code clones generally reduce maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new clone detection technique, which consists of transformation of input source text and token-by-token comparison. Based on the proposed code clone detection technique, we developed a tool named CCFinder, which extracts code clones in C/C++ or Java source files. As well metrics for code clones were developed. In order to evaluate the usefulness of the tool and metrics, we conducted several experiments. As the results, the tool found several subsystems in two operating systems, namely FreeBSD and Linux, that could be traced to the same original. As well, the proposed metrics found interesting clones in a Java library, JDK.

---

†Graduate School of Engineering Science, Osaka University

‡Graduate School of Information Science, Nara Institute of Science and Technology

Contact to: Toshihiro Kamiya

c/o Katuro Inoue

Division of Software Science

Graduate School of Engineering Science, Osaka University,

Machikaneyama-cho 1-3, Toyonaka-city, 860-8531, Japan

Phone: +81-6-6850-6571, Fax: +81-6-6850-6574

e-mail: kamiya@ics.es.osaka-u.ac.jp

Keywords

Code clone, Duplicated code, CASE tool, Metrics, Maintenance

1 Introduction

A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of various reasons such as reusing code by 'cut-and-paste' or intentionally repeating a code portion for performance enhancement[3]. Clones make the source files very hard to modify consistently. For example, assume that a software system has several clone subsystems created by duplication with slight modification. When a fault is found in one subsystem, the engineer has to carefully modify all other subsystems. For a large and complex system, there are many engineers who take care of each subsystem, and modification becomes very difficult. Various clone detection tools have been proposed and implemented [1][3][7][11][14][16], and a number of algorithms for finding clones have been used for them, such as line-by-line matching for an abstracted source program [1], and similarity detection for metrics values of function bodies [16].

We were interested in applying a clone detection technique to a huge software system for a division of government, which consists of millions lines of code in a few thousand modules written in several programming languages, which was developed more than 20 years ago and has been maintained continually by a large number of engineers [22]. It is known that there are many clones inside the system; however the documentation does not provide information regarding the clones. These clones heavily reduce maintainability of the system; thus an effective clone detection tool is expected.

Based on such initial motivation of clone detection, we have devised a clone detection algorithm and implemented a tool named CCFinder (Code-clone finder). The underlying concepts for designing the tool were as follows.

- The tool should have industrial-size strength, and be applicable to million-line size system within affordable computation time.
- The language dependent part of the tool should be limited to small parts, and the tool has to be easily adaptable to many other languages.
- The tool should detect clones of practical interest, not only syntactically same portions, but also similar portions which are considered to be actual clones have to be effectively extracted.

We have used a simplified *suffix-tree*[10] to find clones practically and effectively. Various optimization techniques were also built into the tool. The tool was initially developed for C and C++, and then successfully extended to Java by two person days. The tool transforms a source code with transformation rules so that portions of interest (but syntactically not exactly identical structures) can be detected and uninteresting portions (even when they structurally similar) are not detected. The uninteresting clone portions do not contribute to reduction of total size of code since they are hard to be merged into single portions. It performs an abstraction of a token sequence called parameter-replacement before executing the token-by-token matching algorithm. This parameter-replacement is a pre-process of *parameterized-match*[1], and is very effective for clones with name substitution. Also, token-by-token matching algorithms are able to find clones with modified line structures, which cannot be detected by line-by-line algorithm. Token-by-token matching is much more expensive than line-by-line matching in computing complexity. However, we propose several optimization techniques especially for the token-by-token matching algorithm, which enables the algorithm practically useful for large software.

The clone metrics presented in this paper refer to an equivalence class of clones, and measure where and how frequently clones appear. The metrics enable to estimate how many source lines are reduced by removing the clone codes and to evaluate how widely the clones are spread over the system.

The application of our tool is also a novel contribution of this paper. We have applied CCFinder on million-lines codes from JDK, Qt, Linux, and FreeBSD, to evaluate its effectiveness quantitatively and qualitatively. The similarity of Linux and FreeBSD, as well as nature of JDK, has also been explored. The tool and the metrics have detected clones that are small in size by themselves but many lines can be removed by rewriting them using a shared routine.

In Section 2, we introduce the clone-detecting tool *dup*, which provides a base algorithm of our clone-detecting technique, after which we describe our clone-detecting approach. Section 3 defines clones and explains our clone-detecting algorithm. In Section 4, the metrics of clones are defined, and the clone-detecting approach and the metrics are evaluated empirically with several software systems of industrial size. Section 5 surveys and discusses related works. Finally, Section 6 concludes the paper and presents suggestions for future work.

2 Preliminary

Baker developed tools, named *edup* and *pdup*, which extract clones from source files[1]. The input is source files and the output is clones found in the source files. For these tools, a clone is defined as a pair of subsequent lines, that is, if a sequence of lines is identical or similar to another sequence of lines, this pair is extracted as a clone. Edup determines that two lines are equivalent when the pair has exactly the same characters in the same order. Pdup employs a comparison algorithm called parameterized match and judges that two lines are equal even if the variables and the functions are renamed.

Our approach presented in this paper concerns the following issues in clone detection.

- **Identification of structures**

Our pilot experiment has revealed that certain types of clones seem difficult to be rewritten as a shared code even if they are found as clones. Examples are a code portion that begins at the middle of a function definition and ends at the middle of another function definition, and a code portion that is a part of a table initialization code. For effective clone analysis, our clone detection technique automatically identifies and separates each function definition and each table definition code. For comparison, in [1], table initialization values have to be removed by hand, whereas in [16], only an entire function definition can become a candidate for clone.

- **Regularization of identifiers**

Recent programming languages such as C++ and Java provide *name space* and/or *generic type* [4]. As a result, identifiers often appear with attributive identifiers of name space and/or template arguments. In order to treat each complex name as an equivalent simple name, the clone detecting process has a subprocess to transform complex names into simple form. If source files are represented as a string of tokens, structures in source files (such as sentences or function definitions) are represented as substrings of tokens, and they can be compared token-by-token to identify clones. Identifying structures and transforming names require knowledge of syntax rules of the programming languages. Therefore, the implementation of the clone detecting technique depends on the input. The detail of clone detecting process is described in Section 3.2.

- **Ranking clones by importance**

Large software systems often include many clones, so a clone analysis method must distinguish important clones from many ‘uninteresting’ clones. The metrics presented in Section 4.2 enable to identify such important clones: clones that enable large code reduction by their removal, or clones that have so widely spread in the system that are difficult to find by hand and to maintain. A certain metric value is used to estimate how many lines of source files are reduced by making a shared routine of each clone, and another is used to evaluate how each clone is spread over a software system.

3 Proposed clone-code detection technique

3.1 Definition of clone and related terms

A **clone-relation** is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone-relation holds between two code portions if (and only if) they are the same sequences[1]. For a given clone-relation, a pair of code portions is called **clone-pair** if the clone-relation holds between the portions. An equivalence class of clone-relation is called **clone-class**. That is, a clone-class is a maximal set of code-portions in which a clone-relation holds between any pair of code-portions.

For example, suppose a file has the following 12 tokens:

a x y z b x y z c x y d

We get the following three clone-classes:

C1) $a\ \underline{x\ y\ z}\ b\ \underline{x\ y\ z}\ c\ x\ y\ d$

C2) $a\ \underline{x\ y}\ z\ b\ \underline{x\ y}\ z\ c\ \underline{x\ y}\ d$

C3) $a\ x\ \underline{y\ z}\ b\ x\ \underline{y\ z}\ c\ x\ y\ d$

Note that sub-portions of code portions in each clone-class also make clone-classes (e.g. Each of C3 is a sub-portion of C1). In this paper, however we are interested only in maximal portions of clone-classes so only the latter are discussed.

