2011 15th European Conference on Software Maintenance and Reengineering

Code Clone Detection on Specialized PDGs with Heuristics

Yoshiki Higo
Graduate School of Information Science and Technology
Osaka University
Suita, Osaka, Japan
Email: higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
Suita, Osaka, Japan
Email: kusumoto@ist.osaka-u.ac.jp

***Abstract*—PDG-based code clone detection is suitable for detecting non-contiguous code clones while other detection techniques, line-, token-, or AST-based techniques are not. However, PDG-based detection has lower performance for detecting contiguous code clones than the other techniques. Moreover, PDG-based detection is time consuming, so that application to actual software systems is not feasible. The present paper proposes PDG specializations and detection heuristics for enhancing PDG-based code clone detection. The experiment in this paper shows that the proposed methods are effective for PDG-based code clone detection by applying them to 4 open source systems.**

***Keywords*-code clone, program dependency graph,**

I. Introduction

A number of code clone detection tools have been developed before now. The detection tools reveal the position of the detected code clones in the source code. Since there is neither a generic nor a strict definition of code clone, each detection tool has its own unique definition for code clones that it uses to detect code clones. Consequently, different code clones are detected by different detection tools for the same source code.

Detection tools can be categorized based on the kinds of detection techniques that they employ, with the main categories including line-, token-, metrics-, Abstract-Syntax-Tree- (in short, AST), and Program-Dependency-Graph- (in short, PDG) based detection techniques. Each detection technique has its own relative advantages and disadvantages, and no technique is superior to any of the other techniques in all aspects [1], [2]. It is therefore necessary to understand the features of each detection technique and to use appropriate techniques in the context of code clone detection. The advantages and disadvantages of PDG-based detection are as follows:

- **Advantage**: PDG-based detection can detect non-contiguous code clones, whereas other detection techniques are less effective at detecting them [1]. A non-contiguous code clone is a code clone having elements that are not consecutively located on the source code. It has been reported that, after copy-and-pasting a code fragment, the pasted code is sometimes incorrectly changed or forgotten to be changed [3]. Modifications after copy and paste yield non-contiguous code clones if the modifications are larger than token level (split clones). Consequently, detecting non-contiguous code clones is of great importance for detecting incorrectly modified code.
- **Disadvantage**: The ability of PDG-based techniques to detect contiguous code clones is inferior to the other techniques [1]. In addition, the application of PDG-based detection to practical software systems is not feasible because doing so is time consuming [4], [5]

The present paper proposes PDG specializations for code clone detection. The specializations are as follows:

- **Introducing execution-next link** expands the range of program slicing, so that the ability to detect contiguous code clones is improved.
- **Merging nodes** reduces the computational cost of code clone detection. There are two reasons why PDG-based detection is time consuming. First, an enormous number of node pairs are used as slice points. Second, identifying similar subgraphs requires highly computational cost. This specialization reduces both the costs.

The present paper also uses the following heuristics for enhancing code clone detection.

- **Two-way slicing** enlarges code clone detection result because there are similar subgraphs that cannot be detected by only forward or backward slicing.
- **Reducing slice points** reduces the number of program slicing to detect code clones. Unnecessary slice points are identified and removed by this heuristics.
- **Neglecting small methods** also reduces the number of program slicing. Given a minimum size of code clone to be detected, no code clone is detected from PDGs that is smaller than the size.

II. Preliminaries

A. Program Dependency Graph

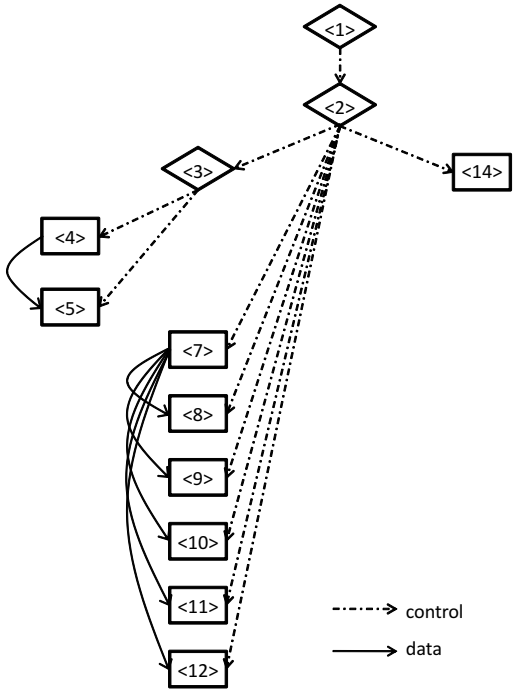
A PDG is a directed graph representing the dependencies between program elements (statements). A PDG node is a program element, and a PDG edge indicates a dependency between two nodes. There are two types of dependencies in the traditional PDG, namely, *control dependency* and *data dependency* [6]. When all of the following conditions are satisfied, a control dependency from statement s_1 to s_2 exists:

- s_1 is a conditional predicate, and
- the result of s_1 determines whether s_2 is executed.

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.12

```
 1: String sample1(){
 2:   if(this.trueOrFalse()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text = new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     return text.toString();
13:   }else{
14:     return "";
15:   }
16: }
```

(a) original source code



(b) generated PDG

Figure 1. Example of traditional PDG

When all the following conditions are satisfied, there is a data dependency from statement s_3 to s_4 via variable v:

- s_3 defines v, and
- s_4 references v, and
- there is at least one execution path from s_3 to s_4 without redefining v.

Figure 1 is a simple example of source code and a PDG generated from the source code. Labels attached to the nodes mean the lines where their elements locate in the source code. The node labeled <1> is the enter node of the PDG. In this example, there are data dependencies between nodes using variables ("prog" or "text"), and there are control dependencies between the control predicates of the if-statements and their inner statements.

