OPEN ACCESS Journal of Software Engineering

ISSN 1819-4311
DOI: 10.3923/jse.2017.266.274

Research Article
Detection of Code Clone Based on Source Fragment Alignment

Jie Wang, Dongjin Yu and Xiang Shu

School of Computer Science and Technology, Hangzhou Dianzi University, 310018 Hangzhou, China

Abstract

Background: Developers often reuse code segments through copy-paste operation with or without modification during software development which leads to so-called code clone. Code clone brings about some convenience for developers. However, it takes difficulties to the understanding and maintenance of software at the same time. **Materials and Methods:** A new code clone detection method based on source code alignment is proposed. First, the source code is transformed to token sequences through code preprocessing. Afterwards, the MD5 hash values of each line are calculated in the token sequences. Finally, the candidate code clones are detected based on the calculation of similarity scores of hash sequences. **Results:** An extensive experiment on 8 open source systems is conducted to measure the precisions and recalls. The results show that the proposed method can detect code clone more effectively than the current methods. **Conclusion:** The acceleration penalty strategy helps improve the accuracy of code clone detection, because the matched source sequences can be broken into two pairs of more-matched ones if some middle source fragments are not so matched. Additional, following the closed trace-back paths, the proposed method could skip some source fragments, thus further improves its effectiveness.

Key words: Code clones, duplicate code, code clone detection, code fragments, mosaic problem, code alignment, trace back, similarity scores, token sequences, hash values

Received: December 02, 2016 Accepted: February 16, 2017 Published: June 15, 2017

Citation: Jie Wang, Dongjin Yu and Xiang Shu, 2017. Detection of code clone based on source fragment alignment. J. Software Eng., 11: 266-274.

Corresponding Author: Dongjin Yu, School of Computer Science and Technology, Hangzhou Dianzi University, 310018 Hangzhou, China

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

Sequences of duplicate code with or without modification are known as code clones or just clones. On one hand, code clones can save time and energy for developers[1], on the other hand, the repeated code segments caused by such copy-paste behavior bring about difficulties to software understanding, maintaining and other related works[2]. Therefore, it is necessary to detect existing code clone in source code for developers to re-engineer software systems.

There are many kinds of classifications for code clone. One of the most widely accepted classifications divided code clones into four types[3]. Type-1 clones are the almost exact clones which are only different on spaces, comments and layout. Type-2 and type-3 are near-miss clones, in which type-2 clones have additional differences of variables, identifiers and constants in addition to type-1 clone differences, whereas type-3 clones add, modify and remove code fragments compared with original ones. Type-4 is semantic clone that the code segments realize the same function while having different grammatical structure. The detection of type-1 and type-2 clones is relatively mature and type-3 clone detection remains to be improved, while the detection of type-4 clones is still in the exploring stage.

Existing code clone detection approach can be categorized as text-based, token-based, AST-based, PDG-based and metric-based[4]. The text-based approach has high precision but its recall is low. Meanwhile, the token-based one has relatively low time and space complexity and is free of languages. However, the token-based method is difficult to detect type-3 code clones. The AST-based one and PDG-based one need to transform the source code to abstract syntax tree or program dependence graph before code clone detection thus cost a lot. Finally, since different code segments are likely to have the same metrics, the metric-based approach would produce a good few of false positives.

In this study, a novel method based on token sequence alignment inspired by Smith-Waterman algorithm is proposed, which can detect type-1, type-2 and type-3 clone. Smith-Waterman algorithm is a gene sequence matching method in biological science, which was firstly proposed by Smith and Waterman[5]. The characteristics of Smith-Waterman algorithm make it able to detect code clones based on the token sequences without complex transformation of the source code. Murakami *et al.*[6] presented a method that detect gapped code clones based on the origin Smith-Waterman algorithm. However, the so-called mosaic problem would occur when Smith-Waterman algorithm is applied to the alignment for long sequences. Here, mosaic problem refers to the conservative region with low similarity, which appears in the optimal alignment of sequence matching, thus leading to the low accuracy of code clone detection[7]. Different from Murakami's method, in this study, an acceleration penalty strategy is introduced to improve the accuracy of code clone detection by eliminating the conservative regions with low similarity scores. Furthermore, following the closed trace-back paths, the proposed method could skip some source fragments, thus further improves the effectiveness.

MATERIALS AND METHODS

Definitions

Definition 1: A token is a minimum meaningful element in source code, such as identifier, constant, symbol and delimiter, denoted by T.

