Rearranging the Order of Program Statements for Code Clone Detection

Yusuke Sabi, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology,
Osaka University, Japan
Email: {y-sabi,higo,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—A code clone is a code fragment identical or similar to another code fragment in source code. Some of code clones are considered as a factor of bug replications and make it more difficult to maintain software. Various code clone detection tools have been proposed so far. However, in most algorithms adopted by existing clone detection tools, if program statements are reordered, they are not detected as code clones. In this research, we examined how clone detection results change by rearranging the order of program statements. We performed preprocessing to rearranging the order of program statements using program dependency graph (PDG). We compared clone detection results with and without preprocessing. As a result, by rearranging the order of program statements, the number of detected code clones is almost the same in most projects. We classified newly detected or disappeared clones manually. From our experimental results, we show that there is no newly detected clone whose statements are reordered and that there are four disappeared clones whose statements are reordered. We think three out of the four clones occurred by copy-and-paste operations. Therefore, we conclude that rearranging the order of program statements is not effective to detect reordered code clones.**

I. Introduction

A code clone (hereafter, clone) is a code fragment identical or similar to another code fragment in software. Copy-and-paste operations in code implementation are the main reason of clone occurrences [1]. It is said that the presence of clones causes bug replications and makes software maintenance more difficult [2] [3]. For example, huge software includes many clones. In this case, It is difficult for developers to check all clones with eyes. Thus, a variety of clone detection tools has been developed before now [4] [5] [6].

A variety of algorithms is used in existing clone detection tools (e.g. Parameterized-Matching algorithm [7], suffix array algorithm [8], and Smith-Waterman algorithm [9]). Tools with different algorithms detect different clones from the same source code.

However, most algorithms used in existing clone detection tools cannot detect reordered clones which are clones where their program statements are reordered. Komondoor et al. developed a clone detection tool can detect reordered clones [10]. Their tool has disadvantages that it takes a long time to detect clones, and many of detected clones are not useful for developers [1].

The authors think that some reordered clones can be detected with existing clone detection tools if the order of program statements is rearranged with a fixed rule. At this moment, there is no clone detection tool that rearranges the order of program statements.

In this research, we examined how clone detection results change by rearranging the order of program statements in target source code.

We answer following two research questions.

- RQ1···How clone detection results change by rearranging the order of program statements?
- RQ2···Do clones that are changed by rearranging occur by copy-and-paste operations?

We use program dependency graph (PDG) to rearrange the order of program statements. A PDG is directed graph representing dependencies between program statements [11]. A node in a PDG represents a program statement. An edge in a PDG represents a dependency between two program statements.

In this research, we use an extended version of PDG, which includes two kinds of dependencies. The first dependency is control dependency, which is the same as control dependency in traditional PDG. The second dependency is variable dependency, which exists between two program statements using the same variables. Variable dependencies are used to keep the current order of program statements using the same variable. Data dependency in traditional PDG is not sufficient to do that. In other words, program statements not sharing the same variables are reordered in a fixed rule in this research.

We conduct experiments on eight open source projects to answer the two RQs. As a result, we show that the number of detected clones is almost the same in most projects by rearranging the order of program statements. Moreover, we show that there is no newly detected clone whose statements are reordered, and there are four disappeared clones whose statements are reordered. We think three out of the four clones occurred by copy-and-paste operations. Therefore, we conclude that rearranging the order of program statements is not effective to detect reordered clones.

The rest of the paper is organized as follows. Section II includes terminologies, Section III describes how we rearrange the order of program statements. Section IV explains experiments, Section V answers RQs by presenting experimental results. Section VI mentions about threats to validity, Section VII discusses related works. Section VIII concludes this paper.

Clone Fragment A

```
80:   int offset = getPage() * length;
81:   Date startDate = null;
82:   if(sinceDays > 0) {
83:       Calendar cal = Calendar.getInstance();
84:       cal.setTime(new Date());
85:       cal.add(Calendar.DATE, -1 * sinceDays);
86:       startDate = cal.getTime();
87:  }
88:  List<SubscriptionEntry> results = new ArrayList<SubscriptionEntry>();
```

Clone Fragment B

```
95:   int offset = getPage() * length;
96:   List<WeblogEntryWrapper> results = new ArrayList<WeblogEntryWrapper>();
97:   Date startDate = null;
98:   if(sinceDays > 0) {
99:       Calendar cal = Calendar.getInstance();
100:      cal.setTime(new Date());
101:      cal.add(Calendar.DATE, -1 * sinceDays);
102:      startDate = cal.getTime();
103:  }
```

Fig. 1. An example of reordered code clone

II. Terminology

In this section, we explain terminologies and techniques used in this paper.