B. Basic algorithm for code clone detection

This section describes the basic algorithm that we use for detecting code clones with PDGs. The algorithm was built based on Komondoor's Method [4].

In **STEP 1**, all nodes in PDGs are hashed based on their contents. Nodes having the same hash value are classified as an equivalence class. Variables and literals used in the node are converted to their type names before hashing. Consequently, the same hash value is generated from syntactically identical program elements, even if the variables are different. In the case of Figure 1, the program elements in the 8th, 9th, 10th, and 11th lines have the same hash value, even though every of the lines uses a different literal.

In **STEP 2**, a pair of nodes, ($r1$,$r2$), is selected from each equivalence class, and pairs of similar subgraphs that include $r1$ and $r2$ are identified. The starting points of the slicings are ($r1$, $r2$), and slicing is performed in lock step. If both predecessors (successors) have the same hash value, they are added to the pair of slices. In the following situations, predecessors are not added to the pair of slices, and the slicings stop:

- Predecessors ($p1$, $p2$) have different hash values.
- Predecessors ($p1$, $p2$) have the same hash value. However, $p1$ ($p2$) already exists in the slice of $r1$ ($r2$). This processing is intended to prevent an infinite loop.
- Predecessors ($p1$, $p2$) have the same hash value. However, $p1$ ($p2$) already exists in the slice $p2$ ($p1$). This processing is intended to prevent the two slices from sharing the same node.

Pairs of identified similar subgraphs are clone pairs in the basic algorithm.

In **STEP 3**, if a clone pair ($s1$, $s2$), which is a pair of similar subgraphs identified in STEP2, is subsumed by another clone pair ($s1'$, $s2'$) ($s1 \subseteq s1' \cap s2 \subseteq s2'$), it is removed from the set of detected clone pairs because reporting the subsumed clone pairs is not useful. The existence of such subsumed clone pairs enlarges the detection results unnecessarily.

In **STEP 4**, a clone set is formed from clone pairs sharing the same subgraphs. For example, two clone pairs ($s1$, $s2$) and ($s2$, $s3$) would be merged into one clone set $\{s1, s2, s3\}$.

C. Weaknesses of existing PDG-based detection methods

Previous studies revealed that PDG-based detection has the following weaknesses [1], [4], [5].

Compared to other detection techniques, the first weakness is that PDG-based detection has lower performance for the detection of contiguous code clones. This is because consecutive program elements in the source code do not necessarily have data dependency or control dependency. On the other hand, line- or token-based detection methods do not consider such dependencies but rather compare program elements textually, so that these methods are good at detecting contiguous code clones. For example, assume that we detect code clones from the source code of Figures 1(a) and 2. The region from the 3rd line to the 11th line of Figure 1(a) is duplicate to the same region of Figure 2. However, if the traditional PDG is used for clone

```
 1: String sample2(){
 2:   while(this.goOrStop()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text = new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     System.out.println(text.toString());
13:   }
14: }
```

Figure 2. compared source code

detection, the region from the 3rd line to the 6th line and the region from the 7th line to the 11th line are detected as different code clones. The two regions are connected via the node <2> in the traditional PDG. However, the nodes <2> of the both PDGs do not have the same hash value, so that program slicings stop at the nodes.

The second weakness is that PDG-based detection has a high computational complexity, which means that it is not realistic to apply PDG-based detection to actual software systems. There are two reasons why PDG-based detection has a high computational complexity. The first reason is that the number of nodes used as slice points is considerably too high, which means that a large number of slicings are performed in code clone detection. The second reason is that identifying similar subgraphs is an NP-complete problem.

III. PDG Specializations for Code Clone Detection

This section proposes methods of PDG specializations for code clone detection, which overcome the weaknesses associated with PDG-based detection.

A. Introducing execution-next link

The first specialization is the introduction of execution-next link. An execution-next link is an edge that represents the order of execution of program elements. That is, there is an execution-next link between two nodes if the program element represented by one of the nodes may only be executed after the program element represented by the other node is executed. An execution-next link is exactly equivalent to the edge of a control flow graph. Execution-next link makes it possible to detect consecutive statements as code clones, even if they have neither data nor control dependency.

Figure 3 shows the PDG including execution-next links for the source code shown in Figure 1(a). In Figure 3, the region from the 3rd line to the 6th line is directly connected with the region from the 7th line to the 11th line by execution-next link. Consequently, it is possible to detect the two regions as a single code clone from two methods in Figures 1(a) and 2.

Introducing execution-next link increases the cost of code clone detection because it leads to more edges. However, it makes it possible to merge multiple nodes, which is

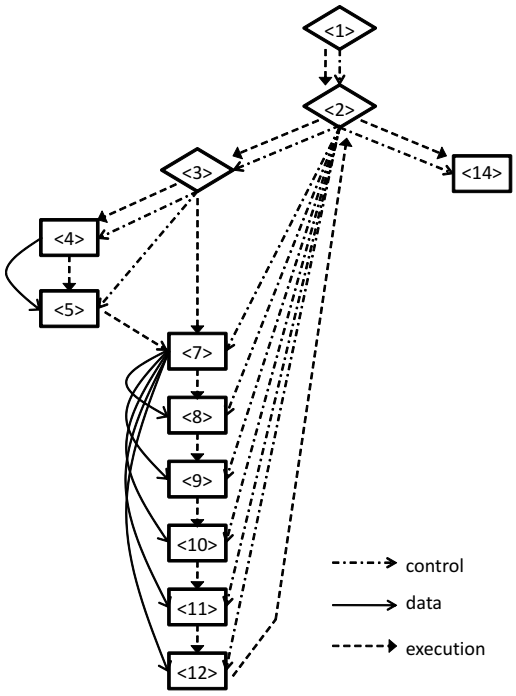


Figure 3. PDG with execution-next link

presented in the next subsection. Application of both the specializations poses no problem for computational cost (see Section VII).

