Scientific Research Publishing

Journal of Software Engineering and Applications, 2017, 10, 891-906
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Code Clone Detection Method Based on the Combination of Tree-Based and Token-Based Methods

Ryota Ami, Hirohide Haga

Graduate School of Science and Engineering, Doshisha University, Kyoto, Japan
Email: rami@ishss10.doshisha,ac.jp, hhaga@mail.doshisha.ac.jp

How to cite this paper: Ami, R. and Haga, H. (2017) Code Clone Detection Method Based on the Combination of Tree-Based and Token-Based Methods. *Journal of Software Engineering and Applications*, **10**, 891-906.
https://doi.org/10.4236/jsea.2017.1013051

Received: November 23, 2017
Accepted: December 25, 2017
Published: December 28, 2017

Abstract

This article proposes the high-speed and high-accuracy code clone detection method based on the combination of tree-based and token-based methods. Existence of duplicated program codes, called code clone, is one of the main factors that reduces the quality and maintainability of software. If one code fragment contains faults (bugs) and they are copied and modified to other locations, it is necessary to correct all of them. But it is not easy to find all code clones in large and complex software. Much research efforts have been done for code clone detection. There are mainly two methods for code clone detection. One is token-based and the other is tree-based method. Token-based method is fast and requires less resources. However it cannot detect all kinds of code clones. Tree-based method can detect all kinds of code clones, but it is slow and requires much computing resources. In this paper combination of these two methods was proposed to improve the efficiency and accuracy of detecting code clones. Firstly some candidates of code clones will be extracted by token-based method that is fast and lightweight. Then selected candidates will be checked more precisely by using tree-based method that can find all kinds of code clones. The prototype system was developed. This system accepts source code and tokenizes it in the first step. Then token-based method is applied to this token sequence to find candidates of code clones. After extracting several candidates, selected source codes will be converted into abstract syntax tree (AST) for applying tree-based method. Some sample source codes were used to evaluate the proposed method. This evaluation proved the improvement of efficiency and precision of code clones detecting.

Keywords

Code Clone, Token-Based Detection, Tree-Based Detection, Tree Edit Distance

1. Introduction

This article proposes the high-speed and high-accuracy code clone detecting method. Code clone is a fragment of source code that is identical or similar to other portion of source code [1]. Code clones often reduce the maintainability of software. Suppose that there are two code fragments *A* and *B*, and fragment *B* is a clone of fragment *A*. If errors (bugs) are included in *A*, *B* must contain same errors and they must be removed at the same time when errors in *A* are removed. If the programmer does not recognize the existence of clone *B*, he or she may forget to revise them in clone *B*. This may cause the deterioration of software quality. It is often said that 10% to 20% of codes are duplicated code (code clones) in large-scale software [2] [3]. If the software is large, finding all code clones are hard work.

From syntactical point of view, there are three types of code clone named **TYPE-1**, **TYPE-2**, and **TYPE-3** [4]. In **TYPE-1**, all parts of original and clones are identical. In **TYPE-2**, some differences such as the difference of identifier name and function name, and the value of constants exist but structure of code is identical. In **TYPE-3**, several statements are inserted or removed from original source code.

In order to detect these code clones, several methods are proposed in previous works. These methods are based on two principles; one is token-based method [5] [6] and the other is tree-based method [7] [8]. Token-based methods can detect **TYPE-1** and **TYPE-2** clones very fast but hard to detect **TYPE-3** clones. This method is relatively fast and can be applied to large-scale software. On the other hand, tree-based methods can detect all types of code clone but require large computing resources (CPU time and memory).

In this article, we propose the new method that can detect all types of code clone and run relatively fast. Our method combines token-based method and tree-based method. By using token-based method that runs fast, some candidates of code clones are extracted. After the extraction of candidates, each candidate fragment is examined by using tree-based method if it is clone or not. By combining token-based method and tree-based method, our method can detect all types of code clone faster.

2. Definition of Code Clone

2.1. Definitions and Types of Code Clones

Figure 1 is a conceptual illustration of code clone. Let us consider one sample source code shown in Figure 1. The code fragment shown at the left-hand side is the original code. There are another two code fragments at the right-hand side of the source code illustrated by rectangles. These two fragments are assumed to be identical or similar to the original. Then these two fragments are code clones. Note that there are only two fragments in Figure 1, three or more code clones may exist in the source code. Suppose there is one bug in the original code shown by the red star in Figure 1. Then similar bugs may exist in another code

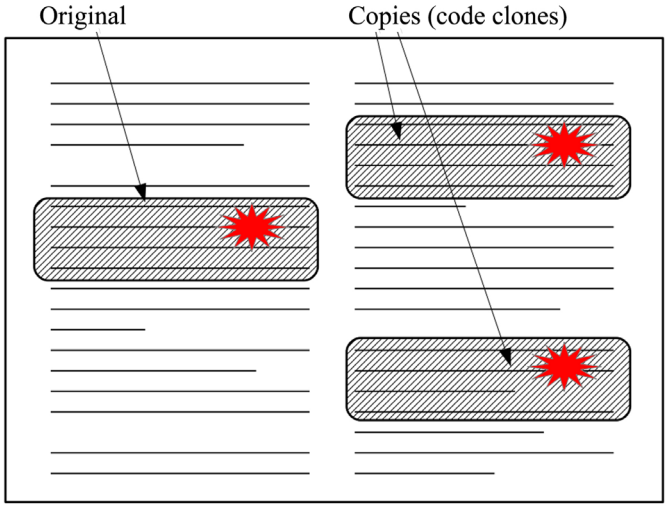


Figure 1. Conceptual illustration of code clone.

fragments of code clones. In order to remove all bugs in the source code, programmer has to find all bugs in other code clones. If he or she forgets to revise other bugs, the quality of software will reduce.