A. Types of Clones

There are three types of clones [12]. The types of clones are as follows.

Type-1 Identical code fragments except for variations of coding style. (e.g. existence of tabs, spaces, and comments)

Type-2 Identical code fragments except for variations of identifier names and literal values in addition to Type-1 variations.

Type-3 Similar code fragments differ at statement level, which means some statements are added, deleted or changed, in addition to Type-2 variations.

A clone pair is a pair of code fragments detected as clones.

B. Program Dependency Graph

Two kinds of dependencies used in PDG are as follows.

Data Dependency

- A variable v is defined in a statement s.
- A variable v is referenced in a statement t.
- There is a path where v is not redefined from s to t.

If all the above three conditions are satisfied, a data dependency exists between s and t.

Control Dependency

- A statement s is a control predicate.
- A statement t may be executed after a statement s.
- Whether t is executed or not is determined by results of s.

If all the above three conditions are satisfied, a control dependency exists between s and t.

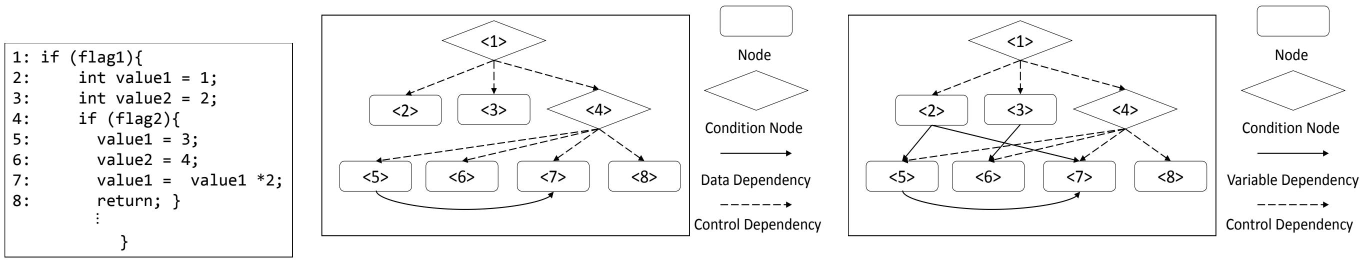
Figure 2(b) is a simple example of PDG. There is a data dependency from line 5 where a variable *value*1 is defined to line 7 where the same variable is referenced. There are control dependencies from conditional predicate of the if-statements to statements their inner statements.

C. Reordered Clones

A Reordered clone is a code clone where program statements are reordered.

An example is shown in Figure 1. In Figure 1, line 80–88 in Clone Fragment A and line 95–103 in Clone Fragment B are reordered clones. In Clone Fragment A, a variable is declared in the last line of the clone fragment (line 88), whereas the same variable is declared in the second line of the clone fragment (line 96) in Clone Fragment B.

In most existing clone detection tools, reordered clones are not detected. However, Komondoor et al. developed a clone detection tool that can detect reordered clones [10]. By using PDGs and detecting isomorphic PDG subgraphs representing clones, their tool can detect reordered clones. It takes a long time to detect isomorphic PDG subgraphs. Therefore, their tool has disadvantages that it takes a long time to detect clones, and many of detected clones are not useful for developers [1].



```
1: if (flag1){
2:     int value1 = 1;
3:     int value2 = 2;
4:     if (flag2){
5:       value1 = 3;
6:       value2 = 4;
7:       value1 =  value1 *2;
8:       return; }
         ⋮
           }
```

(a) Source Code (b) Program Dependency Graph (c) Extended PDG

Fig. 2. An example of Program Dependency Graphs

D. Extended PDG

In this research, we use an extended version of PDG, which includes two kinds of dependencies. The first dependency is control dependency, which is the same as control dependency in traditional PDG. The second dependency is variable dependency. Hereafter, we call an extended version of PDG "Extended PDG". Variable dependency is defined below.

