# Incremental Code Clone Detection: A PDG-based Approach

Yoshiki Higo, Yasushi Ueda, Minoru Nishino, Shinji Kusumoto
*Graduate School of Information Science and Technology, Osaka University,*
*1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan*
*Email: {higo,yss-ueda,m-nisinokusumoto,}@ist.osaka-u.ac.jp*

***Abstract*—It has been noted in recent years that the presence of code clones makes software maintenance more difficult. Unintended code inconsistencies may occur due to the presence of code clones. In order to avoid problems caused by code clones, it is necessary to identify where code clones exist in a software system. Consequently, various kinds of code clone detection techniques have been proposed before now. In particular, incremental code clone detection attracts much attention in the last few years, and line-based and token-based incremental detection methods have been proposed. In incremental detections, code clone detection results or their intermediate products persist by using databases, and it is used in next code clone detection. However, no incremental detection technique has been proposed for PDG-based detection, which requires much more time to detect code clones than line- or token-based detection. In this paper, we propose a PDG-based incremental code clone detection technique for improving practicality of PDG-based detection. A prototype tool has been developed, and it has been applied to open source software. We confirmed that detection time is extremely shortened and its detection result is almost the same as one of an existing PDG-based detection technique.**

***Keywords*-Code Clone, Incremental Detection**

## I. Introduction

No software has no code clones. Recently, the presence of code clones is pointed out as one factor that makes software maintenance more difficult. A code clone is a code fragment that has similar or identical code fragments in the source code. Code clones occur for various reasons such as copy-and-paste programming. Copy-and-paste programming generates similar or identical logics in multiple places in software. If an instance of code clones has to be modified for correcting bugs, adding new functionalities, or adaptive maintenance tasks, it is possible that its correspondents requires the same modification. In such a situation, if developers or maintainers are unaware of its duplication, unintended code inconsistencies occur in the source code.

It is possible to decrease the negative impact of code clone on software maintenance by recognizing where code clones exist. For example, if a set of code clones is merged as a single function or method, inconsistent changes never occur in it in the future. Even if it is impossible to perform such a refactoring, recognizing code clones raises developer awareness, which could help to prevent the source code from including unintended inconsistencies.

Various kinds of automatic code clone detection techniques have been proposed before now. Each detection technique has its own unique definition for code clones, so that different code clones are detected by different detection tools for the same source code. Each detection technique has it own relative advantages and disadvantages, and no technique is superior to any of the other techniques in all aspects [3], [4]. It is therefore necessary to understand the features of each detection technique and to select appropriate techniques in the context of code clone detection.

The advantage of PDG-based detection is that it can detect non-contiguous code clones, whereas other detection techniques are less effective at detecting them [3]. 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 copying and pasting a code fragment, the pasted code is sometimes incorrectly changed or forgotten to be changed [2]. Modifications after copy-and-paste generate 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. On the other hand, there are two big disadvantages in PDG-based detection. One is that the ability for detecting contiguous code clones is inferior to the other techniques [3]. The other is that the application of PDG-based detection to practical software systems is not feasible because doing so is time consuming [14], [15]. For the former problem, the authors have proposed a specialized PDG that including a new dependency, execution-next link, which is intended to improve the detection ability for contiguous code clones [7]. In this paper, we work on the improvement of the latter problem.

In the last few years, *incremental* code clone detection attracts much attention. Incremental detection persists data created during code clone detection, and uses it in the next detection. By using such persistent data, in the second time or later detections, detection time extremely shortens. Before now, line-based and token-based incremental detection techniques have been proposed [6], [9].

In this paper, we propose a PDG-based incremental detection technique. As mentioned above, PDG-based detection requires much time, so that it is unrealistic to apply it to large-scale software systems. However, PDG-based incremental detection realizes instant non-contiguous code

clone detection after the detection database is prepared. Existing incremental detection techniques do not detect non-contiguous code clones. For realizing PDG-based incremental detection, we focus on edges in PDGs, which are suitable to persist PDG data. We define a clone pair by edge-level matching, and propose an algorithm detecting clone pairs under the definition.

Incremental detection is more helpful in various contexts of code clone detection. There are several methods that construct some kinds of historical data with code clone analysis from CVS or SVN repository [8], [13], [16], [18]. If we detect code clones without incremental techniques, all the source files of every revision have to be analyzed. That is very time consuming. However, if we use an incremental detection technique, detection time will extremely shorten because only a few files are updated in a revision and incremental technique analyze only the updated files. Another situation is that, in order to avoid unintended inconsistent changes, code clone detection is applied after maintainer identified a code fragment that causes a fault. In this context, what the maintainer wants is code clones that are related to the buggy code fragment. Detecting code clones from the entire system is overkill and time consuming. Incremental technique can identify only the required code clones instantly. In this situation, detecting code clones from the entire system only once at the beginning and using the result in every bug fixes should be avoided. Because, source code is updated by every bug fix and every new functionality addition. Using an initial detection result after several modifications were added is dangerous because the state of code clones has been changed by the modifications.