As the size of program increases, the number of code clones increases. Code clones are usually brought into original code by copy-and-paste operation. Finding all code clones in large complex program becomes hard. Therefore, detecting code clones and improving the structure of original code play an important role in software quality assurance.

Bellon *et al.* classified code clones into three types based on the features of clones [4]. They are:

- **Type-1 (Exact clone)**: Exact duplication of original part except white space, tab, carriage code and other coding style related characters.
- **Type-2 (Parameterized clone)**: Syntactically identical but some names of identifiers (variable names, function/method names etc.) and the values of constants are different in two code fragments.
- **Type-3 (Gap clone)**: Duplication with some insertion and/or deletion of statements.

2.2. Related Works of Clone Detection

Several methods to detect code clones are proposed in previous works. For example, Baker *et al* tried to detect clones by line-wise comparison of two files [2]. Furthermore, they introduced the parse tree and used it for detecting clones [2]. Currently, detecting methods are classified into three categories.

- **Text-based method (or line-based method)**: This method detects code clones by comparing two codes fragments line by line. This method runs fast and lightweight. But this method can detect only Type-1 clones. Some sophisticated method can detect Type-2 clones. This method is the fastest.
- **Token-based method:** This method detects code clones by comparing two sequences of tokens (minimum unit of lexically meaningful sequence of cha-

racters). As tokens only represent the kind of elements in programming language, this method can detect Type-1 and Type-2 clones. But it requires sophisticated modification to detect Type-3 clones. This method is relatively faster and lightweight.

- **Tree-based method**: This method detects code clones by comparing two abstract syntax tree (AST) [9] or other tree representation of source codes. In this method, a source code is tokenized and parsed firstly. Source code is converted into AST (or other tree-based representation) and compared the similarity of two (or more) trees. This method requires much computing resources (slow and large memory) but detect all types of clones.

Table 1 is a result of the comparison of previous methods.

2.3. Issues of Previous Works

As shown in Table 1, all previous methods have some advantages and disadvantages. Text-based and token-based methods can run relatively fast and do not require much computing resources. For example, Kamiya *et al.* reported that 40 seconds are required to find clones in 235,000 lines source code [5]. Baker showed only 12 seconds were necessary to the same source code [10]. But finding Type-3 clones is difficult by these two methods.

Our preliminaries investigation on some sample programs found that the distribution of the clone types is shown in Figure 2. This graph shows that approximately 97% of clones are Type-2 and Type-3, and Type-3 occupies about 40%. Therefore, ignoring Type-3 clones makes the quality of software worse. As

Table 1. Comparison of previous methods of code clone detection.

	Type-1	Type-2	Type-3	Speed
Text-based	◦	Δ	×	Fast
Token-based	◦	◦	×	Medium
Tree-based	◦	◦	◦	Slow

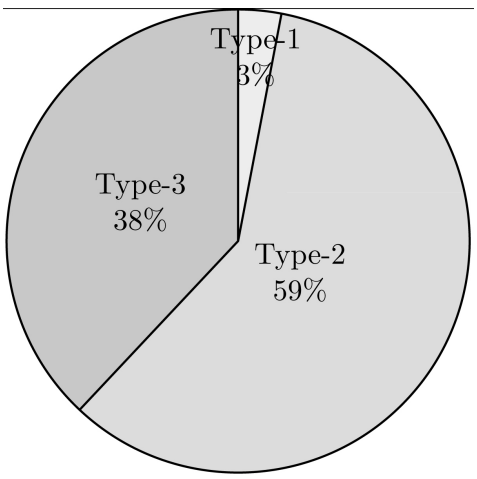


Figure 2. Ratio of code clone type.

you can easily imagine, detecting Type-1 and Type-2 is not so difficult and detecting Type-3 is hard.

With this investigation, we can also see that the number of exact clones is relatively small. This make sense since in most of the cases of code clone, we take a part of the code and change it to fit new need of the function which is near the cloned program in terms of service or computation to provide. In the process of modification, some identifiers may be renamed, some statements are inserted or removed, and some conditional expression may be changed. Therefore, difficulty of finding Type-3 clones is a serious drawback from the practical point of view.

On the other hand, tree-based methods can detect almost all clones including Type-3 clones that are hard to be detected by text or token-based method. But tree-based methods require much computing time and resources. Baxter [7] and Krinke [11] indicated that about 3 hours and 63 hours were necessary to find all code clones in approximately 115,000 lines codes. The reason why tree-based method requires much computing resources is basically the comparison time of two or more trees. When the number of nodes in the tree T is N, the computational time complexity of naive tree comparison is proportion to N^6 [11]. Therefore, when the number of nodes, which corresponds to the size of software, becomes 10 times larger, the computation time becomes 1 million times longer than original one. It means that the tree-based method is hard to apply to large-scale software.

3. Proposed Method

3.1. Overall of Proposed Method

Based on the above consideration, we propose our new code clone detection method that is based on the combination of token-based method and tree-based method. Token-based method runs faster but is not appropriate for detecting Type-3 clones. Tree-based method can find all types of clones but runs slower. The reason of large amount of computing time of tree-based method is its comparison time of trees. The larger the software becomes, the more the number of compared trees. Therefore, we use the token-based method to narrow down the number of trees to be compared. By reducing the number of trees to be compared, we can execute the tree-based method much faster. **Figure 3** is the overall steps of proposed method. Proposed method consists of following steps.

1) Lexical analyzing source code and generate token sequence,

2) Applying token-based method to extract the candidates of code clones,

3) Generating abstract syntax trees (ASTs) of code clone candidates,

4) Comparing ASTs to fix code clones of all types.