Definition 2: A token sequence is a sequence of tokens generated after the normalization of source code. For a given source file f, its token sequence can be represented as: $TS_f = (T_1, T_2, T_3,...,T_i,...,T_t)$, where, T_i is a token in f and t indicates the number of tokens in f.

Definition 3: A source hash sequence is defined as a sequence of values, each representing the hash value of the token sequence of one single line in a source file f, denoted by $SHS_f = (h_1, h_2, h_3,...,h_i,...,h_k)$, where, k is the number of hash values (or lines) in f. In the method, the MD5 hash value is calculated for the token sequence of each line in the source file.

Definition 4: An i-th left prefix is the sub sequence consisting of the first i elements of a source hash sequence. For a given source hash sequence $SHS_f = (h_1, h_2, h_3,...,h_i,...,h_k)$ of file f, its i-th left prefix is $LP_i^f = (h_1,h_2,h_3,...,h_i)$, where, $1 \leq i \leq k$.

Definition 5: An alignment matrix $SSM^{p,q}$ represents the alignment of two source hash sequences for source files p and q, in which the cell locates at the i-th row and the j-th column indicates the alignment score of hash sequences values of the first i lines from file p and the first j lines from file q. More specifically, given two source hash sequences SHS_p and SHS_q for files p and q, respectively, $SSM_{i,j}^{p,q}$ in the alignment score matrix $SSM^{p,q}$ represents the alignment score between LP_i^p and LP_j^q.

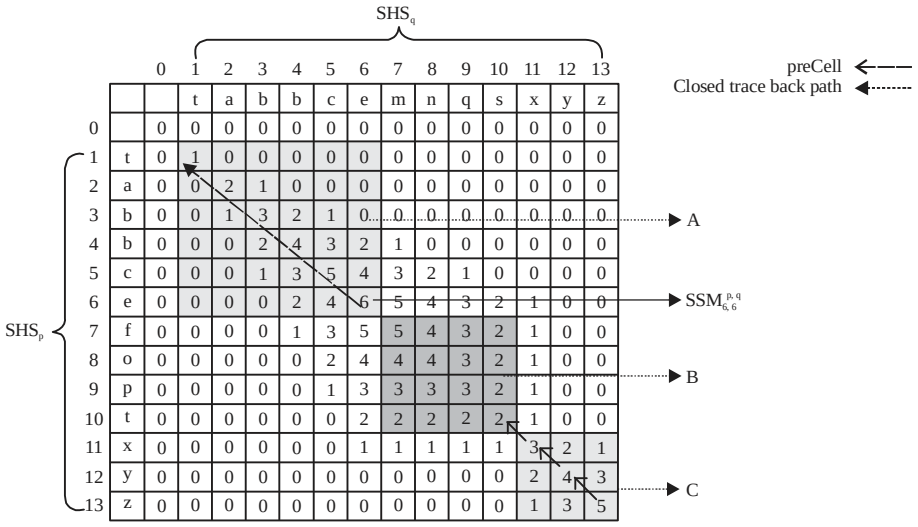


Fig. 1: Example of alignment matrix using the original Smith-Waterman algorithm

Figure 1 gives an example of alignment matrix for sequences "tabbcemnqsxyz" and "tabbcefoptxyz". For instance, $SSM^{p,q}_{6,6}$ indicates the alignment score between the first 6 lines of SHS_p and the first 6 lines of file SHS_q.

Mosaic problem refers to the mismatched fragment existed in the alignment of source hash sequences. When the mismatched fragments are long, the mosaic problem would decrease the accuracy of alignment.

Given two sequences SHS_p = "tabbcefoptxyz" and SHS_q = "tabbcemnqsxyz". Figure 1 shows the alignment matrix of SHS_p and SHS_q using the original Smith-Waterman algorithm. The final alignment would be:

SHS_p: t a b b c e f o p t -- -- -- -- x y z
SHS_q: t a b b c e -- -- -- -- -- m n q s x y z

Figure 1 shows that A and C are high similar areas and B is the low similarity area due to the so called mosaic problem. However, in the proposed method, the alignment would be broken into two more similar alignments:

SHS_p: t a b b c e x y z
SHS_q: t a b b c e x y z

Figure 2 shows that the improved Smith-Waterman algorithm will cut off the low similarity area B and only reserves the areas A and C which have high similarity. Obviously, the alignment A and C can reflect the similarity of p and q in a more accurate way.