B. Merging directly-connected equivalence nodes

The second specialization is merging multiple nodes as a single node. This specialization also allows improvement of detection capability, so that consecutive duplicate statements in the source code are the target of merging. One example of consecutive duplicate statements is the region from the 8th line to the 11th line in Figure 1(a). Almost all code clones detected from such a region is false positives [7]. For example, the region from the 8th line to the 9th line are identified as a code clone of the region from the 10th line to the 11th line by forward slicing on execution-next link from the pair of nodes (<8>,<10>) in Figure 3. However, human will never need such code clones. One method for not detecting such false positives is that we generate only a single node from the region, not generate nodes for every statement in the region. The number of nodes on PDGs is decreased by this method, so that required computational cost for detection is reduced. Note that this specialization does not decrease the ability of the algorithm to detect code clones because graph reachability is preserved in the specialization, so that the ability to detect non-contiguous code clones is not decreased.

A set of consecutive nodes $R = \{s \cdots t\}$ are merged into a single node if they satisfy the following conditions.

- **Condition 1:** There is a path from statement s to statement t, which is formed from only execution-next link.
- **Condition 2:** There is no execution branch in R.

In other word, none of nodes in R is conditional predicate.

- **Condition 3:** All the nodes in R have the same hash value.
- **Condition 4:** Any path (node set) subsuming R does not satisfy both the conditions 2 and 3.

We herein assume that node m was generated by merging all the nodes in R. The following explains how we construct data, control, and execution-next links from/to the node m.

Data Dependency: Assume that $data_{to}(p)$ is the set of nodes having data dependencies to node p, and P is the set of nodes included in path R, the set of node having data dependencies to node m is defined the following formula:

$$data_{to}(m) = \bigcup_{p \in P} data_{to}(p) \cap \overline{P} \quad (1)$$

Herein, $\overline{P}$ means a complement set of P, so that $P \cup \overline{P}$ is all the nodes in the PDG. In the same way, the set of node having data dependencies from node m, $data_{from}(m)$, is defined the following formula:

$$data_{from}(m) = \bigcup_{p \in P} data_{from}(p) \cap \overline{P} \quad (2)$$

Control Dependency: Condition 2 ensures that all the nodes in P have control dependencies from the same node. In the proposed method, we define that the merged node m also has a control dependency from the same node. Consequently, a set of nodes having control dependencies to m is defined as the following formula:

$$control_{to}(m) = control_{to}(s) \quad (3)$$

Also, Condition 2 ensures that there is no conditional predicate in P. Consequently, the set of control dependencies from $control_{from}(m)$ always becomes empty:

$$control_{from}(m) = \emptyset \quad (4)$$

Execution-next Link: Condition 2 ensures that there is no execution branch in R, so that it is very easy to define the sets of execution-next links from/to the node m, $execution_{to}(m)$, and $execution_{from}(m)$.

$$execution_{to}(m) = execution_{to}(s) \quad (5)$$

$$execution_{from}(m) = execution_{from}(t) \quad (6)$$

Figure 4 shows the PDG that the PDG shown in Figure 3 was converted with the proposed merging method. In Figure 4, 4 nodes located between the 8th line and the 11th line in the source code satisfy the 4 conditions, so that they were merged as a single node. This specialization reduces the following 2 kinds of computational cost.

- If the merged node appears in the path of program slicing, the slicing cost is reduced. For example, we assume that a program slicing is performed from the node <7> in Figures 3 and 4. In Figure 3, 4 hops are needed to reach the node <11> from <7>. Meanwhile, in the case of Figure 4, only 1 hop is required to reach the node <11>.

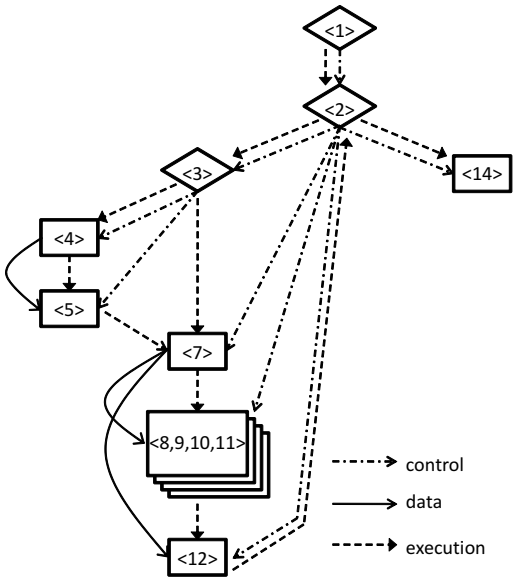


Figure 4. Result of applying the mergind method to the PDG in Fig. 3

- The merging method reduces the number of equivalence nodes, so that the number of program slicing from the nodes is reduced. For example, in the case that we detect code clones from the source code shown in Figure 1(a) and 2, the nodes located between the 8th line and the 11th line of both the methods have the same hash value. If the merging method is not applied, ${}_8C_2 = 28$ pairs of program slicing are performed. However, most of the pairs are consecutive statements in the source code, and pairwise program slicing from such pair does not detect human-wanted code clones. Meanwhile if the merging method is applied to the PDGs, the consecutive 4 nodes of both the methods are merged as a single node. Consequently, only a single pair of program slicing is performed from the nodes.

IV. Heuristics for Enhancing Code Clone Detection

This section introduces some heuristics for enhancing code clone detection.

