International Research Symposium on Engineering and Technology
IRSET2019, pp. 106-116, 2019; ISBN: 978-986-5654-01-6

IRSET-0298
A Method for Detecting Modified Code Clones in a Program

Masakazu Takahashi[a],*, Yunarso Anang[b], and Yoshimichi Watanabe[c]
[a] Department of Computer Science and Engineering, University of Yamanashi, Japan
E-mail address: mtakahashi@yamanashi.ac.jp
[b] Department of Computational Statistics, Institute of Statistics, Indonesia
E-mail address: anang@stis.ac.id
[c] Department of Computer Science and Engineering, University of Yamanashi, Japan
E-mail address: nabe@yamanashi.ac.jp

Abstract

In modern programming, new functions are often developed by coping a portion of an existing program and modifying it. As a result, the resulting program may ultimately contain many portions that are similar with a few modifications, making it difficult to maintain the program. Thus, these similar portions, which are referred to as gapped code clones (GCCs), should be integrated into a single shared function. However, it is difficult to detect GCCs using exact matching criteria because they differ slightly from each other. Hence, in this study, we propose a program line-based GCC detection method using the Smith–Waterman algorithm, which was designed to detect similar character string based on an original character string. A GCC detection tool was developed and applied to various existing programs to validate this approach. Results revealed that the proposed method can be used to detect GCCs with high accuracy compared with existing tools and within an acceptable time.

Keywords: Gapped Code Clone, Smith-Waterman Algorithm, Software Development

1. Introduction

When adding a new function to an existing program, engineers tend to copy a portion of the existing program and modify it as needed to serve the intended purpose [1]. As a result, the final program includes many similar program potions, which are referred to as code clones (CCs). This makes it difficult to maintain the program. For example, if the original portion of the code contains an error, the error will be propagated to all of the CCs that are derived from this portion; hence, it becomes time-consuming to correct the errors in all CCs. Therefore, various methods have been developed to detect CCs; for example, refactoring CCs involves redesigning and integrating the CCs into the appropriate program structure.

CCs are often modified by changing the variable names, modifying the values of the constants, and adding instructions (lines). Hence, the developed CCs contain slight differences and do not match perfectly [2]; these are called gapped code clones (GCCs). However, because of their slight differences, GCCs cannot be accurately detected using exact matching criteria. Thus, in this study, we propose a method to detect GCCs by using the Smith–Waterman (SW) algorithm, which was initially designed to detect similar character strings.

2. Related Works

Many CC detection methods have been proposed so far. These approaches can be categorized as methods based on software metrics, those based on program units, and those focused on the program structure; these approaches are detailed as follows.

The first type of CC detection method is based on software metrics, which are indexes that describe the characteristics of the software. Such metrics include the number of program lines (LOC), the number of branches in the program (CYC), and the degree of cohesion for program portions (COB). The metrics-based method involves calculating the metrics from a program portion (function, method, or block in the program) and comparing the metrics of each program portion to find similarities, which may indicate CCs [3].

The next type of CC detection method is based on the program unit. One such algorithm that was designed to detect GCCs, called LCS (longest common sequence), operates by detecting the longest common sequences (instructions) in the functions, methods, and blocks and selecting the common sequences that exceed a predefined value as CCs [4].

The final type of CC detection method is focused on the program structure: a program structure tree is developed by analyzing the original program, and CCs are detected by comparing the program structure trees and identifying similarities between them [5]. In a modified version of this method, a dependency graph that shows dependencies between program elements is used to detect CCs in the same way [6].

Because the metrics-based method and the LCC algorithm detect CCs based on units of the function, subroutine, block, and method, a CC that comprises only a portion of those units will not be detected. Furthermore, the structure-based methods are accompanied by technical difficulties as they require developing program structure trees or dependency graphs from the existing program.

3. Proposed GCC Detection Method

This section describes the GCC detection method that uses the SW algorithm: Section 3.1 presents an outline of the SW algorithm, and Section 3.2 presents the corresponding GCC detection method based on this algorithm.