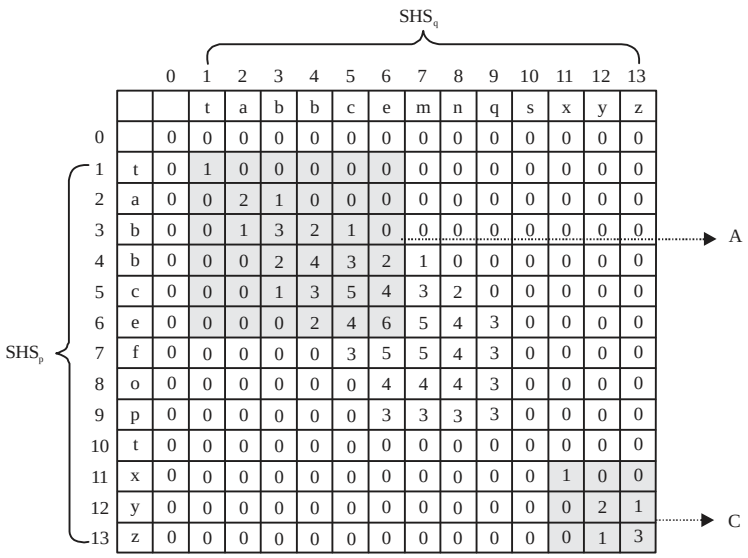
Proposed method: The process of the proposed code clone detection approach can be divided into the following two phases: Code preprocessing and code clone detection. The phase of code preprocessing involves parsing, normalizing and sequencing the source files and persisting the intermediate results, whereas the phase of code clone detection deals with the source hash sequence alignment and identification of candidate code clones.

Code preprocessing

Parse and normalize source code: During this step, the source files are lexically analyzed for the normalization. More specifically, white spaces and comments are removed, whereas the user defined keywords, literals and identifiers are transformed into specific tokens. In this way, the source files are converted into token sequences. Meanwhile, the line numbers are recorded for locating code clones in source files later.

Sequence the source code: The MD5 (Message digest) algorithm is adopted to calculate the hash value of each line in the transformed token sequences. In this way, source files are converted into source hash sequences which are the inputs of the next phase of code clone detection.

Persist the intermediate results: In order to realize the incremental update and fast search, two indexes, i.e.,



		0	1	2	3	4	5	6	7	8	9	10	11	12	13
			t	a	b	b	c	e	m	n	q	s	x	y	z
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	t	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	a	0	0	2	1	0	0	0	0	0	0	0	0	0	0
3	b	0	0	1	3	2	1	0	0	0	0	0	0	0	0
4	b	0	0	0	2	4	3	2	1	0	0	0	0	0	0
5	c	0	0	0	1	3	5	4	3	2	0	0	0	0	0
6	e	0	0	0	0	2	4	6	5	4	3	0	0	0	0
7	f	0	0	0	0	0	3	5	5	4	3	0	0	0	0
8	o	0	0	0	0	0	0	4	4	4	3	0	0	0	0
9	p	0	0	0	0	0	0	3	3	3	3	0	0	0	0
10	t	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	x	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	y	0	0	0	0	0	0	0	0	0	0	0	0	2	1
13	z	0	0	0	0	0	0	0	0	0	0	0	0	1	3

Fig. 2: Example of alignment matrix using the proposed algorithm

GlobalIndex and TimestampIndex are created. GlobalIndex is the global index which maps names of source files with their source hash sequences. Meanwhile, TimestampIndex saves the time stamps of latest source file normalization. When initiating the code clone detection process, TimestampIndex is compared with file modification time. Only the file that has been changed since the latest normalization needs to be re-normalized.

Code clone detection: This phase can be further divided into the following two steps.

Alignment of hash sequences: During this step, an alignment matrix is constructed to represent the alignment between two hash sequences. The key to this step is how to calculate the alignment scores, which is given by Eq. 1, where CT represents the predefined cutting threshold. The $SSM_{i,j}$ is the score of cell(i, j), whereas cell(i, j) is the cell corresponding to the i-th row and j-th column in the matrix. Additional, to calculate the alignment scores, the maxScore of each cell which refers to the max score along the closed trace back path, also needs to be set, as Eq. 3 shows, where, $(m, n) \in \{\alpha | \alpha \in \{(i-1, j), (i, j-1), (i-1, j-1)\}\}$, $i \geq 1, j \geq 1$:

$$SSM_{i,j} = \begin{cases} \max \begin{cases} SSM_{i,j} + \sigma(SSH_p[i], SSH_q[j]) \\ SSM_{i-1,j} + \sigma_{Delete} \\ SSM_{i,j-1} + \sigma_{Insert} \\ 0 \end{cases} & \text{if } maxScore_{i,j} - SSM_{i,j} < CT \\ & i \geq 2, j \geq 2 \\ 0, \text{ if } maxScore_{i,j} - SSM_{i,j} \geq CT \end{cases} \quad (1)$$