[1] Sequences are sometimes original character strings, strings without white spaces, sequences of token type, and transformed token sequences. We will discusses how we deal with such sequences.

In studiers [3][7][11][12][14][16], the clone analyses have been based on clone-pairs. On the other hand, the clone analysis in [1] has also used clone-classes. The analysis by clone-classes and the metrics are described in Section 4.2.

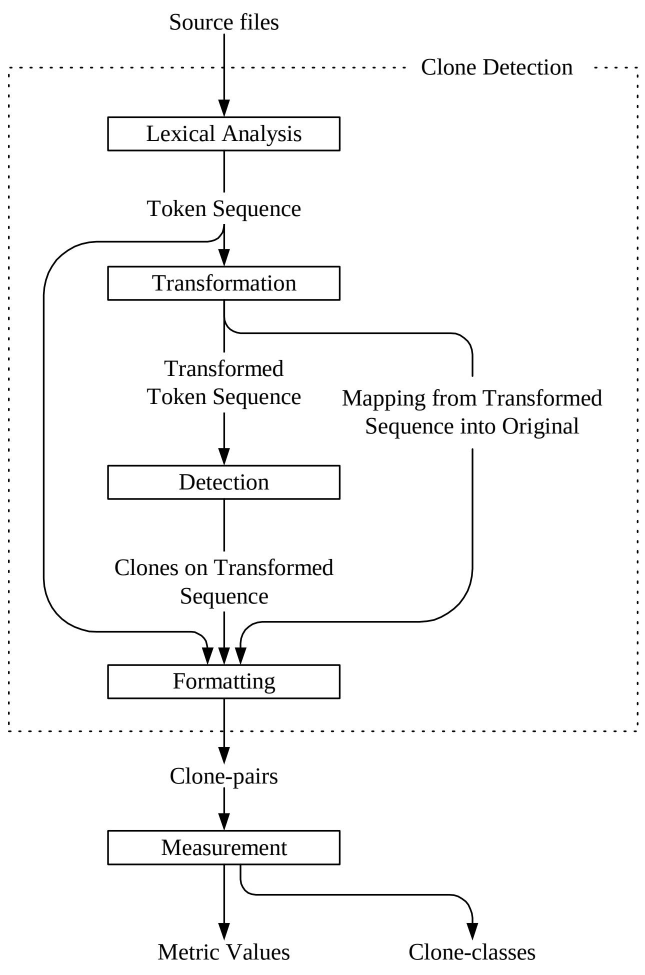


Figure 1. Clone detecting process

Table 1. Transformation rules for C++

#	Rule	Example and Purpose/Effect
RC1	**(Name '::')+ Name2 → Name2** Here, the operator +, a postfix operator of regular expression, means repeat of one or more times.	`std::ios_base::hex` is transformed into `hex`. In C++ source files, a name may belong to a name space or a class and can be spelled in full or in shorter form. The transformation is to neglect the attribution so that they are considered equivalent in clone detection.
RC2	**Name '<' ParameterList '>' → Name** Here, ParameterList is a sequence of Name, Number, String, Operators, ',' and Expression. Expression is a sequence of tokens which starts with '(' and ends with the corresponding ')' and does not include ';'.	`sort<int>` is transformed into `sort`. Template arguments may be omitted because of a type estimation by the compiler or because of the scope of the template. The transformation copes with the case.
RC3	**'=' '{' InitalizationList, '}'** **→ '=' '{' UniqueIdentifier '}'** Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'. UniqueIdentifier is a unique token, which never appears inanother place of a token sequence.	A pilot experiment showed that some tables (such as character code, color code, and wave table) include a continuation of a value and regular repeats of some values. The rule eliminates such large table initialization code.
RC4	Insert UniqueIdentifier at each end of the top-level definitions and declaration.	This rule prevents extraction of clone-pairs of the code portions that begin at the middle of a function definition and end at the middle of another function definition.

3.2 Clone-detecting process

Clone detecting is a process in which the input is source files and the output is clone-pairs. The entire process of our token-based clone detecting technique is shown in Figure 1. The process consists of four steps:

(1) Lexical analysis

Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single

Table 2. Transformation rules for Java

#	Rule	Example and Purpose/Effect
RJ1	**(PackageName '.')+ ClassName** **→ ClassName** Here, PackageName is a word that begins with a small letter and ClassName is a capitalized word.	`java.lang.Math.PI` is transformed to `Math.PI`. In Java source files, a class is referred to with either the full package name or a shorter name by using import sentences. The transformation is to neglect the attribution so that they are considered equivalent in clone detection.
RJ2	**NDotOrNew NClassName '('→ NDotOrNew CalleeIdentifier '.' NClassName '('** Here, NDotOrNew is a token except '.' or 'new'. NClassName is an uncapitalized word. CalleeIdentifier is a token for an omitted callee.	By language specification a method is either an instance method or a class method. Therefore, if an instance calls a method without a callee instance or class then the omitted callee is the instance itself or a class of it.
RJ3	**'=' '{' InitalizationList, '}'** **→ '=' '{' UniqueIdentifier '}'** **']' '{' InitalizationList, '}'** **→ ']' '{' UniqueIdentifier '}'** Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'. UniqueIdentifier is a unique token, which never appears at another place of a token sequence.	These rules are an expansion of rule RC3. The second rule is applied where an array is created with initialization by a new expression. For example, `return new int[] { 1, 2, 3 };`.
RJ4	Insert UniqueIdentifier at each end of the top-level definitions and declaration.	This rule prevents extracting clone-pairs of the code portions that begin at the middle of a class definition and end at the middle of another class definition.

token sequence, so that finding clones in multiple files is performed in the same way as single file analysis. At this step, the white spaces between tokens are removed from the token sequence, but the spaces are sent to the formatting step to reconstruct the original source files.

(2) Transformation

The token sequence is transformed by subprocesses (2-1) and (2-2) described below. At the same time, the mapping information from the transformed token sequence into the original token sequences is stored for the later formatting step.

(2-1) Transformation by the transformation rules

The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules. Table 1 and Table 2 show the transformation rules aiming at regularization of identifiers (RC1, RC2, RJ1, and RJ2) and identification of structures (RC3, RC4, RJ3, and RJ4).

(2-2) Parameter replacement

After step 2-1 each identifier related to types, variables, and constants is replaced with a special token (this replacement is a preprocess of the 'parameterized match' proposed in [1]). This replacement makes code-portions in which variables are renamed to be equivalent token sequences.

(3) Detection

From all the substrings on the transformed token sequence, equivalent pairs are detected as clone-pairs. Each clone-pair is represented as a quadruplet (*cp*, *cl*, *op*, *ol*), where *cp*

```
void print_lines(const set<string>& s) {
    int c = 0;
    set<string>::const_iterator i
        = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
void print_table(const map<string, string>& m) {
    int c = 0;
    map<string, string>::const_iterator i
        = m.begin();
    for (; i != m.end(); ++i) {
        cout << c << ", "
            << i->first << " "
            << i->second << endl;
        ++c;
    }
}
```

Figure 2. Sample code

and op are, respectively, the position of the first and second portion, and cl and ol are their respective lengths.

(4) Formatting

Each location of clone-pair is converted into line numbers on the original source files.