3.1 Smith-Waterman Algorithm

The SW algorithm can detect multiple similar character strings based on an original string [7] . Notably, this algorithm can detect similar strings that contain mismatched and/or additional characters. Three parameters are input to the SW

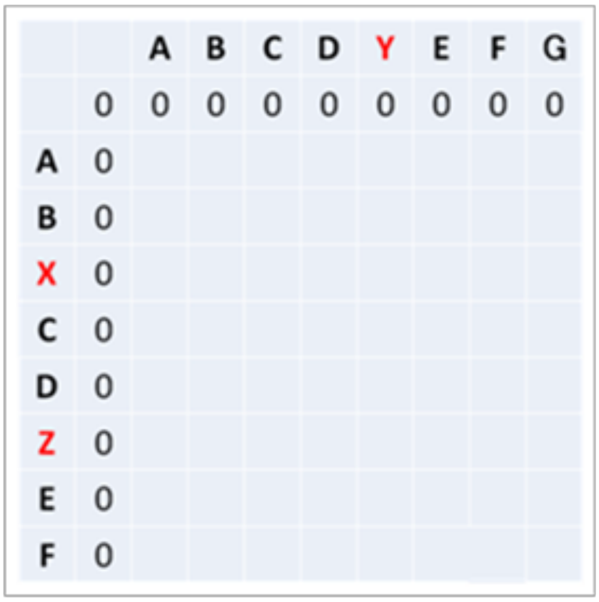
algorithm: match (the weight when the corresponding characters in the compared strings match), mismatch (the weight when the characters do not match), and gap (the weight when a character is added to a string that matches the original string until immediately before the addition). These parameters represent the degrees of tolerance for mismatched and added characters and are specified arbitrarily by the user. Figures 1 – 4 show an example of the calculation of the matching score when the match, mismatch, and gap parameters are set to 1, −2, and −1, respectively.
The method for detecting similar character strings is as follows: where m and n are equal to the lengths of the original and compared character strings, respectively:

(1) Create the table

A two-dimensional table of size (m + 2) by (n + 2) is prepared.

(2) Initialize the table

The characters in the original string (length m) are placed in the cells of the first row (from cell (1, 3) to cell (1, m+2)), and the characters in the compared string (length n) are placed in the cells in the first column (from cell (3, 1) to cell (n+2, 1)). Zeros are placed in the cells in the second row (from cell (2, 3) to cell (2, m+2)) and second column (from cell (3, 2) to cell (n + 2, 2)) (see Fig. 1).



		A	B	C	D	Y	E	F	G
	0	0	0	0	0	0	0	0	0
A	0								
B	0								
X	0								
C	0								
D	0								
Z	0								
E	0								
F	0								

Fig. 1: Initializing the table

(3) Calculate the value in each cell

The value, vi,j, of each cell (i, j) is calculated as shown in Eqs. 1 and 2 (see Fig. 2):In this sample, the following equations are presented as illustration.

$$v_{i,j}(2 \le i, 2 \le j) = \begin{cases} v_{i-1,j-2} + s(a_i, b_j), \\ v_{i-1,j} + gap, \\ v_{i,j-1} + gap. \end{cases} \qquad (1)$$

$$s(ai, bj) = \begin{cases} match & (a_i = b_j), \\ mismatch & (a_i \ne b_j). \end{cases} \qquad (2)$$