Where:

$$\begin{cases} SSM_{0,0} = SSM_{i,0} = SSM_{0,j} = 0 \\ \sigma(SSH_p[i], SSH_q[j]) = \begin{cases} \text{Match} & \text{if } SSH_p[i] = SSH_q[j] \\ \text{Mismatch} & \text{if } SSH_p[i] \neq SSH_q[j] \end{cases} \\ \sigma_{Delete} = \text{Delete}, \sigma_{Insert} = \text{Insert} \end{cases} \quad (2)$$

$$maxScore_{i,j} = \begin{cases} \begin{cases} 0 & \text{if } SSM_{i,j} = 0 \\ \max\{SSM_{i-1,j-1}, maxScore_{i-1,j-1}\} & \text{if } SSH_p[i] = SSH_q[j] \\ \max\{SSM_{m,n}, maxScore_{m,n}\} & \text{others} \end{cases} & \text{if } maxScore_{i,j} - SSM_{i,j} < CT \\ 0, \text{ if } maxScore_{i,j} - SSM_{i,j} \geq CT \end{cases} \quad (3)$$

Here, an acceleration penalty strategy is introduced to avoid the so-called mosaic problem. The main idea of acceleration penalty strategy can be summarized as follows. A cutting threshold is predefined in the process of calculating the score of each cell. If the difference of a cell's maxScore and Score is equal or exceeds than the cutting threshold, the maxScore and Score are reset. In this way, the matched sequences can be broken into two pairs of more-matched ones if some middle fragments are not so matched.

As algorithm 1 illustrates, an initial score matrix for SHS_p and SHS_q (Lines 2-8) is created first. Afterwards, the Score and maxScore of each cell are calculated (Lines 10-18). In particular, if the difference of a cell's maxScore and Score is greater or equal than a predefined cutting threshold, its Score and maxScore are reset to 0 (Lines 19-21). Finally, if a cell's Score is above a predefined cutting threshold and it's Score is

greater than it's maxScore and the i-th hash value of SHS_p and the j-th hash value of SHS_q corresponding to the cell are exactly matched, cell(i, j) is regarded as the start point of a closed trace back path (Lines 22-24).

Algorithm 1: Code sequence alignment

Input: SHS_p, SHS_q // two source hash sequences to be aligned, representing the files p and q, respectively
CT// the predefined threshold used for cutting mismatched parts of SHS_p and SHS_q
Output: TraceSet // saving the starting points of the trace back paths

1. m = SHS_p. length, n = SHS_q. length
2. create an initial alignment matrix $SSM^{p,q}$
3. for each i in [0, m] do
4. $SSM^{p,q}_{i,0} \leftarrow 0, maxScore_{i,0} \leftarrow 0$
5. end for
6. for each j in [0, n] do
7. $SSM^{p,q}_{0,j} \leftarrow 0, maxScore_{0,j} \leftarrow 0$
8. end for
9. TraceSet←ø
10. for each i in [1, m] do
11. for each j in [1, n] do
12. if (SHS_p[i] = = SHS_q[j])
13. $SSM^{p,q}_{i,j} = \max(SSM^{p,q}_{i-1,j-1} + Match, SSM^{p,q}_{i-1,j} + Delete, SSM^{p,q}_{i,j-1} + Insert)$
14. $\max Score_{i,j} = \max(SSM^{p,q}_{i-1,j-1}, \max Score_{i-1,j-1})$
15. else
16. $SSM^{p,q}_{i,j} = \max(SSM^{p,q}_{i-1,j-1} + Mismatch, SSM^{p,q}_{i-1,j} + Delete, SSM^{p,q}_{i,j-1} + Insert)$
17. $maxScore_{i,j} = \frac{\max(SSM^{p,q}_{i-1,j-1}, SSM^{p,q}_{i-1,j}, SSM^{p,q}_{i,j-1}, maxScore_{i-1,j-1}}{maxScore_{i-1,j}, maxScore_{i,j-1})}$
18. end if
19. if $maxScore_{i,j} - SSM^{p,q}_{i,j} \geq CT$
20. $SSM^{p,q}_{i,j} \leftarrow 0, maxScore_{i,j} \leftarrow 0$
21. end if
22. if $SSM^{p,q}_{i,j} \geq CT \;\&\&\; SSM^{p,q}_{i,j} \geq maxScore_{i,j} \;\&\&\; SHS_p[i] == SHS_q[j]$
23. TraceSet← (i, j)
24. end if
25. end for
26. end for

Identification of code clones: After the alignment matrix is calculated, the closed trace back paths are then determined to find the matching subsequences, or code clones.