Figure 2 shows an example input of a C++ source to explain the clone-detecting process. The numbers at the left of the figure are line numbers. The input is divided into tokens. The token sequence transformed by the transformation rules is shown in Figure 3. Lines 1, 3, 11, and 13 become shorter. After this step the token sequence is transformed by parameter-replacement, again. The same code after parameter-replacement is shown in Figure 4. Identifiers are replaced with a token *$p* in the sample. At last, clone-pairs, i.e. equivalent substrings in the token sequence, are identified. Let t_i denote the i-th token ($1 <= i <= 114$) in the token sequence in Figure 4, and let us make a matrix $\{ d_{xy} \}$, here $d_{xy} = 1$ if t_x is equal to t_y, 0 otherwise. The part of the matrix is shown in Figure 5. In this figure, we place '*' for $d_{xy} = 1$ when $x > y$. Since it always holds that $d_{xy} = d_{yx}$ (symmetric) and $d_{xx} = 1$, we

```
 1   void print_lines ( const set & s ) {
 2   int c = 0 ;
 3   const_iterator i
 4   = s . begin ( ) ;
 5   for ( ; i != s . end ( ) ; ++ i ) {
 6   cout << c << ", "
 7   << * i << endl ;
 8   ++ c ;
 9   }
10   }
11   void print_table ( const map & m ) {
12   int c = 0 ;
13   const_iterator i
14   = m . begin ( ) ;
15   for ( ; i != m . end ( ) ; ++ i ) {
16   cout << c << ", "
17   << i -> first << " "
18   << i -> second << endl ;
19   ++ c ;
20   }
21   }
```

Figure 3. The transformed code by the transformation rules

place nothing for $d_{xy} = 1$ when $x <= y$. A clone-pair is found as a line segment of '*' that is parallel to the main diagonal of the matrix. The code portions from line 1 to 7 and from line 11 to 17[2] make a clone-pair. The code portions from line 8 to 10 and from 19 to 21 make another clone-pair. The lines 9, 10, 20, and 21 make a clone-class, but they are very short and trivial and should be filtered out by assigning a minimum length for clone.

Here, a clone-relation is specified with the transformation rules and the parameter-replacement described above. Other clone-relations are derived with a subset of the transformation rules and neglection of the parameter-replacement. In the experiments described in Section 4, a clone-relation with all the transformation rules is compared to a clone-relation with a subset of the transformation rules.

```
 1   $p $p ( $p $p & $p ) {
 2   $p $p = $p ;
 3   $p $p
 4   = $p . $p ( ) ;
 5   for ( ; $p != $p . $p ( ) ; ++ $p ) {
 6   $p << $p << $p
 7   << * $p << $p ;
 8   ++ $p ;
 9   }
10   }
11   $p $p ($p $p & $p ) {
12   $p $p = $p ;
13   $p $p
14   = $p . $p ( ) ;
15   for ( ; $p != $p . $p ( ) ; ++ $p ) {
16   $p << $p << $p
17   << $p -> $p << $p
18   << $p -> $p << $p ;
19   ++ $p ;
20   }
21   }
```

Figure 4. The code after parameter-replacement

[2] More strictly, "…from line 11 to the first token in line 17 and from line 1 to the first token in line 7 make …". The tool reports the locations of clones by line number.

3.3 The implementation techniques of tool CCFinder

Tool CCFinder was implemented in C++ and runs under Windows 95/NT 4.0 or later. CCFinder extracts clone-pairs from C, C++ and Java source files. The tool receives the paths of source files from the command-line (or text files in which the paths are listed), and writes the locations of the extracted clone-pairs to the standard output. Figure 6 shows an example of the output for the sample code. In this case, the user specified for CCFinder to extract clone-pairs that have three or more lines, and the tool reports two clone-pairs (shown at line 14 and 15). The option `-b` at line 3 shows that the user specifies for the tool to extract

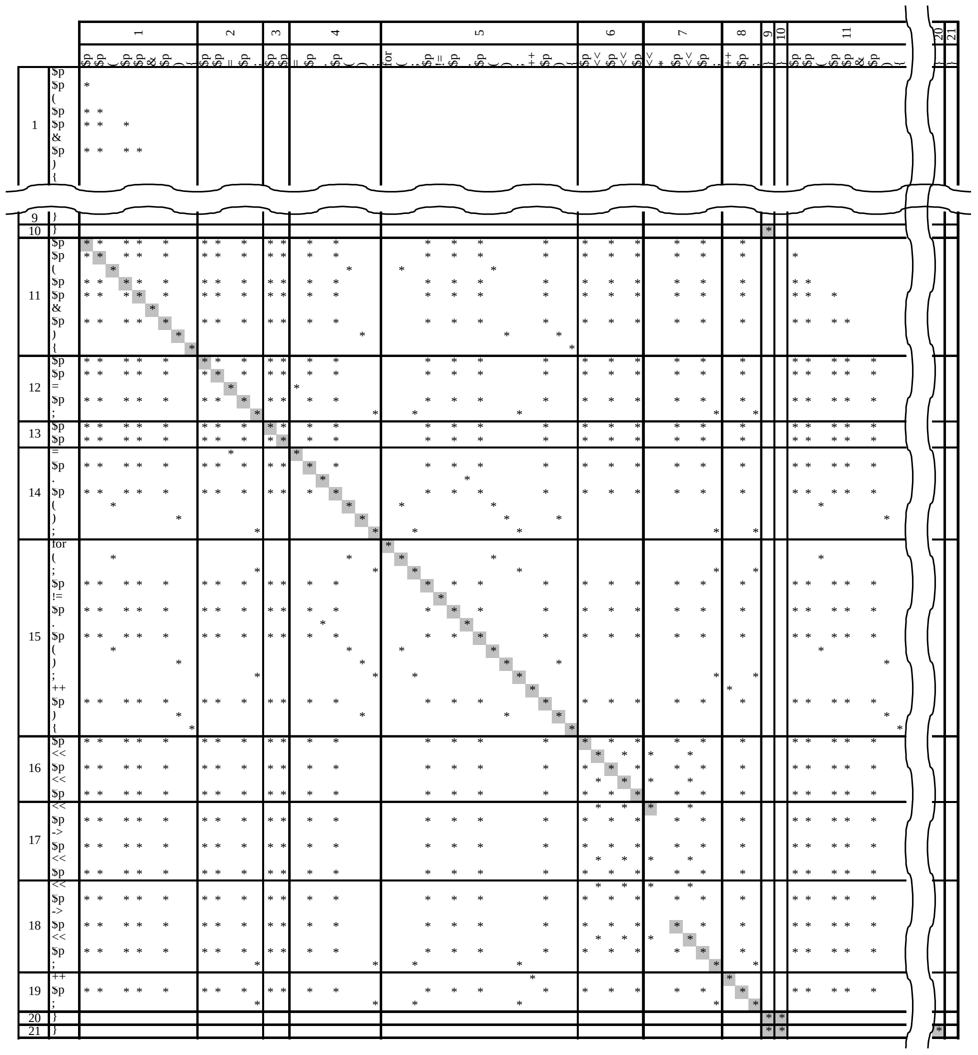


Figure 5. Matrix to show the scatter plot token-by-token

clone-pairs that have at least 3 LOC(lines of code). The examples between LOC and clone-pairs are shown in experiments in Section 4

The straightforward clone-detecting algorithm for *n* tokens with matrix requires the time complexity of $O(n^2)$. A data structure called suffix-tree is devised to detect clone-pairs and it requires $O(n)$ time complexity[1][10]. CCFinder employs a relaxed algorithm of $O(n \log n)$ time using a suffix-tree, which is not only easily implemented but also practically efficient. In Section 4, we will show that our tool can analyze millions of lines in moderate time.