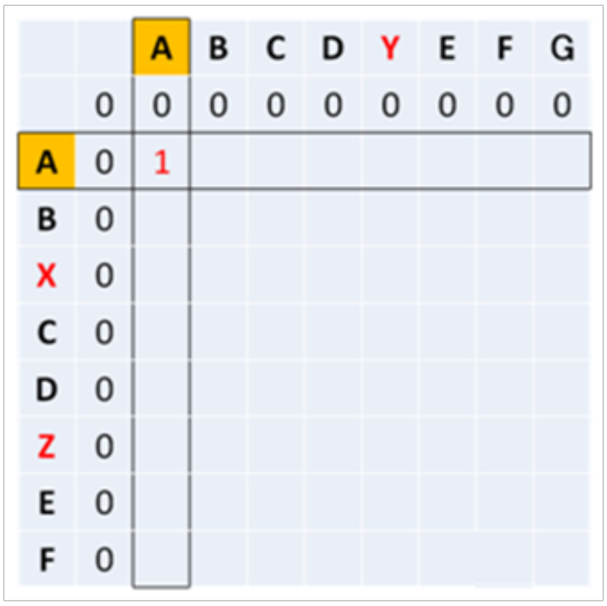
When the calculated value in a cell is not zero, a pointer is set from the value that was used to calculate this value toward the calculated value in order to trace the path toward the output (see Fig. 3). The calculated value is then placed in the cell.

(4) Conduct back-tracing

After calculating all values in the cells, the similar character string is extracted by back-tracing the pointer from the cell that has the maximum value in the table to the cell which has a value of zero (see Fig. 4)

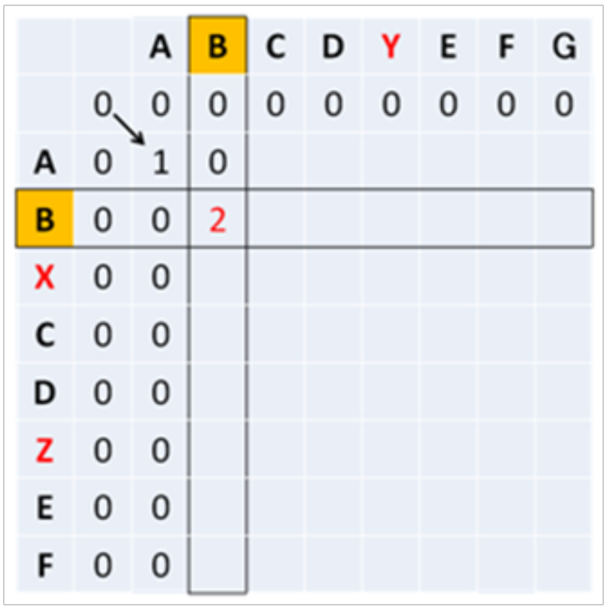
(5) Output the similar character string

The characters that correspond to the cells that back-traced are combined as a string, which is output as the similar character string.



		A	B	C	D	Y	E	F	G
	0	0	0	0	0	0	0	0	0
A	0	1							
B	0								
X	0								
C	0								
D	0								
Z	0								
E	0								
F	0								

Fig. 2 : Calculating the value in each cell



		A	B	C	D	Y	E	F	G
	0	0	0	0	0	0	0	0	0
A	0	1	0						
B	0	0	2						
X	0	0							
C	0	0							
D	0	0							
Z	0	0							
E	0	0							
F	0	0							

Fig. 3: Setting the pointer

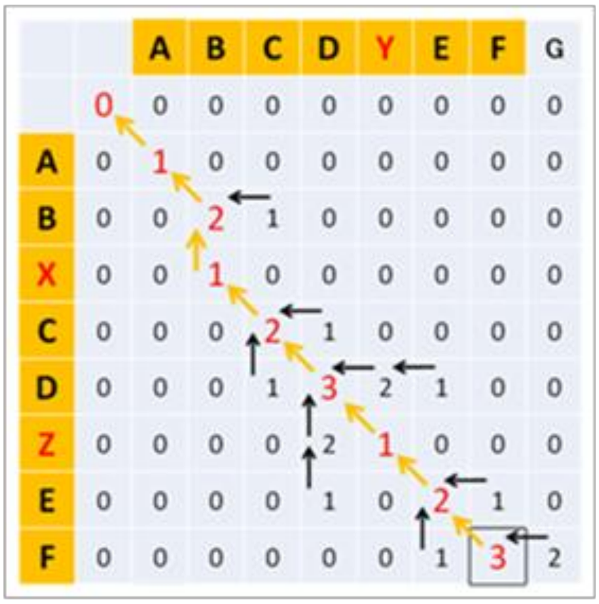


Fig. 4: Back-tracing

3.2 GCC Detection Method Using the SW Algorithm

This section describes the GCC detection method using the SW algorithm that was explained in Section 3.1. Here the objective is to detect GCCs in a program written in Java, which is a representative object-oriented programming language.