Definition 6: The preCell of a cell in the alignment matrix refers to the cell which its score is calculated from. The preCell of cell (i, j) can be obtained as follows:

$$preCell(i, j) = \begin{cases} (i-1, j-1) & SSM_{i,j} = SSM_{i-1,j-1} + \sigma(SHS_p, SHS_q) \\ (i-1, j) & SSM_{i,j} = SSM_{i-1,j} + \sigma_{Delete} \\ (i, j-1) & SSM_{i,j} = SSM_{i,j-1} + \sigma_{Insert} \end{cases} \quad (4)$$

Definition 7: A closed trace back path is the path which starts at a cell in the alignment matrix and follows its preCell until reaching the cell with 0 score. It represents the alignment of the given hash sequences.

The examples of preCell and closed trace back path are presented in Fig. 1. Algorithm 2 introduces how to find closed trace back paths that correspond to code clones. Firstly, the status of each cell is initialized to be unvisited (Lines 1-3). Afterward, the cell with maximum score among all unvisited ones is set to visited. Starting from this cell, the pre-cells are continuously traced back until reaching the cell with 0 score. In this way, a trace back path is obtained. Furthermore, the status of each cells in the obtained trace back path is set to visited and add their row numbers and column numbers to stack1 and stack2, respectively. This process is repeated until all cells are visited (Lines 5-12). Finally, the track back paths are mapped to the code clone pairs according to the records in stack1 and stack2 (Lines 13-14).

Algorithm 2: Detection of closed trace back paths

Input: TraceSet
Output: MatchList

1. for each cell in TraceSet
2. cell.status ← unvisited
3. end for
4. for each cell in TraceSet
5. choose the cell Max with maximum score
6. Stack1←ø, Stack2←ø
7. While (Max.score! = 0 & & Max.status = unvisited)
8. Max.status←visited
9. Push Max.rowNumber to stack1
10. Push Max.columnNumber to stack2
11. Max←Max.preCell
12. end while
13. map the records in stack1and stack2 to the source code SC_1 and SC_2
14. MatchList←(SC_1, SC_2)
15. end for

RESULTS

To evaluate the effectiveness of the proposed method, 8 open source systems were selected and the results are compared with Bellon's benchmark. The benchmark was constructed through the manually verification of the 2% code clone results of all the tools. The characteristic of 8 systems being tested in the experiment are shown in Table 1. The last column (No. of references) indicates the number of code clones in systems. The experiments were ran on Ubuntu with Intel Core2 Duo CPU E7500 and Redis, an open source, key-value database, is used to persistent the intermediate results.

The values of match, mismatch, insert and delete are set to 2, -2, -1 and -1, respectively according to the study performed by Murakami *et al.*[6]. Since the minimum lines of code clones detected in Bellon's benchmark is 6, the Cutting Threshold is set to 12. Recall, precision and F-measure are used to evaluate the performance of the approach, which can be calculated as follows:

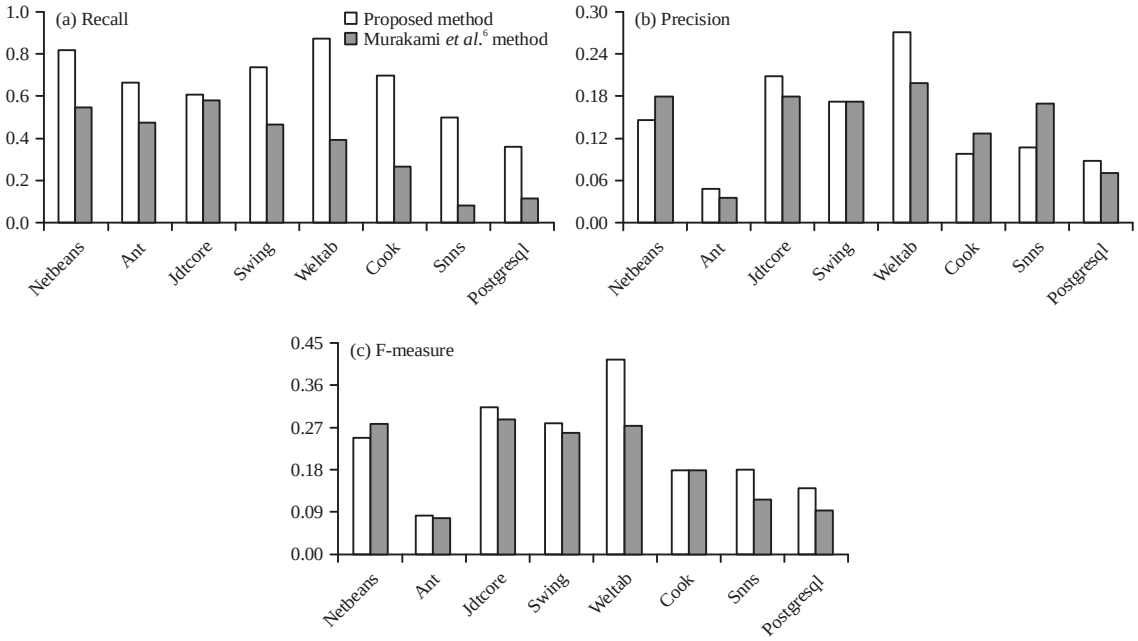


Fig. 3(a-c): Recall, precision and F-measure of the proposed method and Murakami's method

Table 1: Characteristic of the systems

System	Language	No. of files	LOC	No. of references
Netbeans	Java	101	14,360	55
Ant	Java	178	34,744	30
Jdtcore	Java	741	147,634	1,345
Swing	Java	538	204,037	777
Wetlab	C	39	11,460	275
Cook	C	295	70,008	1,036
Snns	C	141	93,867	136
Postgresql	C	322	201,686	555

$$\text{Recall} = \frac{|\text{TC}|}{|\text{RC}|}, \text{Precision} = \frac{|\text{TC}|}{|\text{CC}|}, \text{F} - \text{measure} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \quad (5)$$

where, TC is true clones, RC is reference clones and CC is candidate clones.

The results are compared with Murakami's method. The reason why choosing this method for comparison is that it employs Smith-Waterman algorithm, a similar approach as ours. The detail information of results are given in Fig. 3.

Figure 3a shows that compared with Murakami's method, the proposed method achieves the higher recalls for all systems. In the best case, the recall is increased by almost 40% (weltab), whereas in the worst case, the recall is enhanced by 3% (jdtcore). However, the precisions are not always increased. Figure 3b shows that the precisions of the proposed method are higher than those of Murakami's method for ant, jdtcore, swing, weltab and postgresql but lower for netbeans, cook and snns. By combining the precision and recall, however, the F-measure values of the proposed method outperform those of the Murakami's method for all systems except for netbeans as shown in Fig. 3c. In other words, the method proposed in this study performs significant better than Murakami's method.

Threats to validity: This study discusses the main threats to validity, a key challenge for researchers and practitioners in empirical research work. Generally speaking, several aspects need to be considered, such as construct validity, internal validity and external validity.

Construct validity generally refers to the validity of inferences that observations or measurement tools actually represent or measure the construct being investigated. In the context of study, this is mainly in relation to how the precision and recall are measured. During the experiments, the proposed code clone detection results is evaluated by using Bellon's benchmark which only validates 2% of their code clone detection results. In other words, there are a lot of code clones that are not validated. However, the precisions and recalls are considered to be consistent if all the code clones of the systems had been used to evaluate the methods, simply because of Bellon *et al.*[4] 2% detected clones are evenly distributed in the systems.

Threats to internal validity refers to experimental factors that could influence the results. During the code clone detection process, four parameters, i.e., match, mismatch, insert, delete and cutting threshold are introduced. The four parameters are set to the fixed values for all 8 systems

according to the experience. If the parameters are changed to other values, the results of code clone detection may be different. Therefore, it is necessary to try different parameters to obtain the optimal ones.

As for the external validity, which concerns the extent to which the (internally valid) results of a study can be held to be true for other cases, the proposed method chooses eight systems for code clone detection. Although these systems are often used in code clone researches, they are all programmed in Java or C. It could be worthwhile to replicate the evaluation on other systems programmed in different languages, such as C# and JavaScript.

DISCUSSION

Code clone detection has been one of the main research focuses in the field of software analysis during the past decade. There are quite a few methods proposed for code clone detection at present, which are mainly based on text, token, AST, PDG, metric and some other approach.