In step (1), source code is converted into the sequence of tokens. Then some conversion will be done to the sequence of tokens to detect TYPE-2 clones. This conversion includes the replacement of specific tokens such as identifier and function/method name by special characters. **Figure 4** is an example of conversion. In this example, all identifiers are replace by special character "$" and

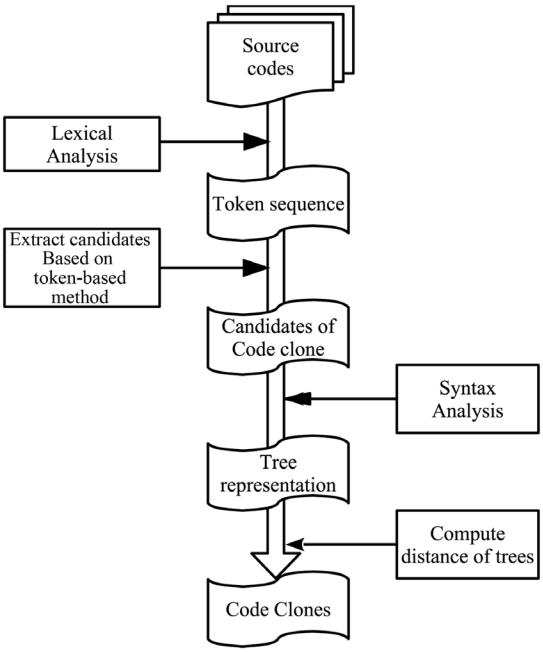


Figure 3. Overall steps of proposed method.

constants are replaced by "#". After the conversion, detection process will be executed. After arranging all tokens as shown in **Figure 5**, check the identical token and mark as "*". Some code fragments, which have sufficient length of diagonal lines of **Figure 5**, are candidates of code clone. After the extraction of code clone candidates in step (2), these code fragments are converted into abstract syntax trees in step (3). Then these trees are compared to find Type-3 code clones by using tree distance in step (4).

3.2. Gap of Diagonal Line

Type-1 and Type-2 code clones draw the continuous diagonal lines when we represent them in a manner shown in **Figure 5**. But as Type-3 code clones have some insertions and/or deletions of statements, there are several gaps within the clone diagonal lines. **Figure 6** shows the example of this gap diagonal line. In this case, original source code has several inserted and/or removed tokens. This will cause a gap as shown in **Figure 6**.

In order to detect Type-3 code clones, we have to detect the candidates with gap in a diagonal line. In order to detect this gap, we need to find a method to look for other part of code that might follow this gap.

Figure 7 shows an illustration of gap diagnose lines. In this case, there are three diagonal lines l_i, l_j, and l_k. To find Type-3 code clones by merging several code clone fragments, we will use the following algorithms.

Algorithm

Step 1 Extracting diagonal lines (code clone candidates) with longer than predefined length.

Step 2 Let l_i be a diagonal line whose starting point is $\left(l_i^s(x), l_i^s(y)\right)$ and

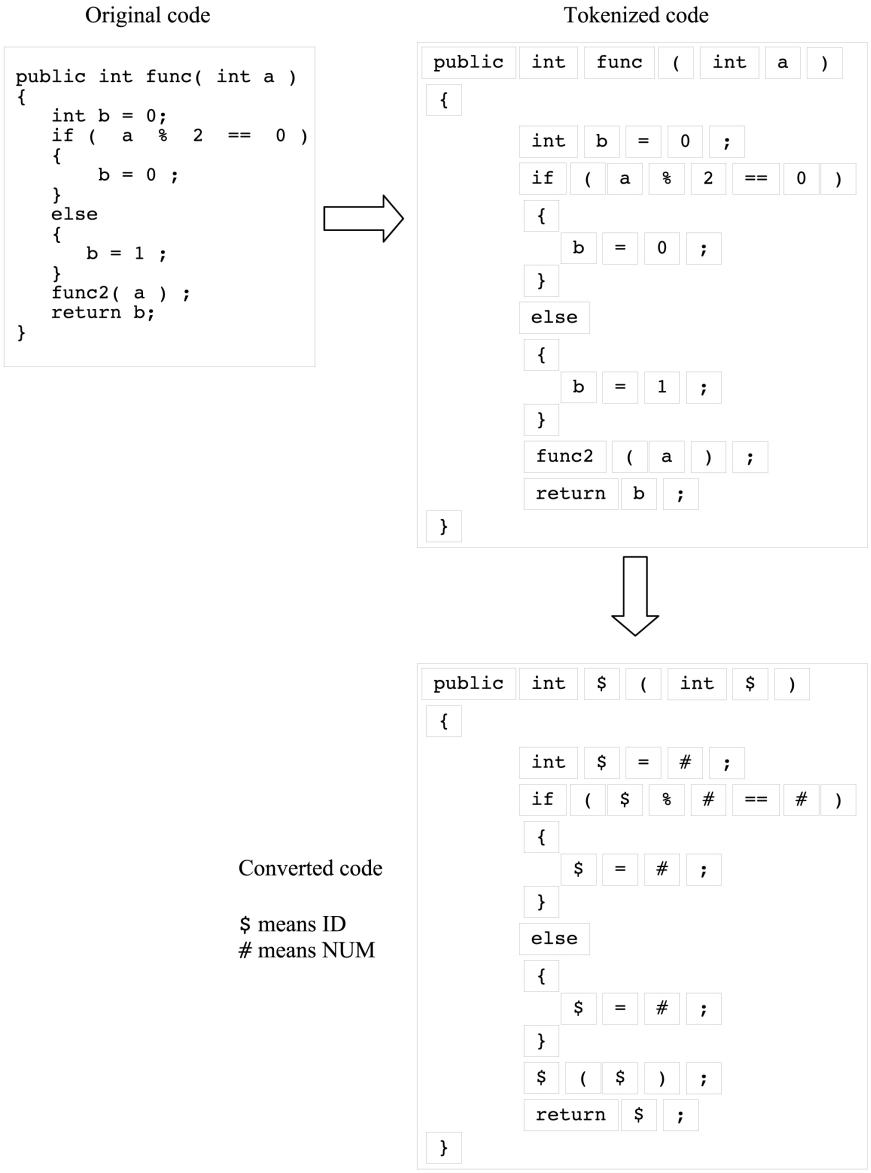


Figure 4. Example of conversion.

ending points is $\left(l_i^e(x), l_i^e(y)\right)$.