A. Two-way slicing

The first heuristics is that both forward and backward slicings are used to identify similar subgraphs, whereas previous approaches primarily used either forward or backward slicing, but not both [4], [5], because a pair of similar subgraphs identified by forward slicing cannot be identified by backward slicing, and vice versa. Figures 5 and 6 show simple examples of such subgraphs with three program elements (<4>,<6>, and <8>).

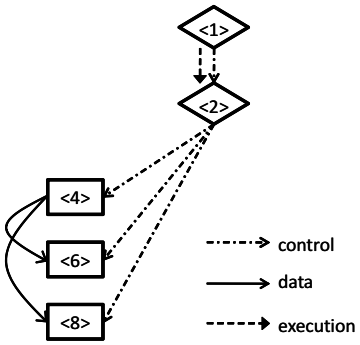
In Figure 5, one variable ("r") is referenced twice in each code clone, so that there are two data dependencies. For example, "<4> → <6>" and "<4> → <8>" are contained in the code clone <4, 6, 8>. In this instance, backward slicings cannot detect the code clones, whereas

```
1:void sample3(){
2: if(this.trueOrFalse()){
3:  ...
4:  float r = 0.05;//tax rate
5:  ...
6:  float taxA = priceA * r;
7:  ...
8:  float taxB = priceB * r;
9:  ...
10: }
11:}
```

(a) Sample code 3

```
1:void sample4(){
2: while(this.goOrStop()){
3:  ...
4:  float r = 0.05//tax rate;
5:  ...
6:  float taxA = priceA * r;
7:  ...
8:  float taxB = priceB * r;
9:  ...
10: }
11:}
```

(b) Sample code 4



(c) generated PDG

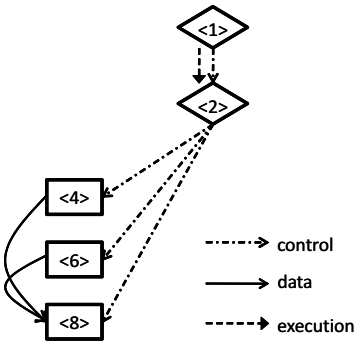
Figure 5. Example code that backward slice cannot detect similar subgraphs

```
1:void sample5(){
2: if(this.trueOrFalse()){
3:  ...
4:  int price = getPrice();
5:  ...
6:  int tax = getTax();
7:  ...
8:  int amount = price + tax;
9:  ...
10: }
11:}
```

(a) Sample code 5

```
1:void sample6(){
2: while(this.goOrStop()){
3:  ...
4:  int price = getPrice();
5:  ...
6:  int tax = getTax();
7:  ...
8:  int amount = price + tax;
9:  ...
10: }
11:}
```

(b) Sample code 6



(c) generated PDG

Figure 6. Example code that forward slice cannot detect similar subgraphs

forward slicings from the element <4> are able to detect them.

In Figure 6, two variables ("price" and "tax") are referenced in a single statement in each code clone, so that there are two data dependencies of the reference nodes. For example, "<4> → <8>" and "<6> → <8>" are contained in the code clone <4,6,8>. In this instance, forward slicings cannot detect the code clones, whereas backward slicings from element <8> can detect them.

Komondoor et al. showed that backward slicing is not always helpful to detect code clones [4], which means that the proposed method performs unnecessary executions of backward slicing. They increase computational cost. However, it is quite difficult to investigate all the situations that backward slicing is not helpful and all the situations that forward slicing is not helpful. That is why the proposed method always performs two-way slicings. The purpose of the proposed method is to detect many and large similar subgraphs as much as possible. The unnecessary executions of backward and forward slicings increase computational cost, but they neither reduce the number of detected similar subgraphs nor minify the size of detected similar subgraphs.

B. Do not use excessively large equivalence classes

In STEP 2 of the detection process, if n nodes are included in an equivalence class, then $n(n-1)/2$ pairwise slicings are performed from the equivalence class. In the case of actual software systems, several thousand or more nodes may have the same hash value, so that an enormous number of pairwise slicings are performed, which is very time consuming.

However, such large equivalence classes can occur, not because instances of such classes are generated by copy-and-paste programming, but rather because such instances are stereotyped instructions. For example, variable declarations, such as `int i;` or `int j;`, and return statements, such as `return a;` or `return b;`, are typical stereotyped instructions. Consequently, the second heuristics is a size threshold for equivalence classes. If the size of an equivalence class is larger than the threshold, then instances of that class are not used as slice points.

C. Small methods are not hashed

When we use code clone detection tools, we configure the threshold to specify the minimum size of code clones detected. Intraprocedural PDG is generally used in the field of code clone detection, so that if the size of the entire PDG of a method is smaller than the threshold, code clone is never detected from it. Consequently, the third heuristics is that small methods are not hashed in STEP 1 of the detection process. This method prevents program slicing from being performed on the PDGs of small methods.

V. Implementation

We developed a software tool, **Scorpio**, based on the proposed methods [8]. In the detection process, STEP 2 is the most time consuming. In STEP 2, a large number of program slicings are performed in order to identify similar subgraphs. We focus on the fact that every slicing is independent from one another, which means that the slicings can be performed in parallel. Consequently, Scorpio identifies similar subgraphs with multithreads. We can specify the number of threads to perform program slicing as a

command line option of Scorpio. Parallel performance efficiently uses the resource of multiple physical CPUs and multi-core CPUs, so that the detection time can be shortened.

Currently, Scorpio can handle only Java source code. However, both the basic algorithm for clone detection and the proposed methods can be applied other programming languages such as C/C++.

VI. Methodology for Evaluation

This section introduces methodologies for evaluating the proposed specialization and heuristics.

A. Clone Reference

We use freely available code clone data from [9] as a reference (a set of code clones that should be detected). In the experiment of this paper, we use the following terms:

- **clone candidates**: duplicated code detected by Scorpio.
- **clone references**: duplicated code included in the reference.

