Repeated Instructions Removal Preprocessing for Lightweight Code Clone Detection

Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {higo,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—This paper proposes a preprocessing removing repeated instructions in the source code for lightweight code clone detection. The proposed method increases the accuracy of code clone detection.**

***Keywords*-Code clone, Detection technique, Tool development**

I. Introduction

At present, there are a variety of code clone detection tools. However, not all of them are often used in other research. Rapid detection tools are commonly-used because it is easy to apply them to large-scale software. Most of them employ line-based or token-based detection techniques. However, the techniques suffer from repeated-instructions in the source code. The presence of repeated-instructions is one factor that the techniques detect unnecessary code clones and they do not detect necessary code clones. A large body of unnecessary code clones buries the necessary code clones, which are related to copy-and-paste bugs.

This paper shows the negative impact of repeated-instructions with an actual source code and a detection tool. Then, it addresses a need of preprocessing removing repeated-instructions for increasing accuracy of code clone detection and proposes a removal method. A simplistic implementation of the proposed algorithm is presented for confirming the usefulness of the proposed method, that is, detection results from preprocessed source code are more reasonable for human than ones from original source code.

II. Natural Enemy of Lightweight Detection

In widely-used programming languages such as C/C++ or Java, there are often repeated-instructions in the source code. In Java, for example, repeated append invocations for generating SQL statements or repeated println invocations for outputting information are typical repetitions. The authors previously reported that the presence of repeated-instructions has a negative impact on code clone detection [1].

Herein, we show the negative impact with actual source code. Figure 1 is a Java class containing three methods. The only difference between the methods is the number of append invocations if we ignore the literals. Simian, which is a line-based detection tool, was applied to the source code. As a result, the following 4 clone groups were detected. For example, a code clone 4-5-6 means that it locates on a region from the 4th line to the 6th line in the source code. In this detection, we set 2-lines as the minimum detection size.

```
 1: public class Sample {
 2:   String method1() {
 3:     StringBuilder txt = new StringBuilder();
 4:     txt.append("A");
 5:     txt.append("B");
 6:     return txt.toString();
 7:   }
 8:
 9:   String method2() {
10:     StringBuilder txt = new StringBuilder();
11:     txt.append("C");
12:     txt.append("D");
13:     txt.append("E");
14:     return txt.toString();
15:   }
16:
17:   String method3() {
18:     StringBuilder txt = new StringBuilder();
19:     txt.append("F");
20:     txt.append("G");
21:     txt.append("H");
22:     txt.append("I");
23:     return txt.toString();
24:   }
25: }
```

Figure 1. Sample Source Code

- {4-5-6, 12-13-14, 21-22-23}
- {3-4-5, 10-11-12, 18-19-20}
- {11-12-13-14, 20-21-22-23}
- {10-11-12-13, 18-19-20-21}

Simian detected many code clones from the source code, which contains only 16 statements. However, Simian did not detect the following clone group, which should be the most reasonable clone group for human.

- {3-4-5-6, 10-11-12-13-14, 18-19-20-21-22-23}

The processing logics of the three methods (generate → append → return) are the same. Hence, there is a point of view that every entire method should be detected as a code clone. If so, only a single clone group is enough to let human to know that the three methods are considarably duplicated. If every method contains only a single append invocation, the entire methods would be detected as code clones. As shown in this example, the presence of repeated-instructions promotes unnecessary code clones detection and prevents necessary code clones from being detected.

```
 1: public class Sample {
 2:   String method1() {
 3:     StringBuilder txt = new StringBuilder();
 4:     txt.append("A"); // 2
 5:     return txt.toString();
 6:   }
 7:
 8:   String method2() {
 9:     StringBuilder txt = new StringBuilder();
10:     txt.append("C"); // 3
11:     return txt.toString();
12:   }
13:
14:   String method3() {
15:     StringBuilder txt = new StringBuilder();
16:     txt.append("F"); // 4
17:     return txt.toString();
18:   }
19: }
```

Figure 2. Optimized Source Code

```
String transform(String original, int threshold) {
 StringBuilder text = new StringBuilder(original);

 // loop for different window size
 for (int w_size = 1; w_size <= threshold; w_size++) {

  // loop for deleting repeated
  // substrings of a fixed window size
  for (int i = 0; (i + w_size) < text.length(); i++) {
   final int b_index1 = i;
   final int e_index1 = b_index1 + w_size;
   final String array1 = text.substring(b_index1, e_index1);
   final int b_index2 = i + w_size;
   final int e_index2 = b_index2 + w_size;
   while (e_index2 <= text.length()) {
    final String array2 = text.substring(b_index2, e_index2);
    if (array1.equals(array2)) {
     text.delete(b_index2, e_index2);
    } else {
     break;
 }}}}

 return text.toString();
}
```

Figure 3. An Implementation of the Proposed Algorithm for Java String

III. Proposed Method

The aim of the proposed method is decreasing the false positives detected. It is a preprosessing that removes the second and latters of repeated-instructions. Figure 2 shows optimized source code, which is converted from the source code of Figure 1. Also, the preprocessing outputs a mapping file between original and optimized source code. It is used for converting locations of code clones in optimized source code to ones in original source code.

If we apply Simian to the optimized source code, then only the following clone group is detected.

- {3-4-5, 9-10-11, 15-16-17}

Then, the mapping file is used for converting locations of code clones, and we obtain the following clone group.

- {3-4-5-6, 10-11-12-13-14, 18-19-20-21-22-23}

Next, we explain an algorithm that converts the source code. The inputs of the algorithm are as follows:

- **A sequence of units**: Herein, an unit is an element in the source code. In line-based detection, a unit corresponds to a line, and in token-based detection, a unit corresponds to a token.
- **A threshold of window size**: A maximum length of consecutive units that is treated as an entity of a repeated-instructions.

The output of the algorithm is a sequence of units that is the same as the input sequence except it does not include repeated-instructions. Figure 3 is an implementation of the algorithm where a sequence is Java String. Note that the implementation is a pilot one, for code clone detection, an implementation for a sequence of Java tokens. If we input abbc, abbbc, or abbbbc to the implementation, it outputs abc for all the inputs. The inputs are the same as the lines forming the 3 methods in Figure 1 where a line corresponds to a character. Also, in the case that a token corresponds to a character, the methods in Figure 1 is converted to the following strings with a correspondence table of Table I.

- method1: abcdaefgbheifgbheifgjbkefg
- method2: abcdaefgbheifgbheifgbheifgjbkefg
- method3: abcdaefgbheifgbheifgbheifgbheifgjbkefg

Whenver the implementation takes any of the above strings, it outputs abcdaefgbheifgjbkefg. Consequently, token-based technique detects the entire methods as code clones.

IV. Conclusion

This paper showed the negative impact of repeated-instructions for code clone detection, and proposed an algorithm to delete them. In the future, we are going to implement the proposed method as an actual detection tool.

Acknowledgment

The present research is being conducted as a part of the Stage Project, the Development of Next Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

[1] 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, Sep. 2007.

Table I
CORRESPONDENCE TABLE BETWEEN TOKEN AND CHARACTER

token	character	token	character
StringBuilder	a	txt	b
=	c	new	d
(	e	)	f
;	g	append	h
¨literal¨	i	return	j
toString	k		