The optimizations employed by CCFinder for large source files are the following:

- **Filtering by header tokens**

We would like to extract the code portions that make real sense as a clone-pair. As a simple filtering for this purpose, the clone-detection algorithm distinguishes "header" tokens. A header token is defined as the token that can be the first token of code portions of code-pairs. For example, on detecting clone-pairs in C/C++ source files, tokens, “#”, “`{`”, and “`(`” are header tokens by themselves. Also, the successors of “`:`”, “`;` ”, “`)`”, “`}`”, and ends-of-line of a preprocessor directive become header tokens. This filtering reduced the number of tokens

```
#version: ccfinder 2.1
#option -s: l
#option -b: 3
#option -k: +
#option -rC: ab-dfikmnpst
#option -rJ: ab-cdfikmnprs
#option -c: wfg
#begin{file description}
0.0     22      sample.cpp
#end{file description}
#begin{syntax error}
#end{syntax error}
#begin{clone}
0.0:1-7, 0.0:11-17, 7
0.0:8-10, 0.0:19-21, 3
#end{clone}
```

Figure 6. The output of CCFinder for the sample code

inserted into suffix-tree by factor 3 in either C/C++ or Java source file, in the experiments described in Section 4.

- **Repeated code skipping**

Repetition of a short code portion tends to generate many clone-pairs, but such clone-pairs are reconstructed by information about which code portion is repeated and where the repetition occurs. For example, consider the following code.

```
    ...
 a1: case '0':
 a2:   value = 0;
 a3:   break;
 a4: case '1:
 a5:   value = 1;
 a6:   break;
    ...
a46: case 'f':
a47:   value = 15;
a48:   break;
    ...
```

From this code section, only one clone-pair, (a1-a45, a4-a48), is extracted as a maximal clone-pair. Now consider that the following code section is also included in the target source files;

```
b1: case 'a':
b2:   flag = 2;
b3:   break;
```

In this case 17 code portions make a clone-class, { a1-a3, a4-a6, ..., a46-a48, b1-b3 }, in which each pair of the code portions makes a clone-pair, thus the number of maximal clone-pairs are 136 = ${}_{17}C_2$ = 17 × (17 - 1) / 2, in total. To avoid this explosion of clone-pairs, a heuristic approach is introduced. Upon building a suffix-tree, if a repetition of a1-a3 is found at a4, the succeeding repetition section a4-a48 is intentionally not inserted in the tree, so that the 136 clone-pairs are not being reported. However, the two clone-pairs, (a1-a45, a4-a48) and (a1-a3, b1-b3), are still extracted, which offers enough information to reconstruct the 136 clone-pairs.

- **Integer token**

A token is represented by a serial number, not as a string or a hash-value. This optimization is enabled by parameter-replacement, which causes a token sequence to consist of only limited kinds of tokens. Otherwise, a set of tokens is infinite in general, thus the tool should use string or hash-value as a representation of a token, which would cost higher time and space in clone detection.

- **Division of large source**

If the total size of source files is too large to build a single suffix-tree on primary store, the tool prepares a 'divide and conquer' approach. The input source files are divided into disjoint subsets. For each pair of the subsets, a sub suffix-tree is built to extract clone-pairs. The total collection of clone-pairs is the final output. Let *m* be the number of subsets of source files, and then number of pairs of the subsets (and thus sub suffix-trees) is ${}_{m}C_2$, therefore the time complexity becomes $O(m^2)$. As a practical example, in the experiment in Section 4.5, source files had about 3 million LOC and were divided into 4 subsets, and this has not caused a serious fall in performance.

By the definition of clone-pair and clone-class, for any clone-pair, the substrings on its code portions become a clone-pair. CCFinder does not report such 'substring' clone-pairs.

4 Experiment

The purpose of the experiment was to evaluate our token-based clone-detecting technique and the metrics. The target source files have ‘industrial’ size and are widely available. The person who performed the analysis did not have preliminary knowledge about the source files; consequently the following results are obtained purely by the analysis with the tool and metrics. In all the following experiments, tool CCFinder was executed on a PC with Pentium III 650MHz and 1GB RAM.

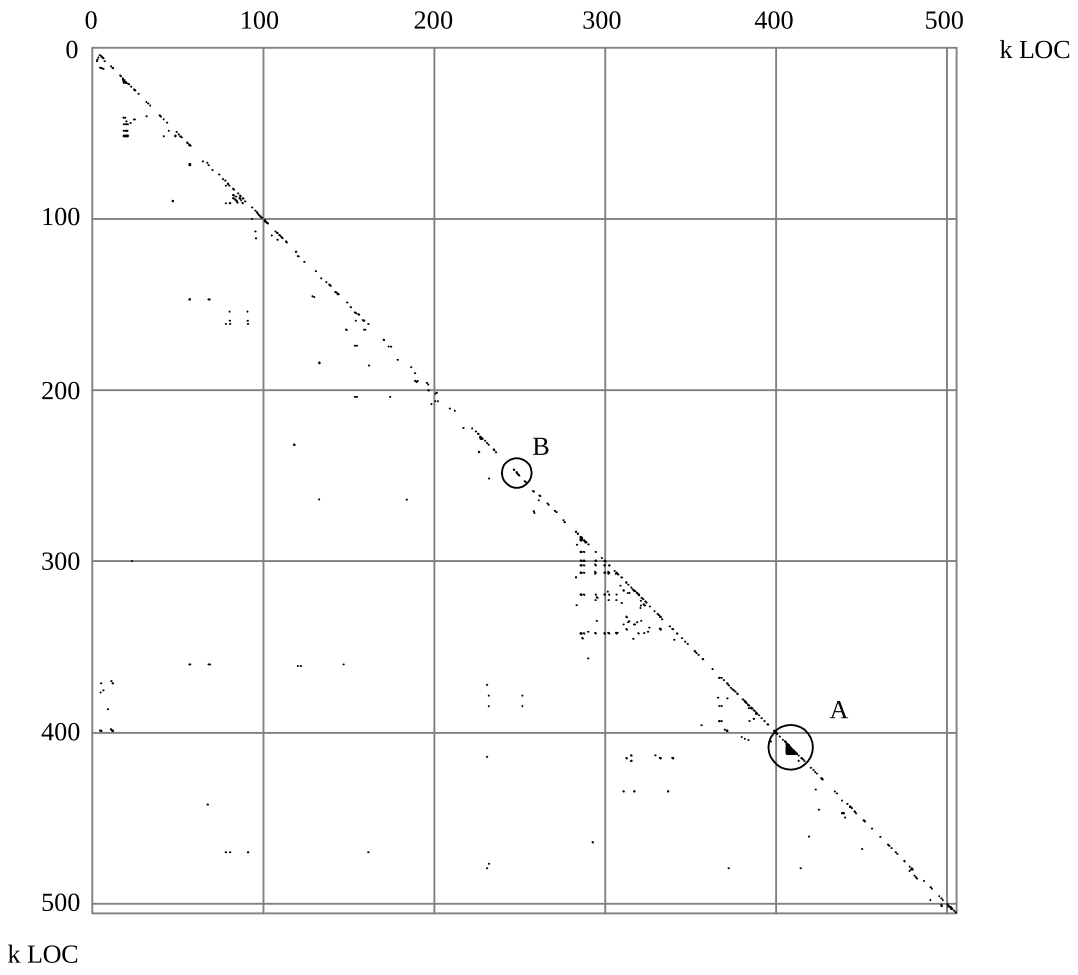


Figure 7. Scatter plot of clones over 20 lines in JDK

4.1 Clones in a Java library, JDK

JDK 1.2.2 [23] is a commonly used Java library and the source files are publicly available. Tool CCFinder has been applied to all source files of JDK excluding examples and demo programs, which are about 500k lines in total, in 1648 files. It takes about 3 minutes for execution on the PC. Figure 7 shows a scatter plot of the clone-pairs having at least 20 lines of code (LOC). Both the vertical and horizontal axes represent lines of source files. The files are sorted in alphabetical order of the file paths, so files in the same directory are also located near on the axis. A clone-pair is shown as a diagonal line segment. Only lines below the main diagonal are plotted as mentioned in Section 3.2. In Figure 7, each line segment looks like a dot because each clone-pair is small (several decades lines) in comparison to the scale of the axis. Most line segments are located near the main diagonal line, and this means that most of the clones occur within a file or among source files at the near directories.