Instances of clone references are manually classified into the following three groups:

- **Type 1**: code clones that are identical to their correspondents. This type permits only differences in spaces and tabs.
- **Type 2**: code clones that include different identifiers from their correspondents.
- **Type 3**: code clones that include differences beyond Type 2 code clones. Pasted code including statement insertions, deletions, or modifications is classified in this type.

Types 1 and 2 code clones are contiguous, and Type 3 code clones are non-contiguous.

B. Good value, Ok value

We use the *good* value and the *ok* value[1] to decide whether clone candidates match clone references. In the experiment, We use 0.7 as the threshold, which is the same value used in the literature [1].

C. Recall

We calculate *recall* for evaluating detection capability. Assume that R is a detection result, S_{refs} is the set of clone references, and $S_{good}(R)$ and $S_{ok}(R)$ are sets of detected clones whose *good* or *ok* value with an instance of clone references is equal to or greater than the threshold in R. Recalls of R by using *good* and *ok* values ($Recall_{good}(R)$ and $Recall_{ok}(R)$) are defined as follows:

$$Recall_{good}(R) = \frac{|S_{good}(R)|}{|S_{refs}|} \quad (7)$$

$$Recall_{ok}(R) = \frac{|S_{ok}(R)|}{|S_{refs}|} \quad (8)$$

This experiment has two limitations related to recall.

- The clone references used in the experiments are not all code clones included in the target systems. Consequently, the absolute values of recall are meaningless. Recall can be used only for relatively comparing detection results. For the same reason, precision is not used for evaluation.
- In the present experiments, the locations of detected code clones are managed with triplet (f,s,e) as well as Bellon's experiments. f is the absolute path of the file including a given code clone, s is the start line of the code clone, and e is the end line of the code clone. Consequently, if a detected code clone is non-contiguous, 1 or more lines of non-duplicated code is incorrectly counted as duplicated code.

D. Content rate of detection result

We used the following content rate to quantitatively compare detection results:

$$CR(R_1, R_2) = 100 \times \frac{\mid S(R_1) \cap S(R_2) \mid}{\mid S(R_1) \mid} \quad (9)$$

where:

- R_1,R_2: detection result, which is a set of clone candidates. A clone candidate consists of a pair of similar graphs. A graph is formed from a set of program elements.
- $S(R)$: program elements included in detection result R. In the proposed method, the unit of program elements is PDG nodes (statements in the source code).

When the detection result R_1 includes all of the program elements in R_2, $CR(R_1, R_2)$ becomes maximum, 100%. When R_1 does not include any of the program elements in R_2, $CR(R_1, R_2)$ becomes minimum, 0%.

E. Number of node comparisons

In order to quantitatively evaluate the computational cost, we investigated the number of node comparisons in STEP 2 of the detection process. The greater the number of node comparisons, the higher the computational cost.

VII. Evaluation of each proposed method

In order to evaluate the effectiveness and efficiency of each of the proposed methods, we conducted experiments on 4 open-source systems. Specifically, we evaluated the effectiveness of the following items:

- **s1**: effectiveness of execution-next link
- **s2**: effectiveness of merging nodes
- **h1**: effectiveness of two-way slicing
- **h2**: effectiveness of neglecting excessively large equivalence classes
- **h3**: effectiveness of neglecting small methods

Table I
Target software

Software	Short name	Lines of code
netbeans-javadoc	netbeans	14,360
eclipse-ant	ant	34,744
eclipse-jdtcore	jdtcore	147,634
j2sdk1.4.0-javax-swing	swing	204,037

Table IV
NUMBER OF DETECTED CLONE REFERENCES

ID	netbeans			ant			jdtcore			swing		
	Type 1	Type 2	Type 3	Type 1	Type 2	Type 3	Type 1	Type 2	Type 3	Type 1	Type 2	Type 3
clone references	6	33	16	4	24	2	120	866	359	145	595	37
A1 *good*	1	3	2	1	2	0	19	29	8	22	34	6
A1 *ok*	2	5	2	2	2	0	76	221	50	39	54	12
A2 *good*	3	4	3	1	3	1	40	179	62	29	47	11
A2 *ok*	4	13	3	3	6	1	92	370	126	46	89	16
A3 *good*	3	5	3	0	3	1	29	148	60	32	68	12
A3 *ok*	4	16	3	3	6	1	94	390	135	56	357	21
B1 *good*	3	8	4	1	5	0	37	251	61	31	59	12
B1 *ok*	5	18	13	3	6	0	99	459	143	54	326	19
B2 *good*	3	8	5	1	6	1	51	196	67	36	71	13
B2 *ok*	4	18	13	3	8	1	98	456	138	58	354	20
B3 *good*	3	9	4	0	5	1	34	170	60	34	71	12
B3 *ok*	5	19	13	3	8	1	100	472	153	58	360	23

The experimental targets are the software systems listed in Table I. The exact versions of the systems that we used are available on [9]. In the remainder of this section, every software is called by the name of 'Short name' column. The 'Lines of code' column indicates the total number of lines of code in the .java files. We detected code clones (similar subgraphs) having six or more nodes in this experiment.

A. h1: Effectiveness of two-way slicing

First, we evaluated the degree to which detected code clones were enlarged with two-way slicings with Eq. 9. Table II lists the configurations of the detections. For example, detection A1 uses backward slicing, however ,it does not use forward slicing and execution-next link. The content rates of the detections are shown in Table III. Table III shows that two-way slicings enlarged the size of detected code clones for all the systems. This is because there often exist code clones that cannot be detected only by backward or forward slicing.