The proposed PDG-based incremental technique realizes contiguous and non-contiguous code clones instantly. The contributions of this paper are as follows:

- We propose a PDG-based incremental code clone detection technique.
- We confirm that the proposed technique is enough rapid as an instant clone detection technique and the detection result is appropriate as a PDG-based detection.

## II. Program Dependency Graph

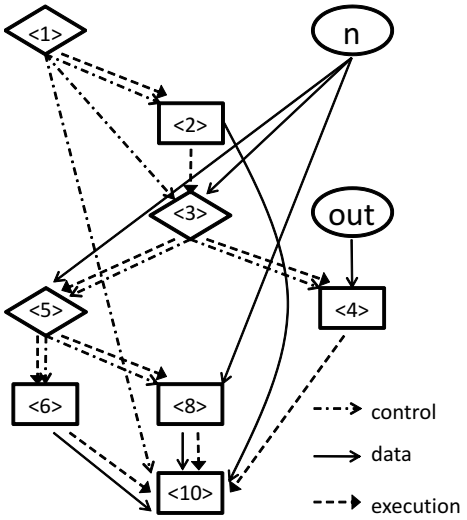
A PDG is a directed graph representing the dependencies between program elements (statements or conditional predicates). A PDG node is a program element, and a PDG edge indicates a dependency between two nodes. There are two types of dependencies in a traditional PDG, namely, *control dependency* and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

- $s_1$ is a conditional predicate, and
- the result of $s_1$ directly influences whether $s_2$ is executed.

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

```
 1: int fibonacci(int n){
 2:   int value = -1;
 3:   if (n <= 0) {
 4:     System.out.println(
                    "Illegal parameter");
 5:   } else if (n == 1 || n == 2) {
 6:     value = 1;
 7:   } else {
 8:     value = fibonacci(n - 2) +
                    fibonacci(n - 1);
 9:   }
10:   return value;
11: }
```

(a) original source code




(b) generated PDG

Figure 1. A PDG Example

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

PDGs used in this research is a specialized one. It has one more dependency, *execution-next link*. By adding execution-next link, its detection ability is enhanced [7]. 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 just after the program element represented by the other node is executed. An execution-next link is equal to an edge of a control flow graph. Execution-next link makes it possible to detect consecutive program elements as code clones, even if they have neither data nor control dependency.

Figure 1 is a sample PDG generated from a simple source code, which calculates Fibonacci series. 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. There are two nodes labeled `n` or `out`. The former

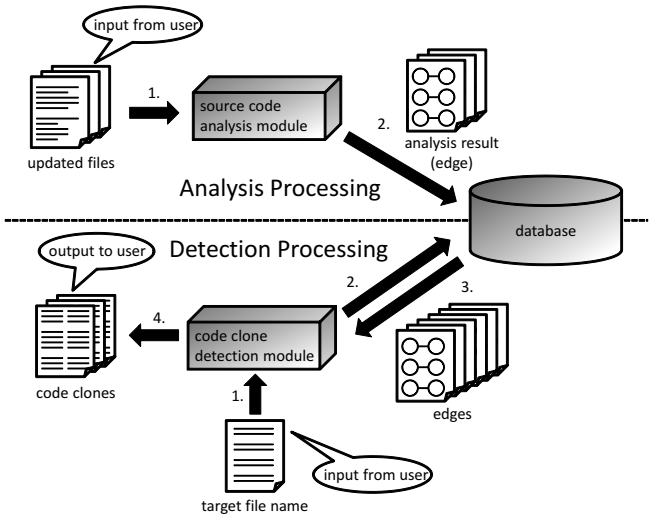



Figure 2. Overview of the Proposed Method

presents a formal parameter of the method, and the latter presents a global variable defined outside the method.

## III. Proposed Method

Figure 2 shows an overview of the proposed method. It consists of two processings, *analysis processing* and *detection processing*. *Analysis processing* analyzes source files and builds PDGs. Then, PDGs information is stored into the database. *Detection processing* detects code clones by using PDG information stored in the database. In the *detection process*, raw source files are not analyzed.

Note that, in the proposed method, data dependencies are built by variables names. The program analysis in the proposed method does not include points-to analysis. Points-to analysis requires a whole-program analysis. Hence, PDG-based incremental detection cannot be made if it includes points-to analysis.

The remainder of this section is organized as follows: Subsection III-A defines several terms used in the proposed method; Subsection III-B and III-C describes *analysis processing* and *detection processing* respectively.

### A. Definition

Firstly, we define PDGs as follows.

**Definition 1** *PDGs used in this paper are connected graphs*[1]*, so that a PDG $g$ can be represented as a set of edges existing in it.*

$$g \quad := \quad \{e_1, e_2, \cdots, e_m\} \tag{1}$$

Next, edges in PDGs are defined as follows.

[1]Strictly, if a method has a parameter that is not referenced in it, the node of the parameter is isolated. We do not care such an isolated parameter node.