Step 3 For all diagonal lines, if there is another diagonal line l_j where $\left|l_i^e(x) - l_j^s(x)\right| < dx$ and $\left|l_i^e(y) - l_j^s(y)\right| < dy$, merge two lines l_i and l_j and draw a diagonal line starting at l_i^s and ending at l_j^e. Note that dx and dy are predefined tolerable limit. The fragment corresponding this new diagonal line is a merged *virtual* diagonal line and will be a candidate of Type-3 code clone.

Step 4 When there are two or more lines $l_{j_1}, l_{j_2}, \cdots, l_{jm}, \cdots, l_{jk}$ whose starting points $l_{j_m}^s$ satisfies $\left|l_i^e(x) - l_{j_m}^s(x)\right| < dx$ and $\left|l_i^e(y) - l_{j_m}^s(y)\right| < dy$, draw a line from l_i^s to l_v^e where l_v^e is an ending point of virtual line l_v where $i_v^e(x) = \max\left\{l_{j_1}(x), l_{j_2}(x), \cdots, l_{jk}(x)\right\}$ and $l_v^e(y) = \max\left\{l_{j1}(y), l_{j2}(y), \cdots, l_{jk}(y)\right\}$. The code fragment corresponding to the

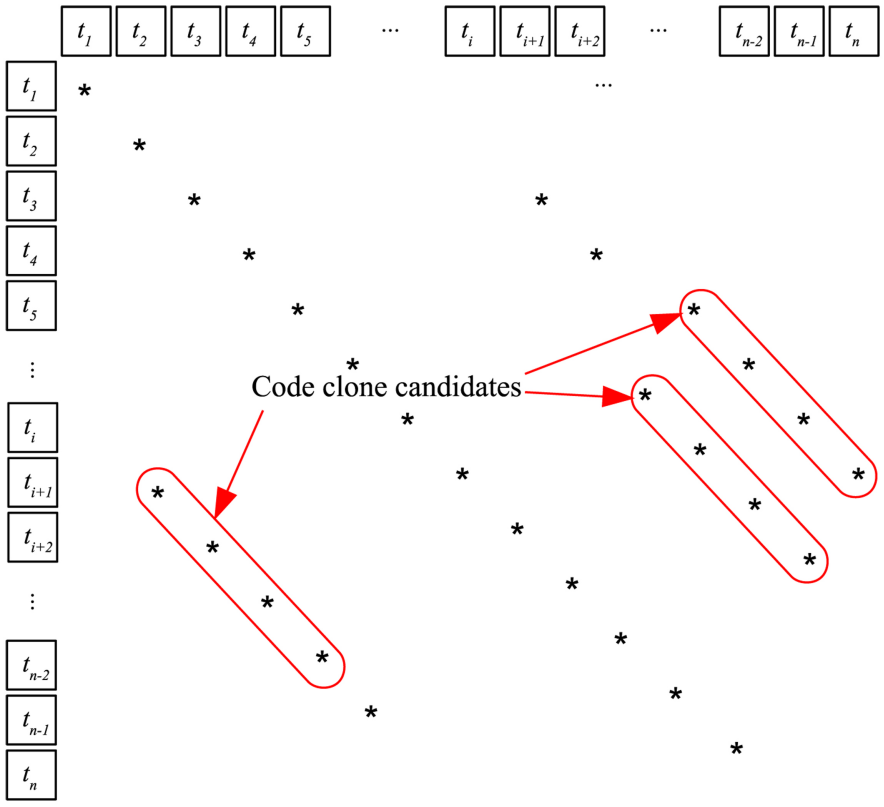


Figure 5. Arranging token streams to find code clones.

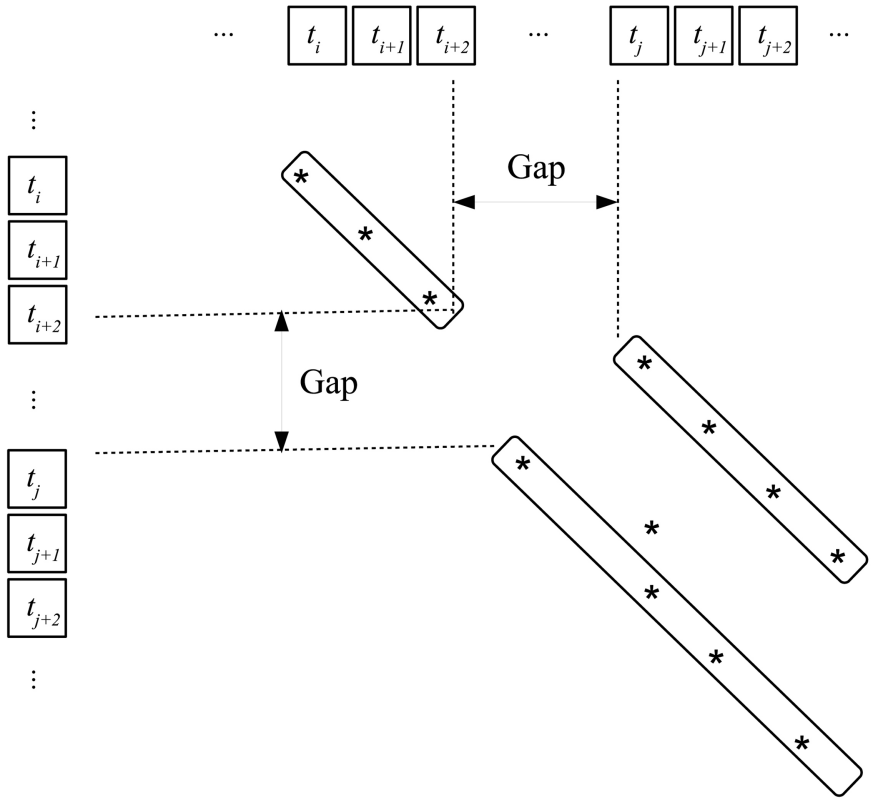


Figure 6. Example of gapped diagonal line.