Crowded clones marked *A* in the graph correspond to 29 files of `src/javax/swing/`

```
 */
public class MultiButtonUI extends ButtonUI {


    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiButtonUI();
        return MultiLookAndFeel.createUIs(mui,
                                          ((MultiButtonUI) mui).uis,
                                          a);
    }
```

(a) MultiButtonUI.java

```
 */
public class MultiColorChooserUI extends ColorChooserUI {


    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiColorChooserUI();
        return MultiLookAndFeel.createUIs(mui,
                                          ((MultiColorChooserUI) mui).uis,
                                          a);
    }
```

(b) MultiColorChooserUI.java

This two files are identical except three identifiers shown in bold style.

Figure 8. A pair of similar source files found in JDK

`plaf/ multi/ *.java`. These files are very similar to each other and some of them contain an identical class definition except for their different parent classes.

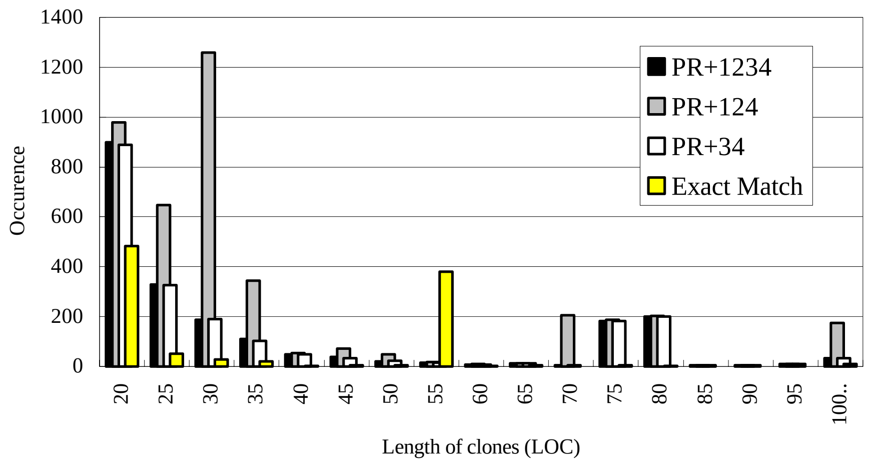
Figure 8 shows parts of the two files, as examples, `MultiButtonUI.java` and `MultiColorChooserUI.java`, and the differences are only in lines 32, 161, and 163. According to the comments of the source files, a code generator named `AutoMulti` creates the files. To modify these files, the developer should obtain the tool (the tool is not included in JDK), edit, and apply it correctly. If the developer does not use the tool, he/she has to update all the files carefully by hand. As the example shows, the modification of clones needs extra work. In this case, these clones are easily rewritten with a shared code if the programming language would support *generic type* [4].

The longest clone (349 lines) is found within `src/ java/ util/ Arrays.java` (marked *B* in Figure 7). Methods named “`sort`” have 18 variations for signatures (number and types of arguments), and they use identical algorithm/routine for sorting.

4.1 Evaluation of transformation rules for JDK

In Section 3.2, we also proposed the transformation rules for Java. To evaluate effectiveness of the transformation rules, we have applied CCFinder with some of their transformation rules disabled. Figure 9 shows the histogram of detected clone-pairs when some of rules are applied. PR+1234 means that the parameter-replacement and all rules (RJ1, RJ2, RJ3, and RJ4) are applied (i.e. original CCFinder). Exact Match means that no parameter-replacement or no transformation is applied. This figure shows that the longer the clone length is, the smaller its occurrence becomes. A noticeable peak around 80 LOC is a set of clone-pairs found in files generated by `AutoMulti`, which cannot be detected by Exact Match by the reason mentioned above. In this experiment, the clone-pairs found by PR+1234 are much fewer than with PR+124. This means that rule RJ3 removes many table initialization codes.

The case PR+1234 extracted 2111 clone-pairs and PR+34 extracted 2093 clone-pairs. There are several clone-pairs that can be detected by introducing RJ1 and RJ2. Figure 10 shows one such code portion. The lower code portion has a method call with a class name (`Utility.arranRegiionMatches`), while the upper code portion has a call without



"PR+1234" means that parameter-replacement and the transformation rule RJ1, RJ2, RJ3, and RJ4 are applied.

Figure 9. Occurences against length of clone-pairs in JDK

```
if (hashes[i] == hashes[j] &&
        arrayRegionMatches(values, iBlockStart,
        values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

```
if (hashes[i] == hashes[j] &&
        Utility.arrayRegionMatches(values, iBlockStart,
        values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

Figure 10. Part of a clone-pair captured by rule RJ2

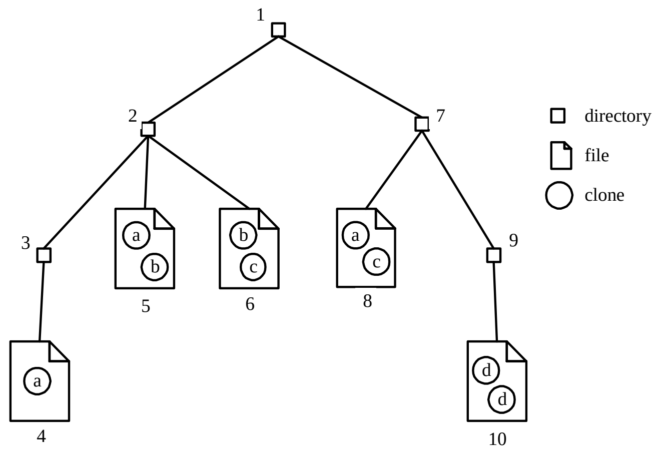
class name (`arrayRegionMatches`). In the case of Exact Match, only a small number of clone-pairs are found. The “exact” clone-pairs are obvious candidates to be rewritten as a shared code. However, our transformation and parameter replacement approach finds more subtle clone-pairs so that the chances to rewrite and reorganize overall structures of software systems become higher.

4.2 Analysis using clone metrics

We define several metrics for clone-classes in order to find important clone-classes, which enable us to perform large code reduction. Also, we use metrics to find clone-classes that are widely spread over a system.

Radius of clone-class; *RAD*(*C*)

For a given clone-class *C*, let *F* is a set of files which include each code portion of *C*. Define *RAD*(*C*) as the maximum length of path from each file *F* to the lowest common ancestor directory of all files in *F*. If all code portions of *C* are included in one file, *RAD*(*C*) = 0. In Figure 11, RAD({4:a, 5:a, 8:a }) = 3 since their lowest common ancestor is 1 and the maximum path length from directory 1 to each file is 3 for file 4. Note that RAD({ 10:d, 10:d }) = 0 since the lowest common ancestor is file 10 itself.