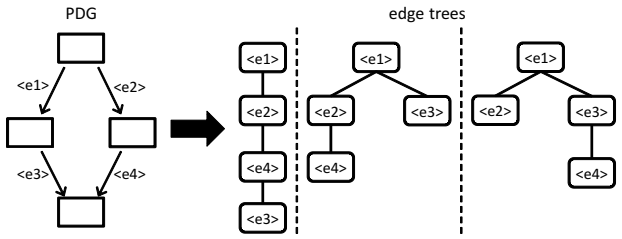



Figure 3. Edge trees generated from a PDG

**Definition 2** *Edge $e$ can be represented as the following formula.*

$$e \quad := \quad (v_s, v_e, t)$$

*Where, $v_s$ is the start node, $v_e$ is the end node, and $t$ is the type of $e$ (data, control, or execution-next).*

Next, we define a incident relationship between two edges as follows.

**Definition 3** *The incident relationship between edges $e_1 = (v_1, v_2, t_1)$ and $e_2 = (v_3, v_4, t_2)$ is represented as the following formula.*

$$\begin{aligned} incident(e_1, e_2) \quad := \quad & (v_1 = v_3 \wedge v_2 \neq v_4) \vee \\ & (v_1 = v_4 \wedge v_2 \neq v_3) \vee \\ & (v_2 = v_4 \wedge v_1 \neq v_3) \vee \\ & (v_2 = v_3 \wedge v_1 \neq v_4) \end{aligned} \tag{2}$$

Next, we define a set of paths between two nodes by using the incident relationship.

**Definition 4** *PDGs used in this paper are connected graphs, so that there are at least a path between any two edges. Herein, we assume that $PATHS(e_i, e_j)$ is a set of paths between two edges $e_i$ and $e_j$. Note that, $PATHS$ does not includes paths with redundant edges.*

Herein, we define a new data structure, **edge tree**.

**Definition 5** *On a PDG $g = \{e_1, e_2, \cdots, e_n\}$, an edge tree is a tree whose nodes are graph edges and whose edges are incident relationships in $g$.*

Figure 3 shows a simple example. There are multiple edge trees that can be generated from the a PDG. Figure 3 shows 3 edge trees whose roots are edge $e_1$. As shown in this example, an edge tree does not have all the incident relationships in a PDG. Consequently, we defines a set of edge trees as follows.

**Definition 6** *$TREES(g)$ is the entire set of the edge trees, every of which can be made from a PDG $g$.*

Also, we define a path on an edge tree as follows.

**Definition 7** *$treepath(t,e)$ is a path from the root node to node $e$ on edge tree $t$.*

Herein, the theorem 1 is self-evident.

**Theorem 1**

$$treepath(t,e) \quad \in \quad PATHS(e_r,e) \tag{3}$$

*where $e_r$ is the root note of edge tree $t$.*

Also, in a connected graph, $PATHS(e_r,e) \neq \emptyset$ is always established, which means that there is at least a path from $e_r$ to any other edge $e$ ($e \in g$). Consequently, there is at least an edge tree that can be generated from a connected graph $g$.

Next, we define a equivalence relationship on edges.

**Definition 8** *The equivalence relationship between two edges $e_1 = (v_1, v_2, t_1)$ and $e_2 = v_3, v_4, t_2$ can be represented as the following formula.*

$$e_1 \equiv e_2 \quad := \quad (t_1 \equiv t_2) \wedge (v_1 \equiv v_3) \wedge (v_2 \equiv v_4) \tag{4}$$

Node equivalence should be defined according to the context of code clone detection. If we want to detect only the exact duplications on the source code, $v_1 \equiv v_2$ should becomes *true* if and only if the strings of the two nodes are exactly identical. If we want to detect exact and similar code as code clones, $v_1 \equiv v_2$ should be reflected by certain kinds of code normalizations.

By using the definition 8, an equivalence relationship of edge paths $p_1 = (e_1, e_2, \cdots, e_n)$ and $p_2 = (f_1, f_2, \cdots, f_m)$ is defined as follows.

**Definition 9**

$$p_1 \equiv p_2 \quad := \quad (|p_1| = |p_2|) \wedge \forall i (e_i \equiv f_i) \tag{5}$$

That is, two edge paths $p_1$ and $p_2$ has the equivalence relationship if they has the same number of edges, and every pair of the edges has the equivalence relationship.

Also, we define a equivalence relationship on edge trees $t_1 \in TREES(g_1)$ and $t_2 \in TREES(g_2)$.

**Definition 10**

$$\begin{aligned} t_1 \equiv t_2 \quad := \quad & (|g_1| = |g_2|) \wedge \\ & \exists (e_1, e_2, \cdots, e_{|g_1|}) \exists (f_1, f_2, \cdots, f_{|g_2|}) \forall k \\ & ( \bigcup_{1 \leq i \leq |g_1|} \{e_i\} = g_1 \wedge \\ & \bigcup_{1 \leq j \leq |g_2|} \{f_j\} = g_2 \wedge \\ & treepath(t_1, e_k) \equiv treepath(t_2, f_k)) \end{aligned} \tag{6}$$

That is, edge tree $t_1$ is equivalent to $t_2$ if and only if the numbers of nodes in $t_1$ and $t_2$ are the same and all the paths from any node to the root on $t_1$ has an equivalant path on $t_2$.

By using the these definitions, clone pair is defined as follows.