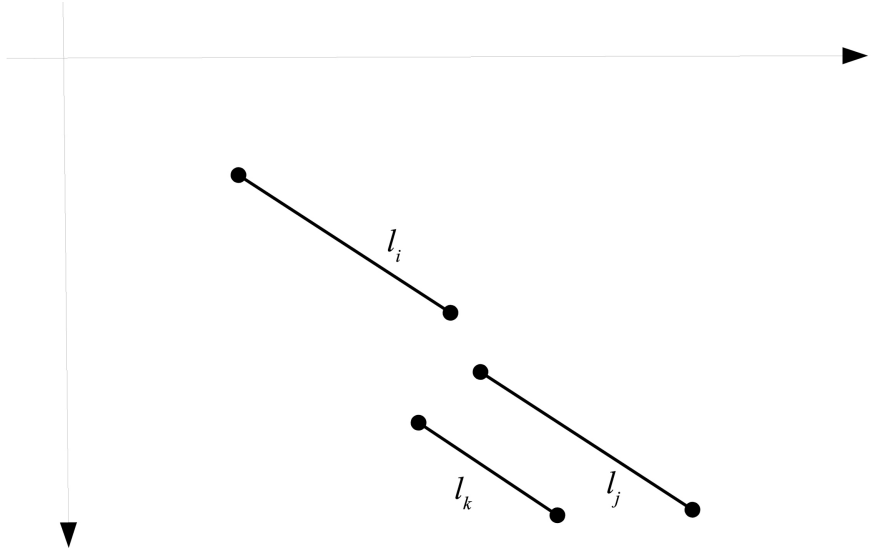


Figure 7. Illustration of gapped diagonal lines.

diagonal line l_i^s to l_v^e is a candidate of Type-3 code clone. Figure 8 illustrates the gap code clone that has virtual diagonal line. This virtual diagnose line represents the candidate of TYPE-3 code clone.

3.3. Applying Tree-Based Method

Exact and parameterized code clones (Type-1 and Type-2) can be found by token-based method. But gap clone (Type-3) cannot be identified by token-based method. In order to identify Type-3 code clone, we need further steps. They are 1) translating source code into abstract syntax tree (AST) or similar tree-based representation method for source code, 2) compute the difference (distance) of any pair of code clone candidates, and 3) identify Type-3 code clone by examination of distance. Transforming source code into AST is a well-known processing of language processor such as compiler. We will not mention this process any more.

In order to compute the distance of two trees, we use the tree edit distance (TED) [12]. In TED, several primitive operations are used to modify trees. For example, we can assume that following three operations are primitive operations for tree transformation: 1) insertion of node, 2) deletion of node, and 3) renaming of node. Let T_1 and T_2 be two trees. By applying above-mentioned three primitive operations for tree modification, we can always transform T_1 into T_2. The trivial method of transforming T_1 into T_2 is that a) delete all nodes in T_1 and generate void (null) tree, b) insert all nodes of T_2 into void tree. Let n_1 and n_2 be the number of nodes in T_1 and T_2 respectively. Then maximum TED of T_1 and T_2 is n_1+n_2 (deleting all nodes of T_1 requires n_1 operations and inserting all nodes of T_2 requires n_2 operations). There may be another shorter sequence of operations that transform T_1 into T_2. TED is defined as *the minimum number of these primitive operators*. For example, let us consider two trees T_A and T_B in Figure 9. The number of nodes in T_A is 6

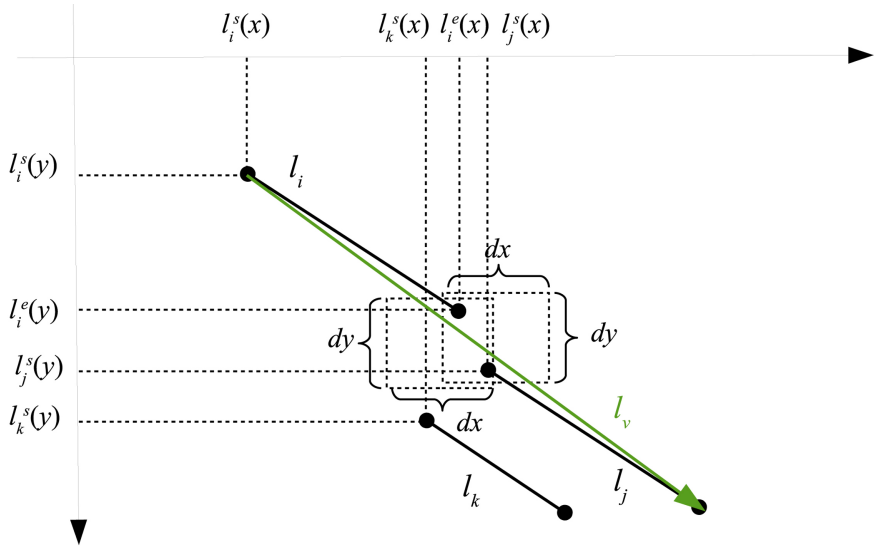


Figure 8. Virtual diagonal line.