First, the characteristics of the GCCs in the program are considered. The GCCs occurred by copying and pasting portions of the original program into the existing program and adding slight modifications, such as changing the names of the variables, adjusting the values of the constants, and adding some instructions. These GCCs are found in the units of blocks (program portions, such as classes, methods, functions, and the subroutines), lines, and tokens (words are punctuated by spaces, brackets, and semicolons). When detecting GCCs, the detection methods based on block units is not adequate because this judgment criterion for similarity is too large so the granularity is insufficient. Similarly, the detection method based on the character unit is not adequate because the granularity for the judgment criteria is too small. Further, the detection method based on the token unit is not adequate because the unit of the GCC does not necessarily match the unit of the original code such as when the detected GCC is partitioned in the middle of the line [8]. Consequently, the proposed detection method is based on the line unit.

The concrete GCC detection method based on the SW algorithm with lines as the units is as follows:

(a) Provide the inputs to the program: target program, minimum length of a GCC (number of lines), maximum gap rate, and parameters for the SW algorithm(match, mismatch, and gap) (Fig. 5(a)).

(b) Identify the target lines (Fig. 5(b)).

(c) Replace the names of the variables, subroutines, classes, and methods and the values of the constants with special characters (Fig. 5(c)).

(d) Identify the sentences (Fig.5(d)).

(e) Calculate the hash value in a line unit (Fig. 5(e)).

(f) Regard one hash value as one character and the combination of multiple characters as a character string (Fig. 5(f)).

(g) Detect similar portions (corresponding to GCCs) from the resulting character string using the SW algorithm (Fig. 5(g)).

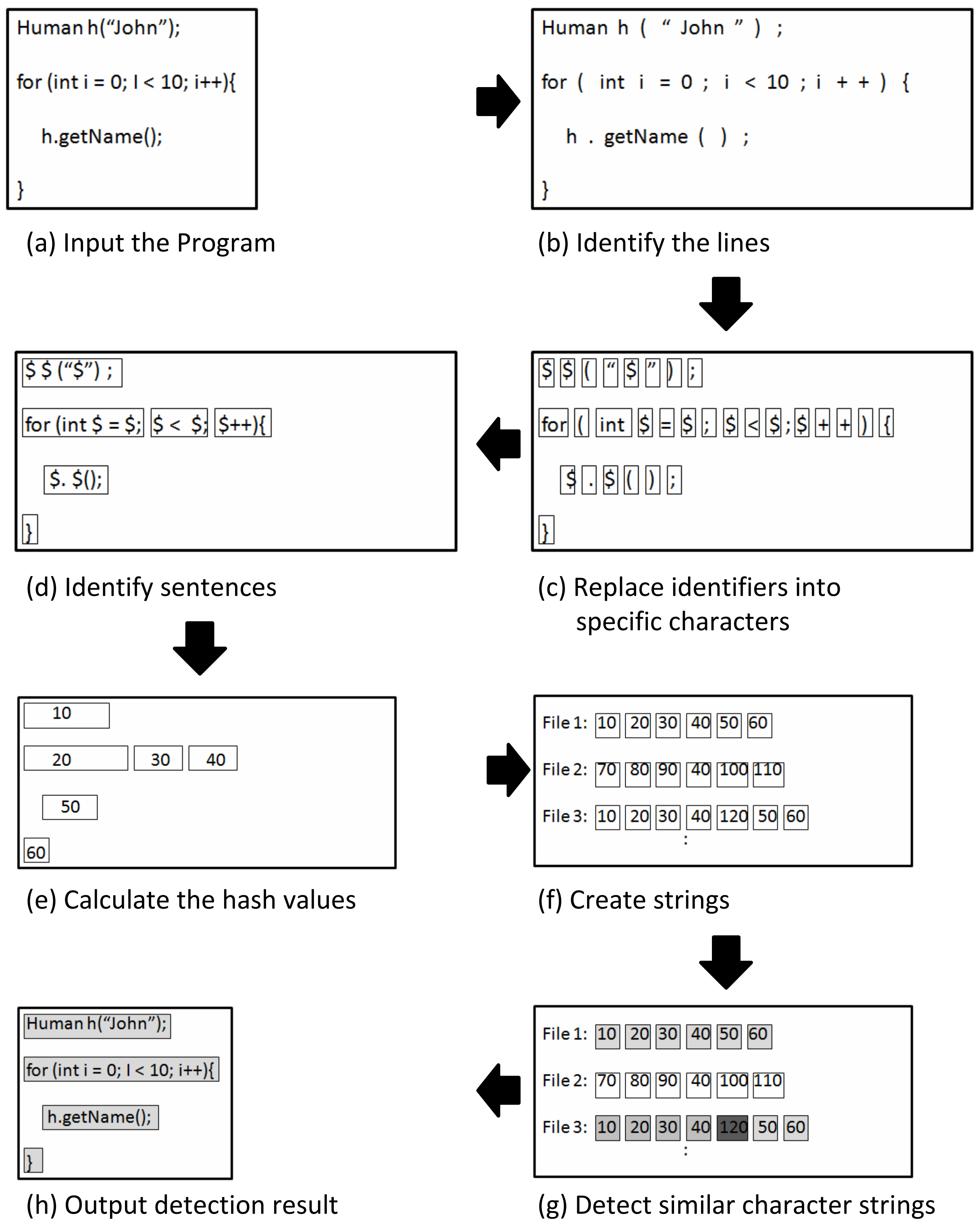
(h) Output the start and end line numbers that delineate the GCCs and the gaps (Fig. 5(h)).