**Definition 11** .

$$\begin{aligned} clonepair(s_1, s_2) \quad := \quad & (s_1 \cap s_2 = \emptyset) \wedge \\ & \exists t_1 \exists t_2 (t_1 \in TREES(s_1) \wedge \\ & t_2 \in TREES(s_2) \wedge \\ & t_1 \equiv t_2) \end{aligned} \tag{7}$$

where $s_1$ and $s_2$ are connected subgraphs on PDG $g_1$ and $g_2$, respectively.

In this paper, if two graphs have at least a pair of equivalent edge trees, they are regarded as a clone pair. However, it is redundant to output all the clone pair under the definition 11. Consequently, only the clone pairs satisfying the following condition is output.

**Definition 12**

$$\begin{aligned} outputclonepair(s_1, s_2) \quad := \quad & clonepair(s_1, s_2) \wedge \\ & \neg \exists (s_1', s_2') (clonepair(s_1', s_2') \wedge \\ & s_1 \subset s_1' \wedge \\ & s_2 \subset s_2') \end{aligned} \tag{8}$$

That is, a *outputclonepair* is a clone pair that is not subsumed by any other *clonepair*.

The differences and relationships of the proposed detection and traditional PDG-based detections are discussed in Subsection V-B.

### *B. Analysis Processing*

Figure 2 shows an overview of *analysis processing*. The input is updated source files and the output is updated database. Firstly, source files are input to the analysis module (label 1), then PDGs are built from methods included in the input files. All the edges are extracted from the PDGs. They are stored into the database (label 2). If the database already has edges extracted from the older version of the files, they are removed before storing the new version.

### *C. Detection Processing*

Figure 2 shows an overview of *detection processing*. The inputs are target source files and database, and the output is a set of clone pairs related to the target source files. In *detection processing*, a user specifies files where he wants to detect code clones (label 1). Then, the detection module queries the database with the specified file names (label 2). The database returns a set of edges that are equivalent to the edges in the specified files (label 3). Then, the detection

module constructs clone pairs from the edges, and output them (label 4).

In order to realize *detection processing*, we construct an algorithm that detects clone pairs related to a specified method. We apply the algorithm to every method included in the target files. The algorithm is shown in Algorithm 1.

**Algorithm 1** detect ($m$)

**Input:** $m$: a method
**Output:** $C$: a set of clone pairs related to $m$

```
1: C ← ∅
2: for all e1 such that e1 ∈ m do
3:    for all e2 such that e2 ≡ e1 ∧ e2 ≠ e1 do
4:       C ← C ∪ create(e1, e2, ∅)
5:    end for
6: end for
7: return C
```

First of all output $C$ is initialized with $\emptyset$ in the 1st line. Then, all the edges included in the input method $m$ are obtained from the database. The edges have method IDs. A method ID indicates a method whose PDG includes the edge. That is, the 2nd line obtain all the edges included in the input method $m$ without re-generating a PDG from it. In the 3rd line, all the edges that are equivalent to the edges in method $m$ are extracted from the database. Then, for every pair of the edges, clone pairs are detected by the *create* algorithm, which is shown in Algorithm 2.

**Algorithm 2** create ($e_1$, $e_2$, $E$)

**Input:** $e_1$, $e_2$:a pair of edges, $E$: a set of edges that have been already checked.
**Output:** $(S_1, S_2)$: a output clone pair satisfying $e_1 \in S_1$ and $e_2 \in S_2$

```
1: (S1, S2) ← ({e1}, {e2})
2: E ← E ∪ {e1, e2}
3: for all ex such that incident(e1, ex) ∧ ex ∉ E do
4:    for all ey such that ex ≡ ey ∧ incident(e2, ey) ∧ ey ∉ E
      do
5:       (Sx, Sy) ← create(ex, ey, E)
6:       (S1, S2) ← (S1 ∪ Sx, S2 ∪ Sy)
7:    end for
8: end for
9: return (S1, S2)
```

Firstly, $(S_1, S_2)$ is initialized and $E$ is updated with the input pair of edges in the 1st and 2nd lines respectively. Then, all the pairs of edges that are incidents of the input pair are obtained in the 3rd an 4th lines. For every of the obtained pairs, algorithm *create* is applied recursively.

## IV. Implementation

Herein, we describe a prototype tool that we have developed based on the proposed method. The prototype is written in Java language, and it uses SQLite database system. The database is stored in disk not memory.

### A. Equivalence relationship on PDG nodes

In order to permit token level differences on code clones, equivalence relationship on PDG nodes is defined as follows:

**Definition 13**

$$v_1 \equiv v_2 \quad := \quad normalize(e_1) = normalize(e_2) \qquad (9)$$

where $e_1$ and $e_2$ are program elements, and $v_1$ and $v_2$ are PDG nodes generated from them. *normalize* is a normalization function, which replaces variables and literals with special tokens with the following policies:

- **variable**: A special token is prepared for every of different variables. The same variable is replaced with the same special token.
- **literal**: A special token is prepared for every type of literal. The same type literals are replaced with the same special tokens.

There are other reasonable replacement policies, for example, replacing variables and literals with the same token would be work well.

### B. Database

Edges in PDGs are stored into SQL database. The hash values of edges are simultaneously stored too. Database queries in *detection processing* are processed with the hash values. In the implementation, we use MD5 algorithm for hashing edges.