Text-based approaches detect code clone after simply preprocessing the source code: remove white space and comments. Cordy's NICAD is a text-based code clone detection tool which can detect type-3 clone effectively[8]. It converts the source code in a specific form of texts and then detects code clone by comparing similarity of texts. Simcad is another code clone detection tool developed by Uddin *et al.*[9], Simcad transforms the source code to hash values through simhash algorithm and applies a three level index to speed up code clone detection. The SDD developed by Lee and Jeong[10] is a tool which can detect code clone for large scale systems, SDD used inverted index to detect both exact and inexact code clones.

Token-based approach firstly converts the source code to token sequence through lexical analysis and detects code clones according to the similarity of token sequences. Finder[11] is a classical code clone detection tool, which transforms the source code into token sequences through normalization at first and then employs suffix tree to detect code clone based on the token sequences. Murakami *et al.*[6] proposed a gapped code clone detection method which applies Smith-Waterman algorithm to clone detection, meanwhile, they rebuild Bellon's benchmark according to add some gap information. Sajnani *et al.*[12] introduced a code clone detection tool SourcererCC, which uses the inverted index to speed up the code clone detection process and decrease the time used for index creating, token comparison times through heuristic filter. The source code is converted to abstract syntax tree according to syntax analysis. If the similarity of two sub trees is above the threshold, the code fragments corresponding to the sub trees are code clone. Baxter *et al.*[13] proposed an AST-based method. They transform the source code into an abstract syntax tree with marked nodes and then calculate the hash values of sub-trees in the syntax tree. Code clones are finally detected through the comparison of hash values of sub-trees. Deckard, developed by Jiang *et al.*[14] calculates the feature vectors base on code fragment's abstract syntax tree, and then it uses LSH algorithm to detect code clones through clustering analysis for the feature vectors. Koschke *et al.*[15] firstly converted the source to abstract syntax tree and serialized the abstract syntax tree to token sequences; code clones are finally detected based on the token sequences through suffix tree algorithm.

The PDG-based approach need to convert the source to PDG, if two sub graphs are similar enough, the code fragments corresponding to the two sub graph are code clone. Higo and Kusumoto[16] presented a heuristic PDG-based clone detection strategy which can reduce the computation complexity. Krinke[17] presented an approach to identify similar code in programs based on finding maximal similar sub graphs in fine grained program dependence graphs. Sargsyan *et al.*[18] introduced a code clone detection method which consisted of two steps: Construct the program dependence graph and find the similar sub graphs. The PDGs were divided into sub graph units to make the method scalable.

Metric-based approach firstly calculates the metric of code fragments and detect code clone through the comparison of metrics. Mayrand *et al.*[19] proposed a method to automatically identify clone functions in source code. The method used datrix to extract 21 metrics which are focused on the control flow metrics and data flow metrics and code clones are detected based on the comparison of metrics. Singh and Sharma[20] combined text based and metric based method to detect file level clones, their method can also detect high level clones in terms of file clones in different or same directories.

Smith-Waterman algorithm is a dynamic programming algorithm which is used to identify the local alignment between gene sequences[5]. As for a very successful approach of gene sequence alignment, there are still a number of methods proposed to improve the efficiency of Smith-Waterman algorithm. Liu *et al.*[21] introduced a parallel version of Smith-Waterman algorithm, which utilizes the emerging Xeon Phis to speed up the implementation of long DNA sequences comparison. According to Sandes and de Melo[22], a high performance computing platform is employed to the parallel implementation of Smith-Waterman algorithm in linear space. With the ability of sequence

alignment, Smith-Waterman algorithm can also be used to detect code clone. For example, Murakami *et al.*[6] applied Smith-Waterman algorithm to detect gapped code cloned.

Mosaic problem appears when the Smith-Waterman algorithm is applied to long sequence alignment. Here, mosaic problem refers to the region with low similarity in the optimal local alignment of sequences. It is necessary to eliminate mosaic problem since it decreases the accuracy of alignment. Huang, Zhang *et al.*[23] tried to solve the mosaic problem during the post process phase. However, their approaches may miss some local alignment with high similarity. Arslan *et al.*[24] introduced a length correction coefficient to avoid the mosaic problem. Nevertheless, it brings new problem of no fixed model to follow if the coefficient is relevant with the data used.