(i) Repeat steps (b) through (f) on the entire target program.

The GCC that has the maximum length in the program can be detected as described above. However, there is a possibility that the program includes multiple GCCs. To detect all GCCs, operation (g) is modified by detecting the cells that satisfy the following conditions from the lower right to the upper left of the table:

$$v(i, j) > 0. \qquad (3)$$

$$v(0, j) = v(i, 0). \qquad (4)$$


Human h("John");
for (int i = 0; I < 10; i++){
h.getName();
}
(a) Input the Program
Human h (" John ") ;
for (int i = 0 ; i < 10 ; i + +) {
h . getName ( ) ;
}
(b) Identify the lines
$ $ (" $ ") ;
for (int $ = $; $ < $; $ + +) {
$. $ ( ) ;
}
(c) Replace identifiers into specific characters
$ $ ("$") ;
for (int $ = $; $ < $; $++){
$. $();
}
(d) Identify sentences
10
20 30 40
50
60
(e) Calculate the hash values
File 1: 10 20 30 40 50 60
File 2: 70 80 90 40 100 110
File 3: 10 20 30 40 120 50 60
(f) Create strings
(g) Detect similar character strings
(h) Output detection result
Fig. 5 GCC detection procedure with lines as the units.

Here, $v(i, j)$ explains the value of cell(i, j). Equation (3) represents the condition that a particular token is the last token in the sentence, and Eq. (4) represents the condition that the last tokens in each sentence coincide. The cells that satisfy those conditions are searched from the lower right cell to the upper left cell, and the back-tracings are conducted from those cells. As a result, all GCCs that satisfy the conditions are detected. Then, back-tracing is conducted from those cells as well.

Furthermore, several countermeasures are included to increase the accuracy of the GCC detection: To avoid detecting GCCs that have already been detected in other GCCs, the cells that have already been back-traced cannot be regarded as starting cells (points). In addition, GCCs that are shorter than a predefined minimum number of lines and those that exhibit rates of mismatches and gaps above the predefined maxima are excluded from the GCC candidates.