Table II
DETECTION CONFIGURATION

ID	Detection configuration		
	backward slicing	forward slicing	execution-next link
A1	○	×	×
A2	×	○	×
A3	○	○	×
B1	○	×	○
B2	×	○	○
B3	○	○	○

Table III
CONTENT RATES BETWEEN DETECTION METHODS USING TWO-WAY AND ONE-WAY SLICING

Software	Content rate	Software	Content rate
netbeans	CR(A3,A1) = 21.6 %	ant	CR(A3,A1) = 44.6 %
	CR(A3,A2) = 50.7 %		CR(A3,A2) = 55.5 %
	CR(B3,B1) = 75.8 %		CR(B3,B1) = 64.1 %
	CR(B3,B2) = 82.3 %		CR(B3,B2) = 70.3 %
jdtcore	CR(A3,A1) = 55.0 %	swing	CR(A3,A1) = 45.6 %
	CR(A3,A2) = 65.7 %		CR(A3,A2) = 56.8 %
	CR(B3,B1) = 77.7 %		CR(B3,B1) = 73.6 %
	CR(B3,B2) = 80.3 %		CR(B3,B2) = 75.8 %

Next, we investigated the effect of two-way slicings on the detection of clone references. Table IV shows the results. The result indicates that, in most cases, two-way slicings could detect more clone references than one-way slicing in isolation. In a few cases, some code clones that were detected using one-way slicing were not detected by two-way slicings (e.g., the Type 1 clone in ant). This is because some code clones were inappropriately enlarged by two-way slicings; inappropriate enlargement yields lower *good* and *ok* values, so that the code clones did not match the clone references.

Finally, we investigated the degree to which two-way slicings increased the computational cost. Table V shows that the order of computational cost is " backward slicings $<$ forward slicings $<$ two-way slicings", with the magnitude of the increase differing among targets. The largest increase, approximately 800-fold, is between A1 and A3

Table V
NUMBER OF NODE COMPARISONS

Software	ID	# of node cmpr.	Software	ID	# of node cmpr.
netbeans	A1	135,385	ant	A1	567,891
	A2	551,331		A2	4,942,013
	A3	110,356,247		A3	10,388,934
	B1	327,889		B1	951,183
	B2	663,348		B2	5,289,877
	B3	101,104,540		B3	11,295,018
jdtcore	A1	58,155,404	swing	A1	16,123,841
	A2	346,223,464		A2	331,698,602
	A3	1,382,195,451		A3	525,716,585
	B1	118,144,589		B1	28,427,935
	B2	408,615,802		B2	342,132,984
	B3	1,431,922,235		B3	528,963,280

Table VI
CONTENT RATES BETWEEN DETECTION WITH AND WITHOUT EXECUTION-NEXT LINK

Softwre	Content rate	Software	Content rate
netbeans	CR(B1,A1) = 25.8 %	ant	CR(B1,A1) = 57.4 %
	CR(B2,A2) = 59.2 %		CR(B2,A2) = 74.1 %
	CR(B3,A3) = 96.1 %		CR(B3,A3) = 93.9 %
jdtcore	CR(B1,A1) = 65.0 %	swing	CR(B1,A1) = 60.9 %
	CR(B2,A2) = 76.9 %		CR(B2,A2) = 71.5 %
	CR(B3,A3) = 94.0 %		CR(B3,A3) = 95.7 %

Table VII
CHANGE IN THE NUMBER OF NODE COMPARISONS BY INTRODUCING COMPUTATIONAL COST REDUCTION METHODS (NUMBERS IN PARENTHESES INDICATE THE RATE OF DETECTION WITHOUT APPLYING THE PROPOSED METHODS)

Software	# of node comparisons ($\times 10^3$)						
	s2: merging nodes	h2: equivalence class					h3: small methods
		thrs.100	thrs.200	thrs.300	thrs.500	thrs.1,000	
netbeans	39,507 (39.1%)	46,276 (45.8%)	100,713 (99.6%)	100,713 (99.7%)	100,713 (99.6%)	101,104 (100.0%)	100,755 (99.7%)
ant	9,818 (86.9%)	2,056 (18.2%)	4,436 (39.3%)	7,388 (65.4%)	7,388 (65.4%)	7,388 (65.4%)	8,844 (78.3%)
jdtcore	1,031,635 (72.0%)	42,941 (3.0%)	344,209 (24.0%)	393,344 (27.5%)	615,456 (43.0%)	699,470 (48.8%)	1,380,464 (96.4%)
swing	408,196 (77.2%)	16,584 (3.1%)	35,860 (6.6%)	41,774 (7.9%)	100,153 (18.9%)	137,676 (26.0%)	438,090 (82.8%)

of netbeans. The above evaluation confirmed that:

- two-way slicings can detect more clone references than one-way slicings in isolation;
- however, this increases the computational cost.

B. s1: Effectiveness of execution-next link

First, we evaluated the degree to which detected code clones were enlarged by execution-next link. In this evaluation, we calculated the content rate using Eq. 9, where R_1 is the detection result with execution-next link and E_2 is the detection result without it. The results are shown in Table VI, which indicates that:

1) content rates were relatively low when only backward slicings were used, and
2) content rates were relatively high when two-way slicings were used.

The reason for 1) is that fewer nodes can be reached using only backward slicings compared to forward slicings. The small number of node comparisons obtained with only backward slicings as shown in Table V support this assertion. In addition, execution-next link drastically increased the number of reachable nodes. The reason for 2) is that more nodes were reachable using two-way slicings, and execution-next link did not drastically increase the number of reachable nodes. In all cases, the content rates were considerably less than 100%, indicating that execution-next link always enlarged the detected code clones.