### C. Approximating Algorithm

The algorithm described in Subsection III-C is not efficient from the viewpoint that it uses a pair of equivalent edges $(e_1, e_2)$ to detect two or more clone pairs. In order to speed up code clone detection, we make a following assumption.

**Assumption 1** *A pair of equivalent edges is included only in a single clone pair. No edge pair is shared by two or more clone pairs.*

Under this assumption, algorithms detect and create are changed as shown in Algorithms 3 and 4.

In detect' and create' algorithms, *DONE*, which is a set of edge pairs that have been already checked, is introduced. If an edge pair is added to a clone pair, it is also added to *DONE*. By checking *DONE*, every pair of equivalence edges is added into a clone pair only once on an execution of detect' algorithm. However, this approximation prevents some code clones from being detected. Figure 4 shows such an example. In this figure, there are two PDGs $G_1$ and $G_2$. Labels attached to the nodes represent their hash values. In this case, there are two clone pairs between $G_1$ and $G_2$. The

**Algorithm 3** detect' ($m$)

**Input:** $m$: a method
**Output:** $C$: a set of clone pairs related to $m$

```
1: C ← ∅, DONE ← ∅, E ← ∅
2: for all e1 such that e1 ∈ m do
3:    for all e2 such that e2 ≡ e1 ∧ e2 ≠ e1 ∧ (e1, e2) ∉ E do
4:       C ← C ∪ create'(e1, e2, E, DONE)
5:    end for
6: end for
7: return C
```

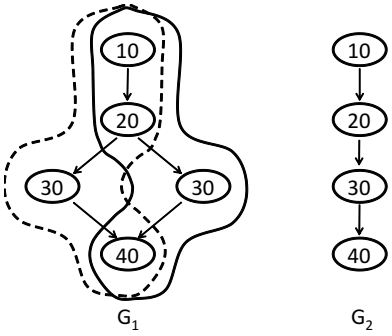



Figure 4. An example that detect' algorithm cannot detect as clone pairs

**Algorithm 4** create' ($e_1$, $e_2$, $E$, $DONE$)

**Input:** $e_1$, $e_2$:a pair of edges, $E$: a set of edges that have been already checked, $DONE$: a set of edge pairs that have been already checked.
**Output:** $(S_1, S_2)$: a output clone pair satisfying $e_1 \in S_1$ and $e_2 \in S_2$

```
1: (S1, S2) ← ({e1}, {e2})
2: E ← E ∪ {e1, e2}
3: DONE ← DONE ∪ (e1, e2)
4: for all ex such that incident(e1, ex) ∧ ex ∉ E do
5:    for all ey such that ex ≡ ey ∧ incident(e2, ey) ∧ ey ∉
      E ∧ (ex, ey) ∉ DONE do
6:       (Sx, Sy) ← create(ex, ey, E)
7:       (S1, S2) ← (S1 ∪ Sx, S2 ∪ Sy)
8:    end for
9: end for
10: return (S1, S2)
```

two code clones in $G_1$ are areas surrounded by a dashed line and a solid line respectively. This situation is conflicting with the assumption 1. The prototype tool has implementations of both the detect and detect' algorithms.

### D. Aborting Detection

The detection algorithm shown in Algorithms 1, 2, 3, and 4 requires very high cost under a special situation. Figure 5 shows such a situation. In this method, there are 1,000 case entries in a switch-statement, and every statement in

```
switch(x){
  case 1:
    statement1;
  case 2:
    statement2;
    :
    :
  case 1000:
    statement1000;
}
```

Figure 5. Code for which Algorithm 4 is very expensive

all the case entries has the same hash value. Thus, there are an enormous number of pairs of equivalent edges, and every pair of the edge pairs satisfies the incident relationship. In order to avoid consuming much cost on such a special situation, the prototype aborts detecting code clones in the method if a large number of edges in the method has the same hash value. Then, the prototype just notices that it aborted detecting from the method.

## V. Evaluation

This section describes an experimental study that we conducted in order to evaluate the proposed method. This experiment consists of the following two sub-experiments.

- **Experiment 1**: Evaluation on execution time of the proposed method. This evaluation was conducted for confirming the efficiency of the proposed method.
- **Experiment 2**: Evaluation on detection quality of the proposed method. This evaluation was conducted for confirming the usefulness of the proposed method.

Both the experiments were performed with and without the optimizations described in Subsections IV-C and IV-D.

The experiments were performed on a personal workstation, which is shown as follows:

- **CPU**: Intel Xeon E5405 (quad-core, 2.0GHz)
- **Memory**: 8GB
- **OS**: Windows 7 Enterprise (64bit)

In this evaluation, we use Ant, which is an open source software system written in Java language because the current implementation can handle only Java language. However, it is possible to handle other programming languages if we develop a component that builds PDGs from the other languages. 5,903 revisions of Ant are analyzed and the the number of files and the LOC of the last revision are 804 and 203,580, respectively.

### A. Experiment 1: Detection Time

We investigated how efficiently the proposed method can detect code clones in the following two contexts.