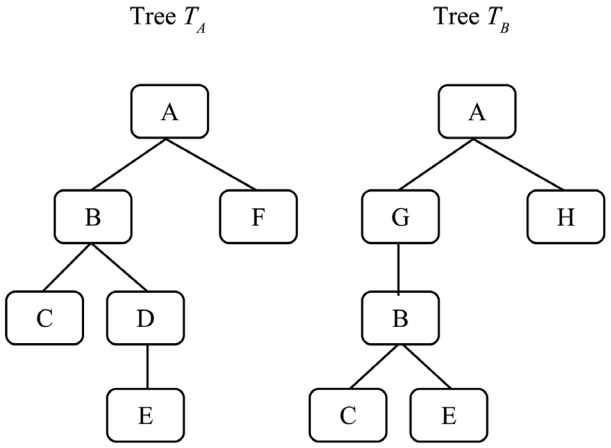


Figure 9. Tree transformation.

and that of T_B is 6. Therefore by using trivial method, T_A can be converted into T_B by 6 + 6 = 12 operations. This is the maximum length of transformation operations. However, T_A can be converted by the following sequence: i) delete node "D" from tree T_A, ii) insert node "G" between node "A" and "B" of tree T_B, iii) rename node "F" of tree T_A into "H". Total number of primitive operation is 3 and this is the minimum number of operations for transforming T_A into T_B. Therefore TED of T_A and T_B is 3[1].

By applying TED and computing the distance of any two trees based on the established algorithms [13], we can filter the Type-3 code clones.

4. Prototype Implementation

4.1. Overall of Prototype

We have implemented the code clone detection system which adopted proposed

[1]If strictly speaking, we have to prove that 3 is the *minimum number* of the sequence of transformation.

method. Following is a target program of experiment.

- Implementation Language: Java
- Program name: JDK 1.5.0
- The number of files: 108
- The number of lines: 33,128 lines

Proposed system is implemented under following environment.

- OS: Window 7 Professional 64 bits
- CPU: Intel Core i7 3.2 GHz
- RAM: 6.0 GB (2.0 GB is used for executing proposed method)

Proposed system is implemented by Java. The total number of prototype code is 4333 lines.

4.2. Experimental Result

Figure 10 shows the result of token-based detection. In this experiment, 50 or longer sequence of tokens were selected as candidates of code clones. As test file contains more than 400,000 tokens, we divided all token sequence into 4000 tokens block. We can identify several clone candidates around the upper left part of this diagram. By using this diagram, we can extract Type-1 and Type-2 code clones easily.

After extracting Type-1 and Type-2 code clones, we furthermore try to extract the candidates of Type-3 (gap) clones. Threshold values of *dx* and *dy* mentioned in section 3.3 were set to 500. Result is shown in Figure 11. For example, line 121 to 210 of code B is a code clone of source code from line 113 to 202 of source code A.

4.3. Computing Time Comparison

Figure 12 shows the computing time of code clone detection. This graph shows

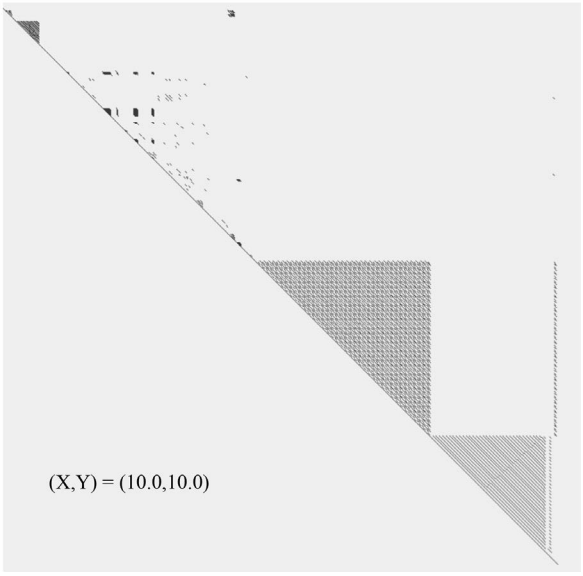
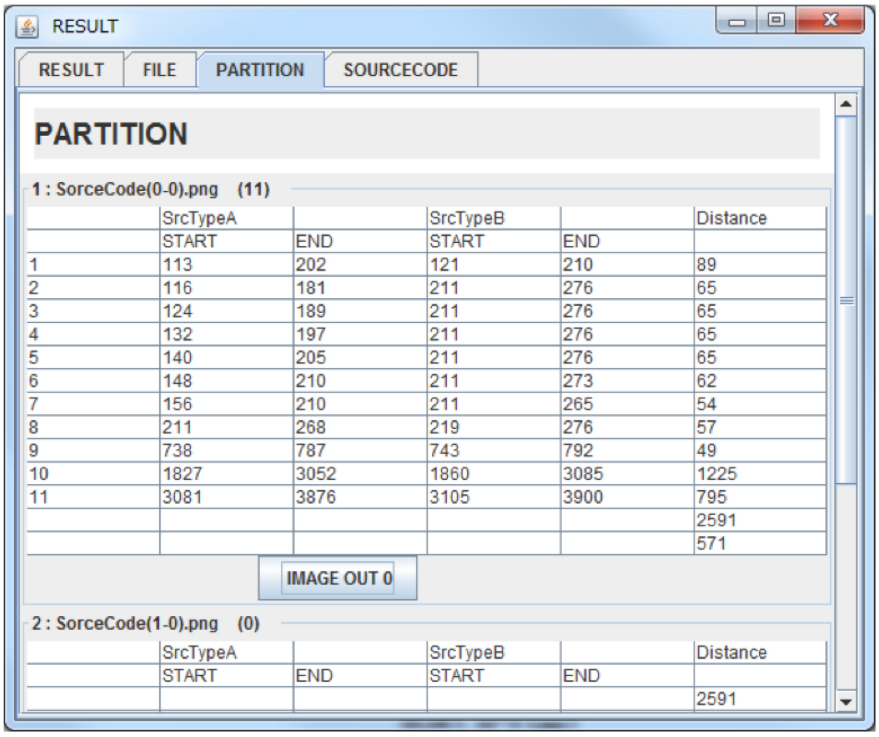


Figure 10. Result of token-based clone detection.