Next, we investigated the effect of execution-next link on the detection of clone references. Table IV shows the result. In most cases, more clone references were detected with execution-next link. However, in other cases, inappropriately enlarged code clones were identified, which means that the number of detected clone references decreased. The purpose of execution-next link is to improve the ability to detect contiguous code clones. However, the results of this investigation indicate that execution-next link increased the number of detected non-contiguous clone references in addition to contiguous clone references.

Table V shows the number of node comparisons for the cases in which execution-next link was either used or not used. The difference between detection with and without execution-next link has the same appearance in all the target systems. When only backward slicings were used, the number of comparisons increased by 2- or 3-fold. When only forward slicings were used, it increased by less than 2-fold. When two-way slicings were used, it remained approximately unchanged.

The above evaluation indicates that:

- Execution-next link enables the detection of both the contiguous and non-contiguous clone references;
- however, this increases the computational cost. The degree of increase depends on which slicing is used for detection.

C. s2,h2,h3: Effectiveness of merging nodes, neglecting excessively equivalence classes, and neglecting small methods

In this evaluation, we used the B3 configuration because it has the highest computational cost.

Table VII shows the degree to which each proposed cost reduction method affected the number of node comparisons. The magnitude of reduction was varied from software when the *Merging Directly-Connected Equivalence Nodes* rule was applied and the number of node comparisons decreased by 39% to 87%. Table VIII shows how much the numbers of nodes and edges are decreased by applying the method. The method reduced a small percent of nodes and edges, so that it merged bottleneck parts of PDGs for node comparisons. When the *Do not use excessively large equivalence classes* rule was applied, i.e., a smaller threshold or a larger target software systems was

Table VIII
THE SIZE OF PDGS. (HEREIN, *traditional* MEANS THE PDG THAT CONTAIN DATA AND CONTROL DEPENDENCIES. *execution* MEANS THE PDG THAT CONTAIN DATA, CONTROL, AND EXECUTION-NEXT LINKS. *merged* MEANS THE PDG WHERE CONSECUTIVE EQUIVALENCE NODES ARE MERGED)

Software	PDG	# of nodes	# of edges		
			data	control	execution
netbeans	traditional	6,557	4,700	5,626	0
	execution	6,557	4,700	5,626	6,144
	merged	6,060	4,362	5,131	5,647
ant	traditional	12,505	11,269	10,423	0
	execution	12,505	11,269	10,423	12,379
	merged	12,126	10,998	10,073	12,002
jdtcore	traditional	77,493	91,617	64,701	0
	execution	77,493	91,617	64,701	77,980
	merged	73,885	88,443	61,263	74,595
swing	traditional	82,824	75,560	68,310	0
	execution	82,824	75,550	68,310	78,110
	merged	78,783	73,026	64,370	74,050

Table IX
CONTENT RATES BETWEEN DETECTIONS USING THE COMPUTATIONAL COST REDUCTION METHODS AND DETECTIONS PERFORMED WITHOUT THE COMPUTATIONAL COST REDUCTION METHODS

Software	Content rate						
	s2: merging nodes	h2: equivalence class					h3: small methods
		thrs.100	thrs.200	thrs.300	thrs.500	thrs.1,000	
netbeans	98.45 %	100 %	100 %	100 %	100 %	100 %	100 %
ant	99.15 %	96.58 %	99.48 %	99.82 %	99.82 %	99.82 %	100 %
jdtcore	99.40 %	91.25 %	94.62 %	96.49 %	97.35 %	99.00 %	100 %
swing	98.45 %	93.94 %	96.43 %	97.34 %	98.52 %	99.70 %	100 %

Table X
NUMBER OF DETECTED CLONE REFERENCES USING THE COMPUTATIONAL COST REDUCTION METHODS

Software	Type	without the methods		s2: merging nodes		h2: equivalence class										h3: small methods	
						thrs.100		thrs.200		thrs.300		thrs.500		thrs.1,000			
		good	*ok*	*good*	*ok*	*good*	*ok*	*good*	*ok*	*good*	*ok*	*good*	*ok*	*good*	*ok*	*good*	*ok*
netbeans	Type 1	2	5	2	5	2	5	2	5	2	5	2	5	2	5	2	5
	Type 2	9	22	10	22	9	22	9	22	9	22	9	22	9	22	9	22
	Type 3	8	14	9	14	8	14	8	14	8	14	8	14	8	14	8	14
ant	Type 1	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3
	Type 2	6	10	6	10	6	10	6	10	6	10	6	10	6	10	6	10
	Type 3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
jdtcore	Type1	33	101	36	102	22	97	26	98	28	100	28	100	32	101	33	101
	Type2	314	619	322	624	131	398	184	450	297	598	298	599	313	619	314	619
	Type3	130	214	130	214	56	128	73	149	128	205	128	205	129	210	130	214
swing	Type1	31	55	31	56	31	55	31	55	31	55	31	55	31	55	31	55
	Type2	74	360	72	360	70	358	72	359	72	359	73	360	73	360	74	360
	Type3	12	22	13	23	12	21	12	22	12	22	12	22	12	22	12	22

used, the more drastically the number of node comparisons decreased. For example, for the case in which the threshold was 100 on jdtcore, the number of node comparisons was 3.0% of the value for the case in which it was not applied. When the *Small methods are not hashed* rule was applied, the number of comparisons decreased by 78% to 99%.

Next, using Eq. 9, we investigated the effect of the proposed cost reduction methods on the number of clone candidates. Table IX summarizes the content rates of the detections. When the *Merging Directly-Equivalence Nodes* rule was applied, the content rates exceed 98% for all the target systems. That means that the rule hardly had an impact on detection result. When the *Do not use excessively large equivalence classes* rule was applied, all of the content rates exceeded 90% for every target. In addition, the greater the threshold, the higher the content rate. When the *Small methods are not hashed* rule was applied, the content rates were 100% for all of the detections. This is because this rule filtered out only small methods in which no clone candidates was detected.