Variable Dependency

- A variable v is defined or referenced in a statement s.
- A variable v is defined or referenced in a statement t.
- There is a path where v is not redefined from s to t.

If all the above three conditions are satisfied, a variable dependency exists between s and t.

Figure 2(c) is an example of extended PDG. There is a variable dependency from line 2 where a variable $value1$ is defined to line 5 where the same variable is also defined.

We explain why we use variable dependency. In this research, we rearrange the order of program statements so that the order of program statements sharing the same variables do not change. Two program statements which have dependency edge in PDG are regarded as program statements that cannot be reordered. If we use data dependencies in traditional PDG, two program statements referencing the same variable may be reordered.

For example, there are two append() invocations at line 6 and 7 in Figure 3. If the two invocations are executed in a different order, the object referenced with variable "result" becomes a different state. If we use data dependencies in traditional PDG, the order of the two append() invocations may be reordered because there is no data dependency between line 6 and 7. Whereas, if we use variable dependencies, the order of the two append() invocations cannot be reordered because there is a variable dependency between line 6 and 7. Therefore, we use variable dependencies to prevent inappropriate reordering between program statements.

Rearrangement the Order of Program Statements

```
1:   StringBuilder result = new StringBuilder();
2:   public String getResponse() throws IOException {
3:           result.setLength(0);
4:           String line = reader.readLine();
5:           if (line != null) {
6:               result.append(line.substring(0, 3));
7:               result.append(" "); }
```

▨ Statements cannot be reordered

Fig. 3. An example of statements cannot be reordered

III. Rearrangement the Order of Program Statements

In this section, we explain how we rearrange the order of program statements. The input and output are as follows.

- Input···Java source code
- Output···Source code where the order of the program statements is rearranged

We perform rearranging the order of program statements with following steps.

Step-1: generating extended PDG from the source code
Step-2: extracting groups of the nodes in the same block
Step-3: selecting the order of nodes which does not change control and variable dependencies
Step-4: generating source code from the extended PDG

We explain each step with an example of Figure 4. In the example of Figure 4, we give Java source code at Figure 2(a) as an input.

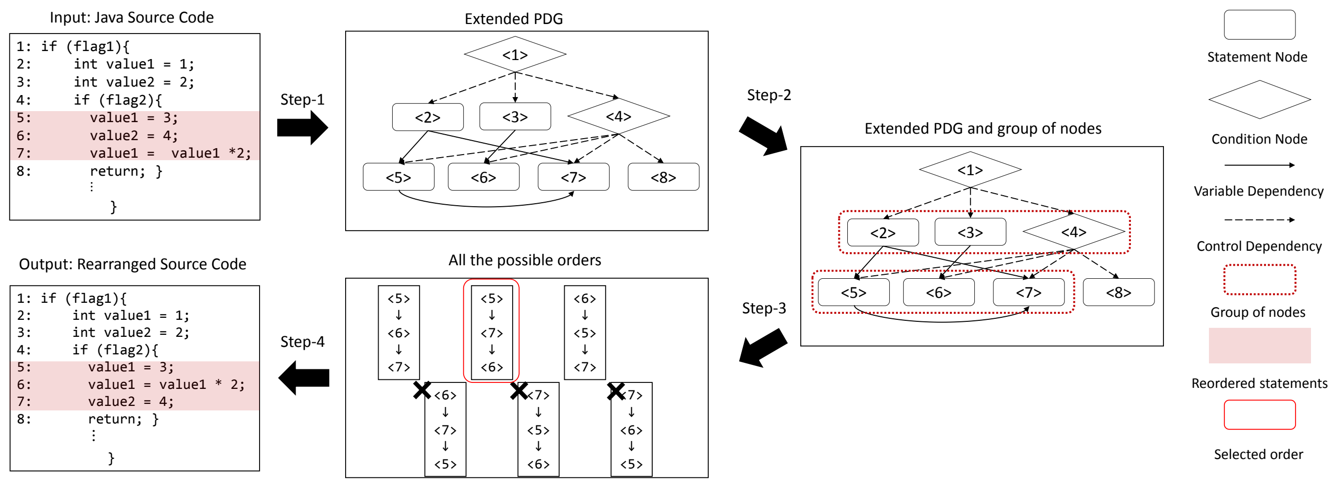
A. Step-1

In this step, we generate extended PDG from source code.