- **Context 1**: Code clones are detected from every revision. This kind of detection is often performed in research related to code clones. Recently, several research efforts have investigated whether the presence of code

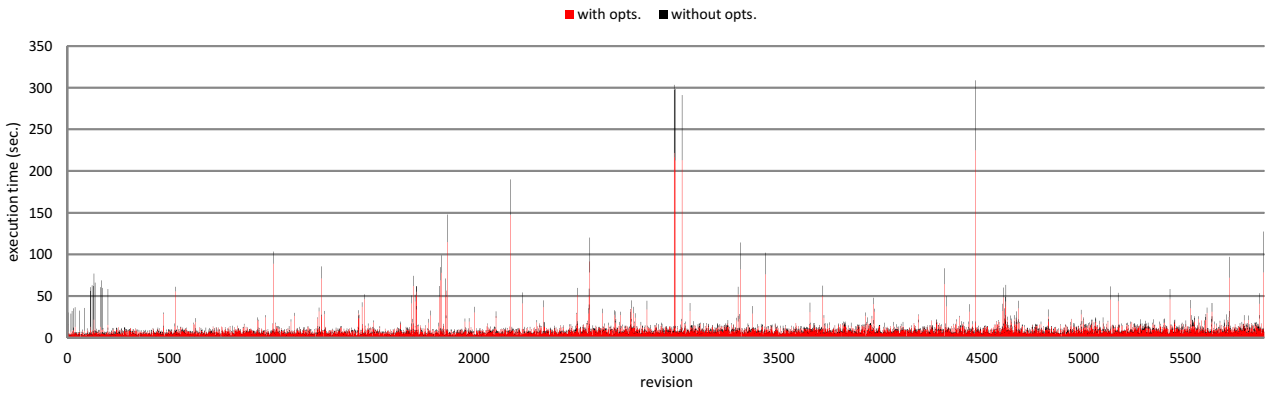



Figure 6. Elapsed time to detection for every revision in Ant

clones actually has the negative impact on software maintenance or not [8], [16], [18]. By using incremental code clone detection, such analyses can be conducted quite efficiently.

- **Context 2**: When a bug is found in a code fragment, it is desirable to modify its related bugs in other code fragments simultaneously. In this context, code clone detection is one of the promising approach for simultaneous modifications.

*1) Evaluation on Context 1:* This evaluation was conducted on the following steps.

- **STEP1**: checkout the initial revision.
- **STEP2**: detect code clones from the initial revision. Note that the analysis processing is performed against all the source files of the initial revision.
- **STEP3**: checkout the next revision.
- **STEP4**: detect code clones from the revision. Note that the analysis processing is performed against only the source files that are added or modified on the revision.
- **STEP5**: go to STEP3 if the next version exists.

We investigated the total time of STEP2 and STEP4. Table I shows the result. The prototype could finished detection within a single day from more than 5,000 revisions. The optimizations shortened detection time by a few hours.

Figure 6 shows time to detecting code clones from every revision. Red and black bars shows time with and without the implementation optimizations respectively. A red bar is drawn in front of a black bar in every revision, which means that a black bar does not appear if the time without the optimizations is shorter than the time with the optimizations. This figure shows that the time without the optimizations is a little longer then the time with the optimization in most revisions. On the other hand, we observed the opposite happens in some revisions. In such revisions, difference in time was quite small, less than 100 milliseconds. Such phenomenon deems to result from a chance. We can see that the detection took less than a minute in most revisions with optimizations. However, there are 19 revisions took over 1 minutes. All of them have many updated files. The average, medium, and largest number of updated files in a single revision were 3.53, 1, and 733, respectively.

For comparison, we investigated the detection time of PDG-based non-incremental code clone detection. However, we could not finish non-incremental detection because it took more than one day to detect code clones from the first 300 revisions. The proposed method makes it possible to detect PDG-based code clones for historical analysis from actual software systems.

*2) Evaluation on Context 2:* This evaluation was conducted on the following steps:

- **STEP1**: the analysis processing is performed against all the source files,
- **STEP2**: the detection processing is performed against every of the source files.

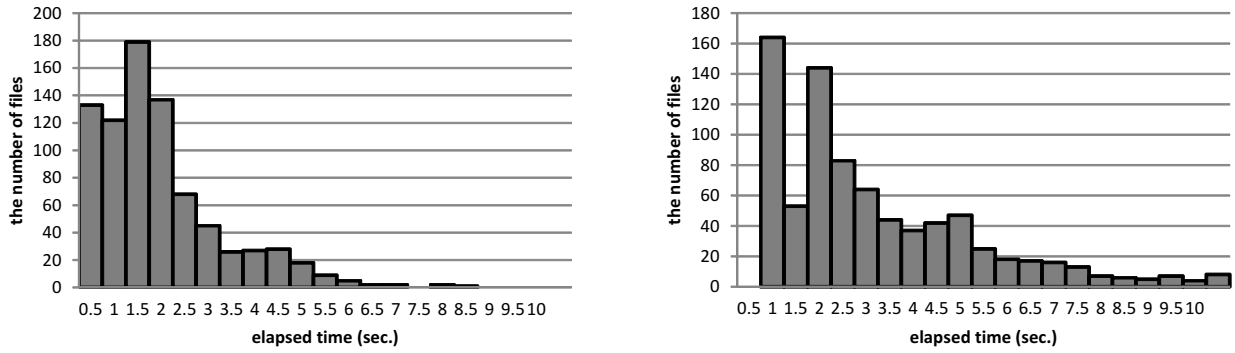
STEP1 is performed only once at the beginning of this evaluation. Table II shows the time of STEP1 and STEP2.