Clone-class *C*	*RAD*(*C*)
{ 4:a, 5:a, 8:a }	3
{ 5:b, 6:b }	1
{ 6:c, 8:c }	2
{ 10:d, 10:d }	0

Figure 11. Radius and population of clone-class

If a clone-class has a large RAD, the code portions widely spread over a software system, and it would become difficult to find those clones and maintain their consistency correctly, since such different subsystems are likely to be maintained by different engineers.

Length; *LEN*(*C*), *LEN*(*p*)

$LEN(p)$ is the number of lines of a code portion p. $LEN(C)$ for clone-class C is the maximum $LEN(p)$ for each p in C.

Population of clone-class; *POP*(*C*)

$POP(C)$ is the number of elements of a given clone-class C.

A clone class with a large POP means that similar code portions appear in many places.

Deflation by clone-class; *DFL*(*C*)

Combination of LEN and POP gives an estimation of how many lines would be removed from source files by rewriting each clone-class as a shared code. Suppose that all code-portions of a clone-class C are replaced with caller statements of a new identical routine (function, method, template function, or so) and that this caller statement is one line. In this case $LEN(C) \times POP(C)$ lines of code are occupied in the original source files. In the newly restructured source files, they occupy $POP(C)$ lines for caller statements and $LEN(T)$ for a callee routine. Now let us define a metric DFL[3] as a rough estimator of reduced source lines:

$$\begin{aligned} DFL(C) &= (\text{old LOC related to } C) - (\text{new LOC related to } C) \\ &= LEN(C) \times POP(C) - (POP(C) + LEN(C)) \\ &= (LEN(C) - 1) \times (POP(C) - 1) - 1, \end{aligned}$$

[3] A similar metric is used in [1], which estimates how many lines are removed in total by rewriting all clone-classes.

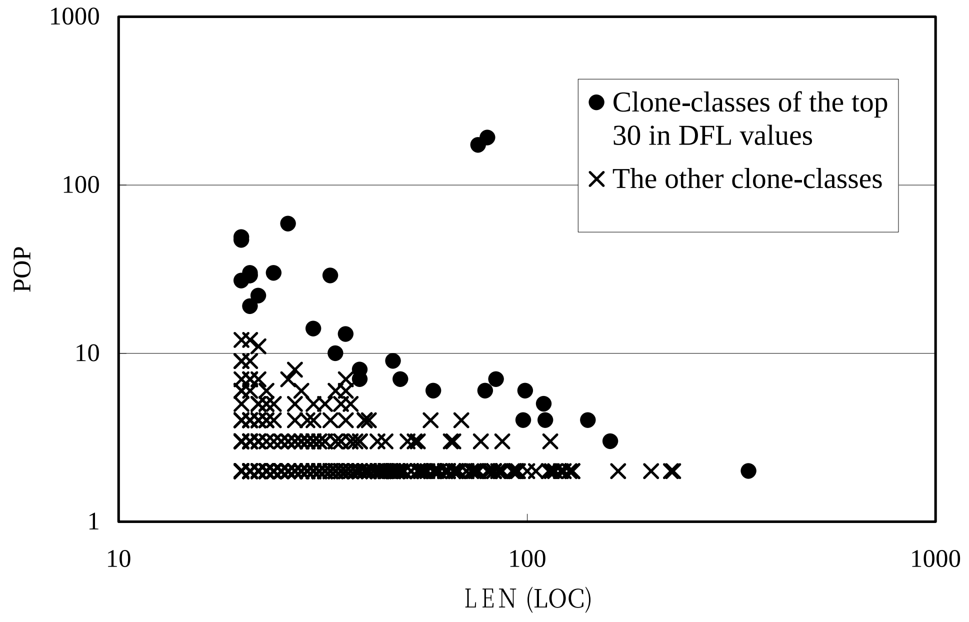


Figure 12. Population and length of clone-classes in JDK

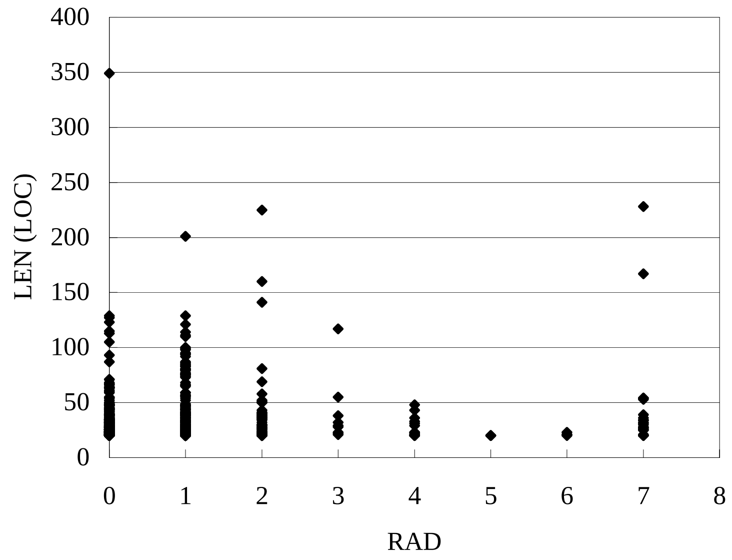


Figure 13. Length and radius of clone-classes in JDK

Note that *DFL*(*C*) >= 0, for all clone-classes *C* that satisfy *LEN*(*C*) >= 2 and *POP*(*C*) >= 2.

Applying Metrics to JDK

The data of JDK were analyzed using the metrics. Figure 12 shows the LEN and POP parameters of each clone-class. The set of clone-classes with the highest 30 DFL values is obviously different from the set with the highest LEN values or the set with the highest POP values. By investigation of source files, the clone-classes of the top 30 DFL values are classified into the following four types:

- Source files generated by `AutoMulti` (10 clone-pairs)
- Part of a switch/case statement which seems to be easily rewritten by an array (3 clone-pairs)
- Routines to apply one algorithm to many data types, that could be rewritten by generic type (5 clone-pairs)
- Instantiations of definitional computations (e.g. methods in order to put or get a value of an instance value and methods in order to change signature or private/public accessibility of the other methods) (12 clone-pairs)

Figure 13 shows the RAD and LEN parameters of each clone-class. Except for clone-classes whose RAD values are 7, most clone-classes with high LEN have small RAD value. That is, in most cases, a clone occurs between files at near directories. One of the reasons would be that copying a code portion from a distant file is a time consuming job because developer needs to search for the target code portion through many files. Another reason would be that the nearer files are more likely to implement similar functionalities.

As for all clone-classes whose RAD values are 7, 6, or 5, we investigated all the corresponding source files. All code portions of 7 are found in ‘`swing`’ subsystem, which has source files located at distant directories, `src/ com/ sun/ java/ swing` and `src/ javax/ swing`. If the all files and subdirectories in the former are moved to the latter, the

RAD values must be 3. The clone-classes of 6 and 5 are classified as access methods. We investigated clone-classes of 4 and found a clone-pair created in cut-and-paste style, within `src/javax/swing/event/SwingPropertyChangeSupport.java` and `src/java/beans/PropertyChangeSupport.java`. A class `SwingPrpertyChangeSupport` is directly derived from a parent class `PropertyChangeSupport`, and it contains methods to override those of the parent, but each overridden method is equivalent to the original. The reason for cloning is performance enhancement (the detail is described in the comment of `SwingPropertyChangeSupport`). Therefore, a careful modification process would be required for each of them.