B. Step-2

In this step, we construct groups of nodes. Each group consists of nodes having control dependencies from the same node. Nodes in a group correspond to program statements in the same block in source code. In this step, we use a heuristic. If there is a return statement, a break statement, or a continue statement at the end of a given block, it is out of targets to be rearranged.

In the example of Figure 4, node <1> has control edges to node <2>, <3>, and <4>. Node <4> also has control edges to node <5>, <6>, <7>, and <8>. Thus, two groups



Input: Java Source Code

```
1: if (flag1){
2:     int value1 = 1;
3:     int value2 = 2;
4:     if (flag2){
5:        value1 = 3;
6:        value2 = 4;
7:        value1 =  value1 *2;
8:        return; }
          ⋮
        }
```

Output: Rearranged Source Code

```
1: if (flag1){
2:     int value1 = 1;
3:     int value2 = 2;
4:     if (flag2){
5:        value1 = 3;
6:        value1 = value1 * 2;
7:        value2 = 4;
8:        return; }
          ⋮
        }
```

Fig. 4. An example of rearrangement

constructed from node <2>, <3>, and <4> and node <5>, <6>, <7>, and <8>, respectively. Then we exclude node <8> from the second group because it is a return statement. Herein, we regard each condition node as its block. For example, node <4> represents node <4>, <5>, <6>, <7>, and <8>. In the example of Figure 4, the group of node <2>, <3>, and <4> and the group of node <5>, <6>, and <7> are extracted.

C. Step-3

In this step, we select an order of nodes where all pairs of nodes having dependency edges are not reordered in all groups. First, we enumerate all the possible orders of nodes for groups. Then, we exclude orders where any pair of nodes having dependency edges are reordered. Third, we normalize all variables of nodes in the remaining orders. Finally, we sort the remaining orders in the alphabetical order and select the first.

We demonstrate this step through Figure 5 and 6. Figure 5 shows source code of all the possible orders (a), (b), (c), (d), (e), and (f). In this Figure, the line numbers mean node IDs. As shown in Figure 4, there is a dependency edge between node <5> and <7>, so that these two nodes cannot be reordered. Thus, three orders (d), (e), and (f) where these nodes are reordered are excluded. Next, the variables in the remainging orders are normalized shown in Figure 6 ((a'), (b'), and (c')). We sort the three orders in the alphabetical order, and (b') get the first position. Therefore, we select (b'). In the case of node <2>, <3>, and <4>, these nodes are not reordered.

D. Step-4

Herein, we have obtained an order of nodes for each group. In this step, we generate source code based on the results of Step-3.

IV. Experiments

A. Steps of Experiments

We conduct experiments with following four steps.

Step-A: We rearrange the order of program statements.
Step-B: We detect clones from original source code and rearranged one, respectively.
Step-C: We extract newly detected or disappeared clones by applying the rearrangement.
Step-D: We classify the extracted clones manually.

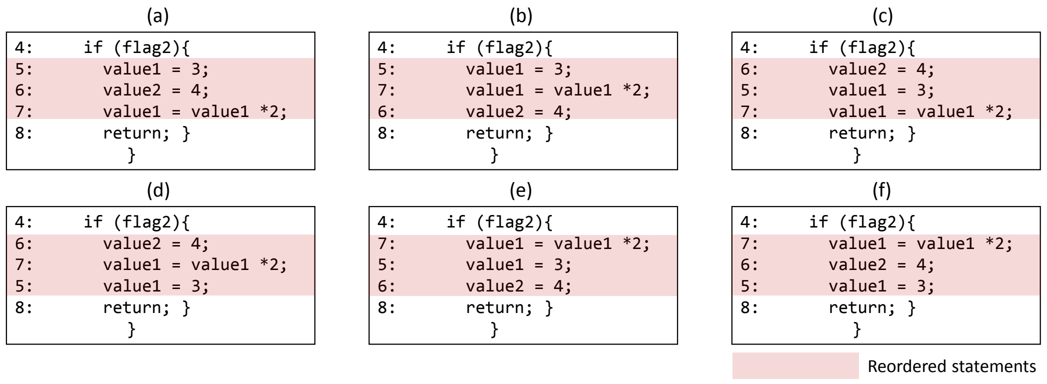
We do not explain Step-A here because it is described already in Section III. From now on, we explain Step-B, C, and D.