4. Proposed GCC Detection Method

To evaluate the proposed method, a GCC detection tool was developed using C++ language. The tool was installed on a PC equipped with Intel Xeon E3-1230v2 processor (3.3 GHz) and 8 GB main memories. This tool was applied to several programs (Table 1) [9, 10], and the accuracy and evaluation time of GCC detection were evaluated for each program.

When detecting GCCs using the SW algorithm, it is known that the detection accuracy depends on the given parameters (match, mismatch, gap, minimum GCC lines, and maximum gap rate). Thus, preliminary experiments were conducted by changing these parameters to obtain adequate values for these parameters (Table 2); the optimized values are as follows:

match: 2
mismatch: −3
gap: −2
minimum statements: 20
maximum gap rate: 0.15

The detection accuracy of the developed GCC detection tool was evaluated in terms of the following indexes:

$$\mathrm{Recall} = \frac{|S|}{|S_{ref}|} \qquad (5)$$

$$\mathrm{Precision} = \frac{|S|}{|S_{cand}|} \qquad (6)$$

$$Fmeasure = \frac{(2 \times \mathrm{Recall} \times \mathrm{Precision})}{(\mathrm{Recall} + \mathrm{Precision})} \qquad (7)$$

Table 1: Target programs for evaluating the GCC detection

Name	Language	Total LOC
Eclipse-ant	Java	70008
Netbeans-javadoc	Java	14360

Table 2: Results of the sensitivity analysis for the parameters

mismatch	gap	gap	num	Correct	Recall	Precision	F-

		rate	of GCC	GCC			measure
-3	0	0.1	94	4	0.13	0.04	0.06
-3	1	0.15	57	8	0.27	0.14	0.18
-3	1	0.20	100	13	0.43	0.13	0.20
-3	1	0.25	175	16	0.53	0.09	0.16
-3	2	015	39	8	0.27	0.21	0.23
-3	2	0.20	60	11	0.37	0.22	0.27
-3	2	0.25	109	12	0.40	0.11	0.17
-4	1	0.15	61	8	0.27	0.13	0.18
-4	1	0.20	103	14	0.47	0.14	0.21
-4	1	0.25	160	16	0.53	0.10	0.17

Table 3: Evaluation of results

Program	Detection Tool	Recall	Precision	F-measure	Detection Time[s]
eclipse -ant	Proposed	0.37	0.22	0.27	22
	NiCad	0.15	0.19	0.16	4
netbeans -javadoc	Proposed	0.42	0.18	0.26	7
	NiCad	0.17	0.11	0.13	3

where S_{ref} represents the set of GCCs that are known to be correct GCCs, S_{cand} represents the set of GCCs that were detected by the tool, and S represents the set of GCCs that are included in the S_{ref} and S_{cand}. Recall represents the ratio of GCC detection among the correct GCCs; a high recall value indicates that few GCCs were missed. Precision represents the ratio of GCCs that were correct among all GCCs that were detected; a high precision value indicates that few of the GCCs were not actually GCCs. As it is known that there is a trade-off relationship between recall and precision, the harmonic mean (Fmeasure) between these two metrics was also calculated.

Table 3 shows the recall, precision, and Fmeasure results from the GCC detection using the proposed method compared with those using NiCad, which is a representative GCC detection tool [11, 12]. These results show that the recall, precision, and Fmeasure attained using the proposed method were superior to those obtained using NiCad. Hence, it was concluded that the proposed method and tool can be used to adequately detect GCCs.

However, the detection time using the proposed method and tool was 2.3–5.4 times that when using NiCad. This difference was attributed to the additional countermeasures that were implemented to improve the GCC detection accuracy, which requires additional calculations; because the proposed method conducts multiple searches of the whole table, the order of the computational complexity becomes several times that of the NiCad algorithm. However, in practical software development, the operation of GCC detection is not conducted frequently, so the calculation time of 20 s for a 7000 LOC program should be acceptable.