4.3 Applied to a C++ library, Qt

To evaluate the token-based comparison for C++ source code, CCFinder was also applied to a C++ GUI framework, Qt 2.0.2 [18], which is about 240k lines in total, in 480 source files. Execution on the PC takes less than one minute. Figure 14 shows the number of clones for

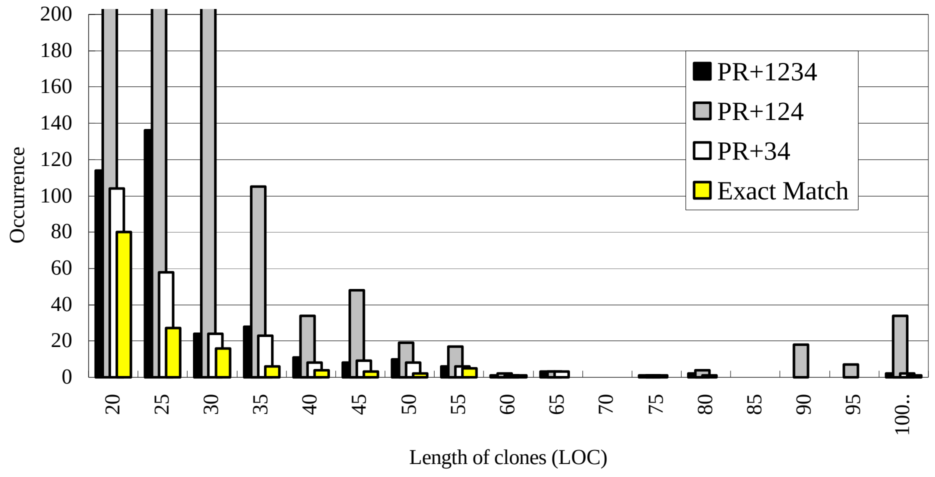


Figure 14. Occurrences against length of clone-pairs in Qt

Table 3. Subsystems cloned between operating systems

Subsystem	Linux files	FreeBSD files
zlib	arch/ppc/coffboot/zlib.c drivers/net/zlib.c	lib/libz/adler32.c lib/libz/deflate.c lib/libz/infblock.c lib/libz/infcodes.c lib/libz/inffast.c lib/libz/inflate.c lib/libz/inftrees.c lib/libz/trees.c sys/net/zlib.c
rocket	drivers/char/rocket.c	sys/i386/isa/rp.c
awe_wave	drivers/sound/lowlevel/awe_wave.c	sys/gnu/i386/isa/sound/awe_wave.c
mpu401	drivers/sound/mpu401.c	sys/i386/isa/sound/mpu401.c
sequencer	drivers/sound/sequencer.c	sys/i386/isa/sound/sequencer.c

each subset of rules. PR+1234 means that the parameter-replacement and all rules (RC1, RC2, RC3, and RC4) are applied. The difference between PR+1234 and PR+124 tells that RC3 removes many table initialization codes. By comparison of PR+1234 and PR+34, we know that RC1 and RC2 extract clone-pairs in the code, which uses templates and namespaces.

4.4 Application of CCFinder to Linux and FreeBSD systems

CCFinder was applied to million lines of code from two operating systems, Linux 2.2.14 [15] and FreeBSD 3.4 [8]. The purpose of this experiment was to investigate where and how similar codes are used between two operating systems. Linux and FreeBSD are well known Unix systems and have independent kernels written in C. The target is the source files of kernel and device-drivers, 2095 `.c` files of 1.6 million lines in Linux, and 2906 `.c` files of 1.3 million lines in FreeBSD. Clone-pairs with 20 LOC or more between two systems are extracted. This operation takes about 40 minutes on the PC.

By investigation of source codes corresponding to the clone-classes of top 30 lengths, such clones belong to 5 files or subsystems, shown in Table 3. The 3 subsystems, `awe_wave`, `mpu401`, and `sequencer` contain files with identical names between two OS's; therefore the mapping of the two OS's for the subsystems could be identified by analysis of file names. On the other hand, '`rocket`' files have different names, `rocket.c` and `rp.c`, so that the identification of the mapping is more difficult.

In case of subsystem `zlib`, the situation is more complex. Linux has two different files with the same name. FreeBSD has 9 files. Figure 15 shows a scatter plot among the files that have any clones in '`zlib`' files. *A* in the graph shows, Linux has two files named `zlib.c`, and `drivers/net/zlib.c` includes all lines of `arch/ppc/coffboot/zlib.c`. In FreeBSD system, `sys/net/zlib.c` is equal to a concatenation of eight `lib/libz/*.c`

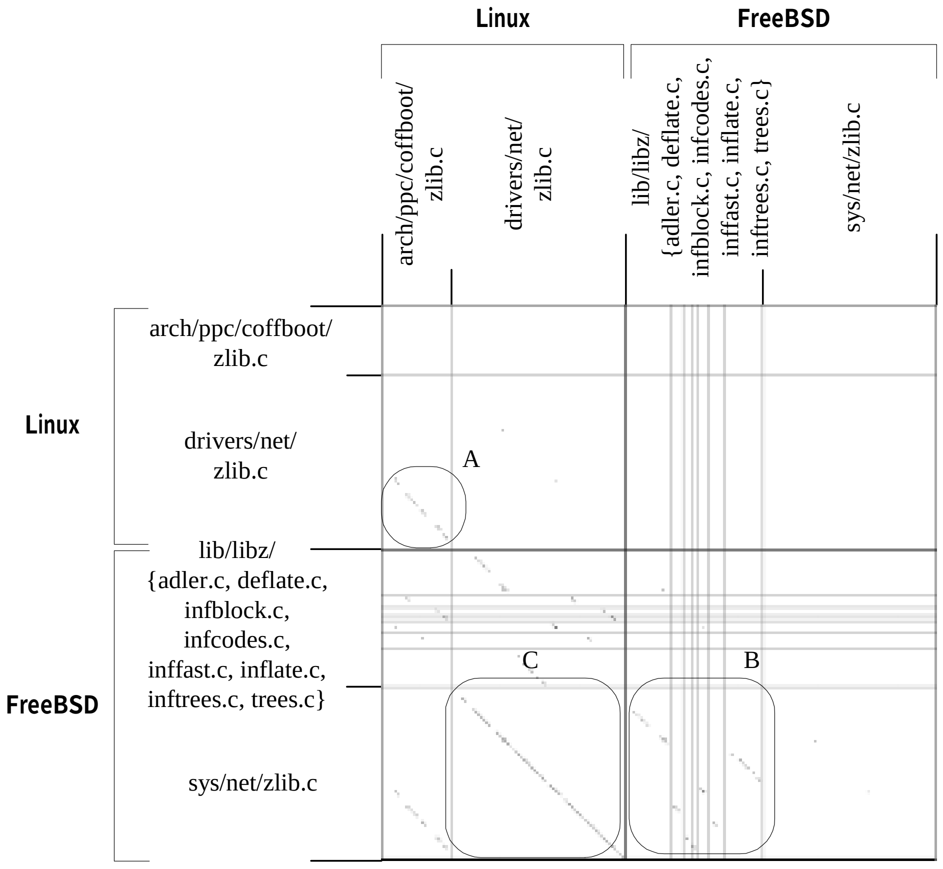


Figure 15 Clones among zlib subsystems.

files, as shown by *B* in the graph. In both operating systems (OS's), the largest `zlib.c` files contain complete source for '`zlib`' subsystem while the other files contain part of the subsystem. The two largest `zlib.c` files are almost identical between Linux and FreeBSD, as shown by *C*.