Third, we investigated the effect of the proposed cost reduction method on the detected clone references. Table X shows the result. When *Merging Directly-Connected Equvalence Nodes* rule was applied, the detected clone references were equal to or greater than the detection where it was not applied. Consequently, it has a small contribution to improve recall of clone detection. When the *Do not use excessively large equivalence classes* rule was applied, the number of detected clone references did not decrease for netbeans, ant, or swing. However, in the case of jdtcore, the number of detected clone references decreased at all of the thresholds. When the *Small methods are not hashed* rule was applied, the number of detected clone references was not affected at all.

Based on the above evaluation, we draw the following conclusions:

- The *Merging Directly-Connected Equivalence Nodes* rule decreases the computational cost. The degree of decreasing is varied from software. In addition, the method is partly-effective to improve recall of clone detection.
- The *Do not use excessively large equivalence classes* rule was effective in reducing the computational cost producing very few false negatives.
- The *Small methods are not hashed* rule had little impact on the computational cost. However, the application of this method did not produce any true negatives.

VIII. RELATED RESEARCH

Komondoor et al. first applied program slicing to code clone detection [4]. In their method, program statements and control predicates are nodes of PDGs. Backward slicing is mainly used in the identification, and forward slicing is performed only from matching predicates. Their method was applied to several open-source software systems written in the C language. The application result demonstrated the capability of program slicing to detect non-contiguous code clones.

Krinke proposed a fine-grained PDG, which he applied to code clone detection [5]. In the fine-grained PDG, nodes are mapped one-to-one onto nodes of an abstract syntax tree, which means that the number of nodes in fine-grained PDGs is much greater than the number of

nodes in traditional PDGs. Consequently, it takes longer to detect code clones using fine-grained PDGs. In order to alleviate this problem, Krinke proposed a k-limited search in which similar subgraphs are searched within k hops.

Jia et al. proposed a hybrid method to detect non-contiguous code clones [10]. In this method, contiguous code clones are firstly detected using a suffix-comparison algorithm as string- and token-based detection tools. Second, surrounding statements having control or data dependency with the detected code clones are merged if the corresponding code clones also have the same surrounding statements. A case study revealed that their hybrid method could rapidly detect more non-contiguous code clones than Duplix, which is an implementation of Krinke's method [5]. The hybrid method of Jia et al. is suitable for detecting both contiguous and non-contiguous code clones [10]. The difference between their hybrid method and the method proposed herein is that their method requires core parts (contiguous code clones of a certain length) to detect non-contiguous code clones, whereas the proposed method does not.

Gabel et al. proposed a scalable detection method for semantic code clones [11]. Their method is an enhanced version of an AST-based detection method, Deckard [12]. They defined the *semantic thread*, which was used for mapping and detecting similar subgraphs in PDGs, and for detecting similar subtrees in ASTs.

Roy et al. proposed a lightweight detection method for near-miss clones, which are intentionally copied and pasted codes [13]. Their method relies extensively on TXL [14] to realize the following:

- detect code clones in meaningful units,
- absorb the differences of source code formats, and
- flexibly normalize tokens and structures.

We are especially interested in the differences among code clones detected by the proposed methods, Jia's method [10], Gabel's method [11], and Roy's method [13], because all of these methods are able to successfully detect both contiguous and non-contiguous code clones and have high scalabilities.

IX. Conclusion

In the present paper we have proposed PDG specializations and heuristics for enhancing code clone detection. In addition, we experimentally confirmed that each of the proposed methods is effective either for improving the stringency of detection or for reducing detection time.

In the future, we intend to use interprocedural PDGs to detect code clones. Although using interprocedural PDGs is more time consuming, more interesting and more beneficial code clones will be detected, because if functionalities straddling two or more methods are identical, they are detected as a single clone set. We can therefore recognize larger identical functionalities in the system.

Acknowledgments

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. This study has been supported in part by Grants-in-Aid for Scientific Research (A)(21240002) and (C)(20500033) from the Japan Society for the Promotion of Science, and Grant-in-Aid for Young Scientists (B)(22700031) from Ministry of Education, Science, Sports and Culture.

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, Oct. 2007.

[2] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," in *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, Oct. 2002, pp. 36–43.

[3] M. Balint, T. Girba, and R. Marinescu, "How Developers Copy," in *Proc. of the 14th IEEE International Conference on Program Conprehension*, June 2006, pp. 56–68.

[4] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proc. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, Jan. 2000, pp. 155–169.

[5] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proc. of the 8th Working Conference on Reverse Engineering*, Oct. 2001, pp. 301–309.

[6] M. Weiser, "Program slicing," pp. 439–449, May 1981.

[7] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and Implementation for Investigating Code Clones in a Software System," *Information and Software Technology*, vol. 49, no. 9-10, pp. 985–998, Sep. 2007.

[8] "Scorpio," http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e/.

[9] "Detection of Software Clones," http://bauhaus-stuttgart.de/clones/.

[10] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsusita, "KClone: A Proposed Approach to Fast Precise Code Clone Detection," in *Proc. of the 3rd International Conference on Software Clones*, Mar. 2009.

[11] M. Gabel, L. Jiang, and Z. Su, "Scalable Detection of Semantic Clones," in *Proc. of the 30th International Conference on Software Engineering*, May 2008, pp. 321–330.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones," in *Proc. of the 29th Intenational Conference on Software Engineering*, May 2007, pp. 96–105.

[13] 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 IEEE International Conference on Program Comprehension*, June 2008.

[14] J. R. Cordy, "The txl source transformation language," in *Proc. of the 4th workshop on Language Description, Tools, and Applications*, 2006, pp. 190–210.