	SrcTypeA		SrcTypeB		Distance
	START	END	START	END	
1	113	202	121	210	89
2	116	181	211	276	65
3	124	189	211	276	65
4	132	197	211	276	65
5	140	205	211	276	65
6	148	210	211	273	62
7	156	210	211	265	54
8	211	268	219	276	57
9	738	787	743	792	49
10	1827	3052	1860	3085	1225
11	3081	3876	3105	3900	795
					2591
					571

	SrcTypeA		SrcTypeB		Distance
	START	END	START	END	
					2591

Figure 11. Result of the detection of gap code clone.

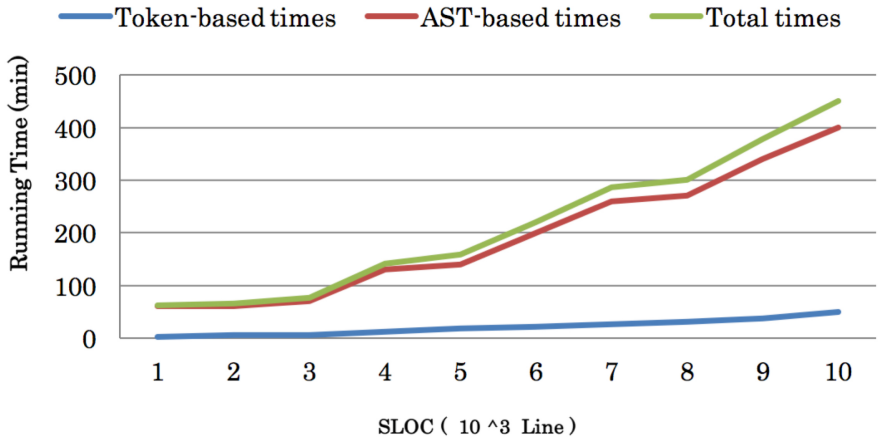


Figure 12. Computing time of three methods.

that the total computing time is heavily influenced by AST-based time. Approximately AST-based computing time occupies 80% to 90% of total computing time.

Let n be the number of tokens. Then complexity of token-based time is $O(n^2)$. And let m be the number of nodes in the tree. Then complexity of tree-based time is $O(m^4)$ [15]. Therefore the complexity of total computing time is $O(n^2+m^4)$. Table 2 shows the comparison result of proposed system and other systems.

CCFinder is the fastest system. But it uses token-based method, therefore it is not easy to detect Type-3 code clone. Other two systems and the proposed system

Table 2. Comparison of computing time.

	CCFinder [5]	Proposed Method	DECKAR [14]	CloneDR [7]
TYPE	Token-based	Tree-based	Tree-based	Tree-based
CPU	-	3.2 GHz	3.2 GHz	2.0 GHz
Memory	-	2.0 GB	2.0 GB	1.0 GB
Time	40 sec	2174 sec	7200 sec	9000 sec
		(0.6 h)	(2.0 h)	(2.5 h)

adopt tree-based method. Therefore, they can detect all types of code clones. This table shows that proposed system is three to four times faster than other two tree-based systems (DECKARD and CloneDR). Note that DECKARD uses same CPU clocks and memory size, and those of CloneDR are lower and small. Therefore comparison to CloneDR is not accurate. Clone detection computing does not include so many disk accesses. The computing time mainly depends on the CPU time. The CPU time is roughly in proportion to clock frequency. Therefore if CloneDR runs on the same clock frequency of DECKARD and proposed system, the computing time is approximately 6000 sec[2]. The reason why proposed system is faster than other two tree-based systems is the difference of the number of tree comparison. Our system narrows the candidates of code clones by token-based method.

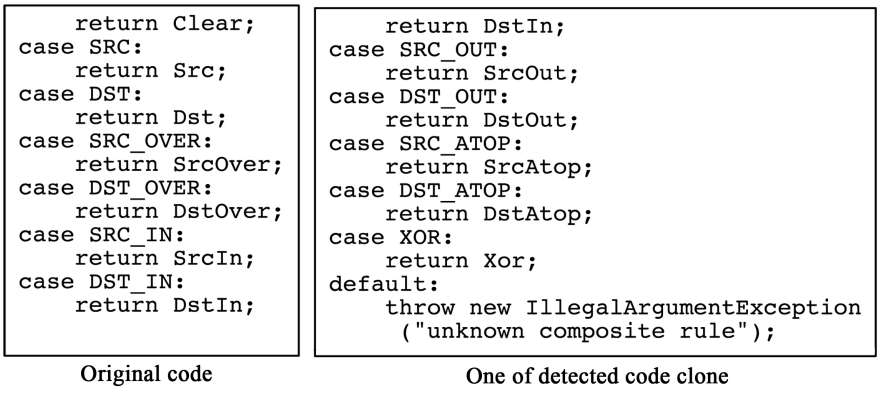
4.4. Quality of Proposed Method

Currently there is no standard criterion of the quality of code clone detection method. One of the simplest quality measures is the ratio of the number of detected code clones and the number of all code clones. But this measure is virtually useless because we cannot count all code clones in large-scale software. Bellon [16] proposed new assessment metric. Select several sample code clones from the set of detected clones and evaluate whether they are clones or not. By using this metric, we can assess the precision rate of detection. But we cannot assess the recall rate by using this metric[3]. However, in the detection of code clone, precision rate is more important than recall rate. Low precision rate means that the system generates many noisy code fragments that are not clones. This causes the waste of time to check if they are clones or not. On the other hand, even if undetected code clones remain in the source code, serious negative effect to the quality of software will not occur. Therefore, we use the Bellon's method for assessing the proposed method.

Figure 13 is an example of detected code clone. Left hand code contains 39 tokens and right hand code contains 46 tokens. The number of common tokens

[2] $9000 \times \frac{2.0}{3.2}$.