5 Discussion and Comparison with Related Works

5.1 Clone Detection Using AST

Baxter et al. proposed a technique to extract clone-pairs of statements, declarations, or sequences of them from C source files[3]. The tool parses source code to build an abstract syntax tree (AST), and compares its subtrees by hash values. The parser needs a 'full-fledged' syntax analysis for C to build AST. The clone detecting method works in bottom-up way; that is, the tool first finds small clone-pairs such as an expression and a statement, then it gathers the small clones to find larger clone-pairs such as a block and a function. There are no transformation rules as our tool has. Our transformation rules help to find more practically useful clones.

Baxter's tool expands C macros (`define`, `include`, etc) to compare code portions written with macros. The tool also detects clone-pairs in which operands are re-ordered. These features aim at some 'semantic' comparison, brought by the knowledge of the programming language. Therefore, the usrs have to know what *dialect* of C is used in source files, and also need consistent source files just like in source code computation. Our clone-extracting technique, which does not employ AST or macro-expansion, has robustness for incorrect source codes, flexibility regardless of dialect of the programming languages, and easy adaptability for various programming languages. Moreover, our tool has ability to cope with the context of tokens and the omitted tokens. Though our tool has no ability to identify clone-pairs in which expressions are re-ordered, a 'match with hole' technique described below will be complement.

5.2 An approach to calculate match with hole

Duploc[7], a clone-extractor, extracts clone-pairs of sequences of lines from source files in various programming languages. It transforms lines in an input to eliminate white-spaces and comments, and compares lines to identify clone-pairs. The tool captures clone-pairs including unmatched lines (called *holes*). Duploc also offers a visual support for clone analysis. Its user can click the scatter plot to edit code sections of clone.

Duploc employs a simple transformation rule, e.g. neglect too commonplace lines such as `break;`. However, it does not handle the cases for which our transformation rules are applied.

5.3 Abstraction and annotation

The clone-detecting method proposed in [16] uses a representation named Intermediate Representation Language (IRL) to characterize each function in the source code. A clone is defined as a pair of the function bodies that have similar metric values.

A tool named QBO [2] stores *outline* of source code (a kind of abstracted representation of source code) and answers queries on the outline. There are many other tools that analyze and store abstracted information of source code. For example, a metric tool[17] inputs source code and outputs metric values that express some characteristics of the source code. A reverse engineering method[21] and tool[19], which extract a design of software from source code, hides detail of the source code. Also, a general framework named GENOA for such code analyzers has been developed[6]. Essentially, these tools remove information of no interest from the source code. As the alternative approach, there is a tool [20] that adds annotations to source code, which are obtained through analysis of the source code. Such annotation tools do not remove any information from the source code, so that they are useful both as a supporting tool for human understanding and as a preprocessor of the other tools.

5.4 Clone analysis over versions

In [14], Laugë et al. performed tracking of clones over versions of a system, using the clone-detecting technique presented [16].

In [12], Johoson examined changes between two versions of a compiler *gcc*[9], with clone-pairs of line sequences. As for gcc, another approach to compare versions is studied in [5]; Burd and Munro observed large-scale changes to occur between version 2.7.2 and 2.8.0 of gcc, by using *dominance relation* of functions.

6 Conclusions and Further Works

In this paper, we presented a clone detecting technique with transformation rules and a token-based comparison. We also proposed metrics to select interesting clones. They were applied to several industrial-size software systems in the experiments. An experiment to compare two OS's found several subsystems that would come from a same original. Some of them have distinct file names between OS's, and some are duplicated with in a system.

The current clone-detection approach does not intend to compare source files using two or more programming languages. However, today some software systems are implemented in multi-languages (e.g. Java, SQL, HTML, and etc). We started to survey intermediate representations such as those mentioned in [2][13][16][24][25], which are suitable to compare source files in various programming languages and plan to develop a clone-detecting tool which can be used to compare versions of a particular software system when the new version is rewritten by a different methodology or in a different programming language.

References

[1] B. S. Baker, "On finding Duplication and Near-Duplication in Large Software System",

Proc. Second IEEE Working Conf. on Reverse Eng., pp. 86-95 Jul. 1995

[2] F. Balmas, “Query by Outlines: a new paradigm to help manage programs”, *Proc. of ACM SIGPLAN-SIGSOFT Program Analysis for Software Tools and Engineering (PASTE) ’99*, pp. 86-94. Toulouse, France. Sep. 1999.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees”, *Proc. of IEEE Int’l Conf. on Software Maintenance (ICSM) ’98,* pp. 368-377, Bethesda, Maryland, Nov. 1998.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. “GJ Specification”. http://cm.bell-labs.com/cm/cs/who/wadler/pizza/gj/

[5] E. Burd and M. Munro, “Evaluating the Evolution of a C Application”, *Proc. of ACM SIGSOFT Int’l Workshop on Principles of Software Evolution (IWPSE) 99*, pp. 1-5. Fukuoka, Japan. Jul. 1999.

[6] P. Devanbu , “GENOA - A Customizable, front-end Retargetable Source Code Analysis Framework”, *ACM Trans. on Software Eng. and Methodology*, vol. 9 , no. 2, Apr. 1999.

[7] S. Ducasse, M. Rieger, and S. Demeyer. “A Language Independent Approach for Detecting Duplicated Code”, *Proc. of IEEE Int’l Conf. on Software Maintenance (ICSM) ’99*, pp. 109-118. Oxford, England. Aug. 1999.

[8] FreeBSD. http://www.freebsd.org/

[9] Gnu Project. http://www.gnu.org/

[10] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, pp. 89-180. Cambridge University Press 1997.

[11] J. H. Johnson, “Identifying Redundancy in Source Code using Fingerprints”, *Proc. of IBM Centre for Advanced Studies Conference (CAS CON) ‘93*, pp. 171-183, Toronto, Ontario. Oct. 1993.

[12] J. H. Johnson, “Substring Matching for Clone Detection and Change Tracking”,

Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '94, pp. 120-126. Victoria, British Columbia, Canada. Sep. 1994.

[13] B-K. Kang and J. M. Bieman, "Using design abstractions to visualize, quantify, and restructure software", *The Journal of Systems and Software*, vol. 24, no. 2, Elsevier Science, pp. 175-187. Aug. 1998.

[14] B. Laugë, E. M. Merlo, J. Mayrand, and J. Hudepohl. "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '97*, pp. 314-321, Bari, Italy. Oct. 1997.

[15] Linux Online. http://www.linux.org/

[16] J. Mayland, C. Leblanc, and E. M. Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '96*, pp. 244-253, Monterey, California, Nov. 1996.

[17] Metamata Metrics. http://www.metamata.com/products/

[18] Qt On-line Reference Documentation. http://doc.trolltech.com/

[19] Rational Rose. http://www.rational.com/products/rose/

[20] Sapid Home Page. http://www.agusa.nuie.nagoya-u.ac.jp/person/Sapid/

[21] J. Seemann and J. W. Gudenberg, "Pattern-Based Design Recovery of Java Software", *ACM Software Eng.*, vol. 23, no. 6, pp. 10-16 1998.

[22] S. Takabayashi, A. Monden, S. Sato, K. Matsumoto, K. Inoue, and K. Torii, "The detection of fault-prone program using a neural network", *Proc. SEA-UNU/IIST Int'l Symposium on Future Software Technology (ISFST) '99*, pp.81-86. Nanjing, China. Oct. 1999

[23] The source for Java Technology. http://java.sun.com/

[24] Unified Modeling Language (UML) Resource Center. http://www.rational.com/uml/

[25] Mark Weiser, “Program slicing” *IEEE Trans. on Software Eng*. vol. SE-10, no. 4 pp. 352-357, Jul. 1984.