In the Step-B, we detect clones with an existing clone detection tool from original source code and rearranged one, respectively. We use CCFinder [13] for detecting clones. CCFinder is a clone detection tool that can detect Type-1 and Type-2 clones in a short time. We choose CCFinder because it is a popular tool. We use the default setting of CCFinder in the experiment. If we use a clone detection tool can detect Type-3 clones, there is a possibility that a part of reordered clones is detected in original source code. In this case, we can not measure the effectiveness to rearrange the order of program statements correctly. To measure the effectiveness to rearrange the order of program statements correctly, we use a clone detection tool that can detect Type-1 and Type-2 clones for detecting clones.

In the Step-C, we extract newly detected or disappeared clones by rearranging. To extract these clones, we take a mapping of clones which are included in both results. We regard non-mapped clones as newly detected or disappeared clones. We use good-value [12] to take a mapping of clones. The good-value between two clone pairs p_1 and p_2 is as follow. $p_1.f_1$,$p_1.f_2$,$p_2.f_1$, and $p_2.f_2$ represent code fragment of each clone.

$$good(p_1,p_2) = min(overlap(p_1.f_1,p_2.f_1),\\ overlap(p_1.f_2,p_2.f_2))$$

Overlap between two code fragments f_1 and f_2 is as follow. $lines(f)$ represents a group of lines in code fragment f.



```
(a)
4:     if (flag2){
5:         value1 = 3;
6:         value2 = 4;
7:         value1 = value1 *2;
8:         return; }
             }
```

```
(b)
4:     if (flag2){
5:         value1 = 3;
7:         value1 = value1 *2;
6:         value2 = 4;
8:         return; }
             }
```

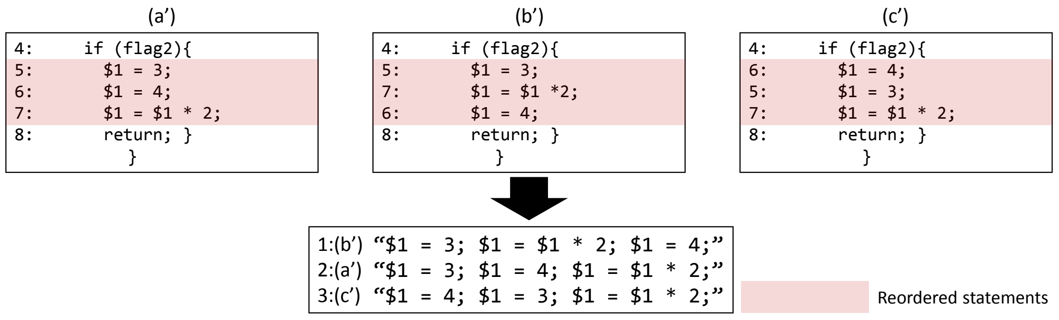
```
(c)
4:     if (flag2){
6:         value2 = 4;
5:         value1 = 3;
7:         value1 = value1 *2;
8:         return; }
             }
```

```
(d)
4:     if (flag2){
6:         value2 = 4;
7:         value1 = value1 *2;
5:         value1 = 3;
8:         return; }
             }
```

```
(e)
4:     if (flag2){
7:         value1 = value1 *2;
5:         value1 = 3;
6:         value2 = 4;
8:         return; }
             }
```

```
(f)
4:     if (flag2){
7:         value1 = value1 *2;
6:         value2 = 4;
5:         value1 = 3;
8:         return; }
             }
```