[3]The terms "precision rate" and "recall rate" are commonly used in the field of information retrieval and pattern recognition. Precision rate is the fraction of detected instance that are relevant, and recall rate is the fraction of relevant instance that are detected.



```
    return Clear;
case SRC:
    return Src;
case DST:
    return Dst;
case SRC_OVER:
    return SrcOver;
case DST_OVER:
    return DstOver;
case SRC_IN:
    return SrcIn;
case DST_IN:
    return DstIn;
```

Original code

```
    return DstIn;
case SRC_OUT:
    return SrcOut;
case DST_OUT:
    return DstOut;
case SRC_ATOP:
    return SrcAtop;
case DST_ATOP:
    return DstAtop;
case XOR:
    return Xor;
default:
    throw new IllegalArgumentException
     ("unknown composite rule");
```

One of detected code clone

Figure 13. Example of detected code clones.

in these two is 22 and the ratios of common tokens are 56% (left) and 48% (right). After converting these two codes into two ASTs, we compare the ratio of common nodes in ASTs. The number of nodes of left code is 77 and that of right code is 79. The number of common nodes of these two ASTs is 63. Therefore ratios of common nodes are 78% and 82% respectively. Higher common ratio (tokens and nodes) means the high similarity of two codes. Comparing the common tokens with common nodes, ratio of common nodes is higher than that of common tokens. This means that using tree-based method can detect code clones more precisely.

Total average ratio of common tokens in our experiment is 57% and that of common nodes is 78%. Bellon's experiment says average ratio of precision of previous works is approximately 64% [16]. Therefore, we can conclude that our proposed method has sufficient precision ratio.

5. Conclusions

In this article, we proposed the new method of code clone detection which can detect all types of code clones. Our method combines token-based method and tree-based method. The former can run faster but cannot detect all types of code clones, the latter can detect all types of code clones but requires large computing resources (memory and CPU time). Therefore, applying tree-based methods to large-scale software is virtually impossible. We combine these two methods; token-based method is used to extract the candidates of code clones. Extracted candidates are transformed into abstract syntax trees (ASTs) representation and tree-based method is applied to these trees. By narrowing the candidates of code clones and reducing the number of comparison operations, computing time is reduced to reasonable time.

Experimental evaluation is conducted using sample files with approximately 35,000 lines source code. Proposed method is 3 to 4 times faster than conventional tools such as DECKARD and CloneDR, all of which adopt tree-based method. Detection accuracy is assessed using Bellon's criterion. Proposed method keeps almost the same accuracy of conventional tools. Based on these evaluation

results, we can conclude that the proposed method keeps the accuracy of detection and runs faster than conventional tools and therefore is useful for the improvement of code clone detection of large-scale software.

References

[1] Rattan, D., Bhatia, R. and Singh, M. (2013) Software Clone Detection: A Systematic Review. *Information and Software Technology*, **55**, 1165-1199. https://doi.org/10.1016/j.infsof.2013.01.008

[2] Baker, B. (1995) On Finding Duplication and Near-Duplication in Large Software Systems. *Proceedings of the* 2*nd Working Conference on Reverse Engineering* (*WCRE* 1995), 86-95. https://doi.org/10.1109/WCRE.1995.514697

[3] Roy, C.K. and Cordy, J.R. (1995) An Empirical Study of Function Clones in Open Source Software Systems. *Proceedings of the* 15*th Working Conference on Reverse Engineering* (*WCRE* 2008), 81-90.

[4] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E. (2007) Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, **SE-33**, 577-591. https://doi.org/10.1109/TSE.2007.70725

[5] Kamiya, T., Kusumoto, S. and Inoue, K. (2002) CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, **SE-28**, 654-670. https://doi.org/10.1109/TSE.2002.1019480

[6] Li, Z., Lu, S., Myagmar, S. and Zhou, Y. (2006) CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, **SE-32**, 176-192. https://doi.org/10.1109/TSE.2006.28

[7] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L. (1998) Clone Detection Using Abstract Syntax Trees. *IEEE International Conference on Software Maintenance* (*ICSM* 1998), 368-377. https://doi.org/10.1109/ICSM.1998.738528

[8] Yang, W. (1991) Identifying Syntactic Differences between Two Programs. *Software Practice and Experience*, **21**, 739-755. https://doi.org/10.1002/spe.4380210706

[9] Anil Kumar, G., Reddy, C.R.K. and Govardhan, A. (2014) Software Code Clone Detection Using AST. *International Journal of P2P Network Trends and Technology*, **9**, 33-39.

[10] Baker, B. (1992) A Program for Identifying Duplicated Code. *Computing Science and Statistics*, **24**, 49-57.

[11] Krinke, J. (2001) Identifying Similar Code with Program Dependence Graphs. *Proceedings of the* 8*th Working Conference on Reverse Engineering*, 303-309.

[12] Bille, P. (2005) A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science*, **337**, 217-239. https://doi.org/10.1016/j.tcs.2004.12.030

[13] Demaine, E.D., Mozes S., Rossman, B. and Weimann, O. (2009) An Optimal Decomposition Algorithm for Tree edit Distance. *ACM Transactions on Algorithms*, **6**, 2:1-2:19.

[14] Jiang, L., Misherghi, G., Su, Z. and Glondu, S. (2007) DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. *Proceedings of* 29*th International Conference on Software Engineering* (*ICSE* 2007), 96-105.

[15] Zhang, K. and Shasha, D. (1989) Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *Society for Industrial and Applied Mathematics Journal on Computing*, **18**, 1245-1262. https://doi.org/10.1137/0218082

[16] Bellon, S., Koschke, R., Antonio, G., Krinke, J. and Merlo, E. (2007) Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, **SE-33**, 577-591. https://doi.org/10.1109/TSE.2007.70725