Table I
DETECTION TIME IN CONTEXT 1

| # of revisions | Detection Time | |
|---|---|---|
| | with opts. | without opts. |
| 5,903 | 13 hrs. 11 mins. | 15 hrs. 2 mins |

Table II
DETECTION TIME IN CONTEXT 2

| Processing | | execution time | |
|---|---|---|---|
| | | with opts. | without opts. |
| STEP1 | | 3 mins. 18 secs | |
| STEP2 | ave. | 1.7 secs. | 2.9 secs. |
| | med. | 1.4 secs. | 2.2 secs. |
| | max. | 8.0 secs. | 12.8 secs. |




(a) with opts.

(b) without opts.

Figure 7. Elapsed time to detection for every file in Ant

the prototype took about 3 minutes for the initial analysis processing, which is for preparing database used in the detection process. In the cases of most source files, detection processing took only several seconds, which shows that the maintainer can instantly obtain code clones after specifying a source file.

Figure 7 shows histograms of detection time for every source file on STEP2 with and without the optimizations. Every bar of the histograms means the number of files where detection time is between $500n$ and $500(n+1)$ milliseconds ($n = 0, 1, 2, \cdots$). Table II shows average, median, and maximum time for both the detections. Detection processing finished within a few second in most cases, and maximum time was 8.0 and 12.8 seconds, which means that we can obtain code clone information instantly in Context 2. When the optimizations were not used, there was no file within 500 milliseconds meanwhile 133 files appeared within the same region with the optimization. Also, all the average, median, and maximum time shortened, which means that the optimizations are effective to reduce detection time.

### B. Experiment 2: Detection Quality

We compared code clones detected by the proposed method with ones detected by an existing PDG-based detection tool, Scorpio [7]. There are two reasons why we chose Scorpio as the comparison target. One is that Scorpio counts execution-next link of PDG as well as the proposed method. Consequently, comparing the proposed method with Scorpio can reveal how the difference between similar-graph-based detection and edge-based detection has an impact on code clone detection on actual source code. The other is that Scorpio has been developed in our research group. Consequently, we are well-acquainted with how to use Scorpio. Note that this experiment is not for evaluating whether code clones detected by the proposed method are worth to be checked by human but for evaluating differences between PDG-based and edge-based detections.

We calculated precision and recall for conducting a quantitative evaluation. In this evaluation, code clone matching between the proposed method and Scorpio was performed with *good* and *ok* measures, which were proposed by Bellon et al. [3]. We specified 0.7 as the threshold, which is the same value used in the literature [3].

Table III shows the number of clone pairs detected by Scorpio and the proposed method. Table IV shows precision and recall. Both the recall and precision are almost 1 with and without the optimizations. The optimizations do not decrease the detection ability of the proposed method as a PDG-based detection tool. This evaluation shows that the proposed method detects a quite similar set of clone pairs to PDG-based non-incremental detection tool.

Figure 8 shows a pair of unmatched clone pairs. The lines with prefix "+" indicate that they are included in a clone pair detected by the proposed method, and the lines with "–" mean that they are included in a clone pair detected by Scorpio. For example, the 64th line is included in both the instances of a clone pair detected by the proposed method,

Table III
NUMBER OF DETECTED CLONE PAIRS

| Scorpio | Proposed Method | |
|---|---|---|
| | with opts. | without opts. |
| 724 | 740 | 831 |

Table IV
PRECISION AND RECALL

| measure | *good* | | *ok* | |
|---|---|---|---|---|
| | with opts. | without opts. | with opts. | without opts. |
| precision | 0.962 | 0.922 | 0.995 | 0.970 |
| recall | 0.972 | 0.990 | 0.994 | 1.000 |

```
    50:  e = replyToList.elements();
    51:  while (e.hasMoreElements()) {
    52:    mailMessage.replyto(
                    e.nextElement().toString());
    53:  }
+   54:  e = toList.elements();
+   55:  while (e.hasMoreElements()) {
+   56:    String to = e.nextElement().toString();
    57:    try {
    58:      mailMessage.to(to);
+   59:      atLeastOneRcptReached = true;
+   60:    } catch (IOException ex) {
+   61:      badRecipient(to, ex);
    62:    }
    63:  }
++- 64:  e = ccList.elements();
++- 65:  while (e.hasMoreElements()) {
++- 66:    String to = e.nextElement().toString();
    67:    try {
    68:      mailMessage.cc(to);
++- 69:      atLeastOneRcptReached = true;
 +- 70:    } catch (IOException ex) {
 +- 71:      badRecipient(to, ex);
    72:    }
    73:  }
 + -74:  e = bccList.elements();
 + -75:  while (e.hasMoreElements()) {
 + -76:    String to = e.nextElement().toString();
    77:    try {
    78:      mailMessage.bcc(to);
 + -79:      atLeastOneRcptReached = true;
   -80:    } catch (IOException ex) {
   -81:      badRecipient(to, ex);
    82:    }
    83:  }
```