Fig. 5. Source code of all the possible orders



```
(a')
4:     if (flag2){
5:         $1 = 3;
6:         $1 = 4;
7:         $1 = $1 * 2;
8:         return; }
             }
```

```
(b')
4:     if (flag2){
5:         $1 = 3;
7:         $1 = $1 *2;
6:         $1 = 4;
8:         return; }
             }
```

```
(c')
4:     if (flag2){
6:         $1 = 4;
5:         $1 = 3;
7:         $1 = $1 * 2;
8:         return; }
             }
```

```
1:(b') "$1 = 3; $1 = $1 * 2; $1 = 4;"
2:(a') "$1 = 3; $1 = 4; $1 = $1 * 2;"
3:(c') "$1 = 4; $1 = 3; $1 = $1 * 2;"
```

Fig. 6. Normalization of variables and sorting in the alphabetical order

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|}$$

If a good-value between two clone pairs exceeds a given threshold, we take a mapping of these clones. As with literature [12], we use 0.7 as a threshold.

Originally, good-value is an evaluation method for judging whether or not the detected clones is the same as correct answer set when clone detection tool is used for one source code. In this research, we use good-value for mapping between clones included in both detection results, and it is different from the original usage method. Therefore, there is a possibility that mapping is not taken between clone pairs that should be mapped. However, even in such case, only the target of Step-D increases.

In the Step-D, we classify clones extracted in Step-C manually. One author (a graduate student) of this paper classified clones. We classify clones based on whether there is a reordering of program statements in them or not.

B. *Target Projects*

We checked out 79 Java projects from Apache repository in 2016/9/13 and chose eight target projects with their project sizes of 1MB or more (Table I).

C. *Results*

Table II shows our experimental results. We discuss the results and answer the research questions in Section V.

V. Answer to Research Questions

In this section, we answer the research questions by presenting our experimental results.

A. *RQ1*

Table II shows clone detection results. In Table II, the number of detected clones gets decreased in three (Gora, Qpid, Subversion) out of eight projects. However, the number of detected clones is almost the same. We conduct paired t-test on these two groups. As a result, no difference is found in the groups with the significance level of 5%.

As above, we answer to RQ1 as below.

Our answer to RQ1

By rearranging the order of program statements, the number of newly detected clones and the number of disappeared clones are almost the same.

TABLE I
Target Projects

Project	# of Java Files	LOC
Any23	396	46,957
BVal	350	37,520
Flume	314	48,986
Giraph	252	60,814
Gora	435	31,162
Qpid	118	27,037
Subversion	170	47,917
Wookie	295	48,537

We think that the rearranging the order of program statements is not effective to detect more clones.

B. RQ2

Table III shows the classification results of newly detected and disappeared clones. Clone pairs whose statements are not reordered are classified into the column of "No rearrangement". Whereas, clone pairs whose statements are reordered are classified into the column of "Rearrangement".

There is no newly detected clone whose statements are reordered. There are four disappeared clones whose statements are reordered. We investigate why the four clones disappeared. The reason why the clones disappeared is that statements outside the clones got inside the clones by rearranging. Because statements outside the clones were located in the clones by rearranging, these clones disappeared.

Copy-and-paste operations frequently occur in software development. It is a high possibility that bugs are duplicated by copying and pasting code fragments that have not been fully tested. Thus, there is a possibility that clones occurring by copy-and-paste operations contain duplicated bugs [3]. Therefore, we investigate whether disappeared clones occurred by copy-and-paste operations or not.

We investigate project's repositories and find that three out of four clones are changed in the same commit and changed by the same developer. From the above and code statements, we think the three clones occurred by a copy-and-paste operation. The remaining one clone is changed by different developers. We think this clone occurred by chance.

Also, we took a manual mapping of clones whose statements are not reordered. As a result, we confirmed that 196 newly detected and 196 disappeared clones that classified into the column of "No arrangement" correspond in one-to-one.

As above, we answer to RQ2 as below.

Our answer to RQ2

There is no newly detected clone whose statements are reordered by rearranging. There are four disappeared clones whose statements are reordered by rearranging. We think three out of the four clones occurred by a copy-and-paste operation. Thus, our answer to RQ2 is "Yes".

In this experiment, we showed that there is no reordered clone in newly detected clones from our experimental results.

As a future work, we plan to compare clone detection results between our tool and Komondoor's tool.