5. Conclusion and Future Work

Here, we proposed a novel GCC detection method using the SW algorithm. Evaluation experiments were conducted to determine the detection accuracy, and it is found that the proposed method and tool provide higher detection accuracy compared with an existing detection method and tool. Additionally, it is found that GCCs could be detected within a practical time using the proposed approach.

Here, the parameters that are necessary for GCC detection (such as match, mismatch, gap, minimum sentences, and maximum gap rate) were determined experimentally. However, the optimal values may differ depending on the program structure. Therefore, in future studies, a method to determine adequate parameters by evaluating different possibilities (i.e., parameter sensitivity analysis) should be developed.

In addition, it should be noted that copying and pasting portions of a program is generally conducted at the scale of several tens of LOC. Therefore, it is considered that GCC detection on the line unit is closer to what practically occurs than the GCC detection using sentences and/or tokens as the units. Hence, to further refine the proposed method, additional investigations should be conducted to determine the best detection unit by comparing the results of GCC detection using sentences and lines as the units.

Finally, it should be considered that engineers generally conduct GCC refactoring to improve the maintainability of the program, but not all detected GCCs can be refactored. Therefore, it would be desirable to develop a method to select GCCs that are suitable for refactoring from all detected GCCs [13].

Acknowledgments

This research was supported by the Scientific Research Grant of the SUZUKI foundation “A Safety Analysis Method Cooperating FMEA, FTA, and HAZOP for Embedded Control Software” and Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science "Integrated Analysis Method for hazard cased by software interaction cooperating with multiple safety analysis methods."

6. References

[1] Higo Y., Kusumoto S., and Inoue E. (2008) A Survey of Code Clone Detection and Its Related Techniques, IEICE Transactions on Information and Systems, Vol. 91-D, No. 6, 1465-1481.

[2] Kim M., Bergman L., Lau T., and Notkin D. (2004) An Ethnographic Study of Copy and Paste Programming Practices in OOPL, Proc. 2004 International Symposium on Empirical Software Engineering, 83-92.

[3] Mayland J., Leblanc C., and Merlo E. (2013) Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, in Proc. of the 12th International Conference on Software Maintenance, 244-253.
[4] Roy C., and Cordy J. (2008) NiCad: Accurate Detection of Near-miss Intentional Clones Using Flexible Pretty-printing and Code Normalization, Proc. 16th International Conference on Program Comprehension, 172-181.
[5] Jiang L., Misherghi G,, Su Z., and Glondu S. (2007) DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, in Proc. of the 29th International Conference on Software Engineering, 96-105.
[6] Krinke J. (2001) Identifying Similar Code with Program Dependence Graphs, in Proc. of the 8th Working Conference on Reverse Engineering, 301-309.
[7]Smith-Waterman, bioinformatics (online), < http://bi.biopapyrus.net/seq/smith-waterman. html> (2017-2-20 Accessed).
[8] Murakami H., Hotta K., Higo Y., and Igaki H., Kusumoto S. (2013) Gapped Code Clone Detection Using The Smith-Waterman Algorithm, Proc. of Software Engineering Symposium 2013, 1-8.
[9] The Apache Software Foundation, Welcome, Apache Ant (online), <http://ant.apache.org /> , (2017-2-20 Accessed).
[10] Oracle: Javadoc support, NetBeans (online), <https://edu.netbeans.org /quicktour/ java doc.html> (2017-2-20 Accessed).
[11] Bellon S., Koschke R., Antniol G., Krinke J., and Merlo E. (2007) Comparison and Evaluation of Clone Detection tools, IEEE Trans. on Software Engineering, Vol. 31, No. 10, 804-818.
[12] Bellon S. Detection of Software Clones (online), <http://www2.informatik.uni- stuttgart.de/iste/ps/clones/index.html> (2017-2-20 Accessed).
[13] Choi E., Yoshida N., Ishio T., Inoue K. and Sano T. (2011) Extracting Code Clones for Refactoring Using Combinations of Clone Metrics, in Proc. of IWSC'11, 7-13.