Figure 8. Unmatched Clone Pairs Detected from Ant

and it also included in an instance of a clone pair detected by Scorpio. The 64th, 65th, 66th, and 69th lines are included in both the instances detected by the proposed method. On the PDG, the code clones do not share the same edge, however they shares the same node. The definition of clone pair in the proposed method do not prohibit both code clones in a clone pair from sharing the same node. On the other hand, Scorpio does not permit code clones to share the same node. Such a difference yielded this unmatched clone pairs. however, in this experiment, we cannot say which clone pair is better to be detected because what kinds of code clones are different from what for we detect code clones.

## VI. Related Work

Hummel et al. proposed a line-based incremental detection methodology [9]. Their method firstly replaces user-defined identifiers with special tokens in every line of the source code. Then, hash values are calculated from them. Next, the method stores their hash values, their line numbers, and their files names into the database. By using the database, lines that are duplicated with specified lines can be instantly obtained. multiple-lines duplication can be easily constructed by combining single-line duplication stored in the database.

Göde and Koschke proposed a token-based incremental detection methodology [6]. They proposed a generalized suffix tree, which is suitable to node insertion and deletion. Suffix tree is a tree-structure where the number of suffixes is the same as the number of leafs on a specified string. A path from the root to a leaf $i$ means the suffix started from $i$-th of the specified string. It is possible to detect repeated substrings within a specified string by using suffix tree. In code clone detection, a suffix tree is generated from the entire of a software system [1], [12]. A generalized suffix tree is built on not a string but a set of strings, and paths from the root to leafs represent suffixes of the strings. In their method, every source file is a string in a generalized suffix tree, which makes it easy to add or delete source files. Consequently, the generalizes suffix tree is suitable to incremental code clone detection.

Jiang et al. proposed a AST-based code clone detection methodology [11]. They defined *characteristic vector*, which is a vector representation of subtrees in ASTs. Elements of *characteristic vector* are the number of various kinds of tokens (e.g., variable names, literals, preserved names). *Characteristic vectors* are compared by LSH algorithm [5]. LSH algorithm detects similar vectors with ignoring small differences between them. By applying LSH algorithm to *characteristic vectors*, similar subtrees in ASTs are identified. Also, Lee et al. proposed a multidimensional indexing methodology for quick code clone detection with *characteristic vectors* [17]. Literatures [11] and [17] do not explicitly describe about incremental code clone detection with *characteristic vectors*, however *characteristic vectors* can be a intermediate representation of incremental detection.

Jia et al. developed a detection tool, KClone, which is an implementation of the hybrid approach of token-based and PDG-based techniques [10]. Firstly, KClone detects contiguous code clones with token-based techniques, then it uses data and control dependencies from/to the contiguous code clones for enlarging them to non-contiguous ones. KClone's detection requires only lightweight program analyses. Actually, in the experiement of their paper, KClone detected code clones more rapidly than CCFinderX. The differences of the proposed method and KClone are as follows.

- KClone detects contiguous code clones in the first step of detection process. That is, if a code clone does not have a contiguous part that is longer than a threshold, it is not detected. On the other hand, the proposed method does not require contiguous parts for code clones.
- KClone is not an incrmental detection. Consequently, in both the Context 1 and 2 of the experiment, detection speed of the proposed method is more rapid than KClone, which will be especially prominent in large-scale software.

## VII. Conclusion

This paper proposed a PDG-based incremental code clone methodology, and introduced a prototype tool developed based on the proposed method. The prototype tool has optimizations for obtaining code clones more efficiently. A case study was conducted with the prototype tool on open source software systems. The experiment showed that the proposed method could obtain code clones within a short timeframe and its detection result was quite similar to the detection result of an existing PDG-based detection tool.

In the future, we are going to enhance the proposed method for more shortening detection time. At present, the proposed method updates the database by a source file. However, a single source file includes multiple methods and a part of them is modified in a revision. Consequently, database by a method is more efficiently for incremental code clone detection.

## 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. This study has been supported in part by Grants-in-Aid for Scientific Research (A) (21240002) and Grant-in-Aid for Exploratory Research (23650014) from the Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (B) (22700031) from Ministry of Education, Science, Sports and Culture.

## References

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

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

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

[4] 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*, pages 36–43, Oct. 2002.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proc. of the 20th Symposium on Computational Geometry*, pages 253–262, June 2004.

[6] N. Göde and R. Koschke. Incremental Clone Detection. In *Proc. of the 13th European Conference on Software Maintenace and Reengineering*, Mar. 2009.

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

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

[9] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proc. of the 26th IEEE International Conference on Software Maintenance*, pages 1–9, Sep. 2010.

[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] 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*, pages 96–105, May 2007.

[12] 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*, 28(7):654–670, July 2002.

[13] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, Sep. 2005.

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

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

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

[17] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim. Instant Code Clone Search. In *Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 167–176, Nov. 2010.

[18] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. of the 24th International Conference on Software Maintenance*, pages 227–236, Sep. 2008.