TABLE II
CLONE DETECTION RESULTS FOR TARGET PROJECTS

	# of clone pairs		Newly detected	Disappeared
	Original	Rearranged		
Any23	389	389	0	0
BVal	199	199	2	2
Flume	1,051	1,051	0	0
Giraph	582	582	62	62
Gora	703	701	0	2
Qpid	426	425	132	133
Subversion	708	707	0	1
Wookie	6,090	6,090	0	0

VI. Threats to Validity

A. Target Projects

In this research, we used eight projects from Apache repository. These projects may not be enough to generalize our findings. As a future work, we conduct experiment toward more targets which have various domains and scales.

B. Clone Detection Tool

We used CCFinder for detecting clones. If we use other clone detection tool, results will be different.

However, if we use a clone detection tool can detect Type-3 clones, there is a possibility that a part of reordered clones is detected in original source code. In this case, we may not measure the effectiveness to rearrange the order of program statements correctly. To measure the effectiveness to rearrange the order of program statements correctly, we use a clone detection tool that can detect Type-1 and Type-2 clones for detecting clones.

C. Rearranging

In this research, we consider only an order of program statements for each block. If we consider all possible orders of program statements for each block, results will be different.

VII. Related Work

Komondoor et al. developed a clone detection tool using PDG [10]. Their tool can detect reordered clones and clones which are intertwined with each other. However, their tool has disadvantages that it takes a long time to detect clones, and many of detected clones are not useful for developers.

Choi et al. examined how clone detection results change by using various types of normalization to source code [14]. As a result, it can detect clones faster by using specific normalization than without the normalization. This research is in common with our research with performing preprocess for clone detection. However, we examine how clone detection results change.

VIII. Conclution

In this research, we examined how clone detection results change by rearranging the order of program statements. In the rearranging the order of program statements, we use an extended version of program dependency graph. We compared clone detection results with and without the rearrangement. As a result, by rearranging the order of program statements, the number of detected clones is almost the same in most projects. We classified newly detected or disappeared clones manually. We showed that there is no newly detected reordered clone and that most disappeared clones occurred by copy-and-paste operations. Therefore, we conclude that by rearranging program statements is not effective to detect reordered clones. As a future work, we plan to experiment toward more targets by using different clone detection tools. Moreover, we plan to discuss similarity and differences between our rearranging technique and optimization/randomization techniques in compilers.

TABLE III
NEWLY DETECTED OR DISAPPEARED CLONE PAIRS BY REARRANGING

	Newly detected clone pairs		Total	Disappeared clone pairs		Total
	No rearrangement	Rearrangement		No rearrangement	Rearrangement	
Any23	0	0	0	0	0	0
BVal	2	0	2	2	0	2
Flume	0	0	0	0	0	0
Giraph	62	0	62	62	0	62
Gora	0	0	0	0	2	2
Qpid	132	0	132	132	1	133
Subversion	0	0	0	0	1	1
Wookie	0	0	0	0	0	0
Overall	196	0	196	196	4	200

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 25220003.

REFERENCES

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[2] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can I Clone This Piece of Code Here?" in *Proc. of the 27th International Conference on Automated Software Engineering*, 2012, pp. 170–179.

[3] J. Islam, M. Mondal, and C. Roy, "Bug Replication in Code Clones: An Empirical Study," in *Proc. of the 23rd International Conference on Program Comprehension*, 2016, pp. 68–78.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. of the 6th International Conference on Software Maintenance*, 1998, pp. 368–377.

[5] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *Proc. of the 16th International Conference on Program Comprehension*, 2008, pp. 172–181.

[6] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big-code," in *Proc. of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[7] B. S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1343–1362, 1997.

[8] H. A. Basit and S. Jarzabek, "Detecting Higher-level Similarity Patterns in Programs," in *Proc. of the 10th European Software Engineering Conference*, 2005, pp. 156–165.

[9] H. Murakami, Y. Higo, and S. Kusumoto, "Gapped code clone detection with lightweight source code analysis."

[10] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proc. of the 8th International Symposium on Static Analysis*, 2001, pp. 40–56.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

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

[14] E. Choi, H. Yoshida, Y. Higo, and K. Inoue, "Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files," *IEICE Transactions on Information and Systems*, vol. E98.D, no. 2, pp. 325–333, 2015.