Different from the above code clone detection methods, the proposed one can detect the gapped type-3 clones effectively. In addition, it does not need to transform the source code into a complex intermediate representation (e.g., AST and PDG). Instead, it only needs to convert code into token sequence, which is straightforward.

CONCLUSION AND RECOMMENDATIONS

In this study, a new code clone detection method inspired by the Smith-Waterman algorithm is proposed. In order to resolve the mosaic problem when the Smith-Waterman algorithm is applied to the detection of code clone among the long token sequences, an acceleration penalty strategy is designed to enhance the accuracy of code clone detection. Additional, the trace back mechanism is employed to identify the closed trace back path, which further help identify code clone effectively. Finally, an experiment is demonstrated to validate the proposed method. The results show that the proposed method can detect code clone more effectively.

In the future, the reasons leading to some false positives and improve the precision of the proposed method needs to be investigated. Furthermore, the proposed method will be realized as a tool that can be used in the real software development process. Finally, in order to reduce the threats of validity, we plan to construct a complete benchmark with all clones in the systems (including the systems programmed with C# and JavaScript) and different parameters to further evaluate our method.

SIGNIFICANT STATEMENTS

Sequences of duplicate code with or without modification are known as code clones or just clones. They occur either within a program or across different programs owned or maintained by the same entity. Code clones are generally considered undesirable although they do bring about some convenience for developers. A new method for detecting code clones based on the alignment of two source code fragments is proposed. The acceleration penalty strategy and the closed trace-back paths helps improve the accuracy of code clone detection. An extensive experiment on 8 open source systems is conducted to measure the precisions and recalls. The results show that the proposed method can detect code clone more effectively than the current methods.

ACKNOWLEDGMENT

This study is supported by Natural Science Foundation of China (No. 61100043), Zhejiang Provincial Natural Science Foundation (No. LY12F02003) and the Key Science and Technology Project of Zhejiang (No. 2016F50014 and No. 2017C01010).

REFERENCES

1. Roy, C.K., M.F. Zibran and R. Koschke, 2014. The vision of software clone management: Past, present and future (keynote paper). Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering, February 3-6, 2014, Antwerp, Belgium, pp: 18-33.
2. Giesecke, S., 2006. Generic modelling of code clones. Proceedings of the Seminar on Duplication, Redundancy and Similarity in Software, July 23-26, 2006, Dagstuhl, Germany, pp: 1-23.
3. Sheneamer, A. and J. Kalita, 2016. A survey of software clone detection techniques. Int. J. Comput. Applic., 137: 1-21.
4. Bellon, S., R. Koschke, G. Antoniol, J. Krinke and E. Merlo, 2007. Comparison and evaluation of clone detection tools. IEEE Trans. Software Eng., 33: 577-591.
5. Smith, T.F. and M.S. Waterman, 1981. Identification of common molecular subsequences. J. Mol. Biol., 147: 195-197.
6. Murakami, H., K. Hotta, Y. Higo, H. Igaki and S. Kusumoto, 2013. Gapped code clone detection with lightweight source code analysis. Proceedings of the IEEE 21st International Conference on Program Comprehension, May 20-21, 2013, San Francisco, CA., USA., pp: 93-102.
7. Altschul, S.F., W. Gish, W. Miller, E.W. Myers and D.J. Lipman, 1990. Basic local alignment search tool. J. Mol. Biol., 215: 403-410.
8. Roy, C.K. and J.R. Cordy, 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. Proceedings of the 16th IEEE International Conference on Program Comprehension, June 10-12, 2008, Amsterdam, The Netherlands, pp: 172-181.

9. Uddin, S., C.K. Roy and K.A. Schneider, 2013. SimCad: An extensible and faster clone detection tool for large scale software systems. Proceedings of the IEEE 21st International Conference on Program Comprehension, May 20-21, 2013, San Francisco, CA., USA., pp: 236-238.
10. Lee, S. and I. Jeong, 2005. SDD: High performance code clone detection system for large scale source code. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, October 16-20, 2005, San Diego, CA., USA., pp: 140-141.
11. Kamiya, T., S. Kusumoto and K. Inoue, 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Eng., 28: 654-670.
12. Sajnani, H., V. Saini, J. Svajlenko, C.K. Roy and C.V. Lopes, 2015. SourcererCC: Scaling code clone detection to big-code. Proceedings of the 38th International Conference on Software Engineering, May 16-24, 2015, Firenze, Italy, pp: 1157-1168.
13. Baxter, I.D., A. Yahin, L. Moura, M. Sant'Anna and L. Bier, 1998. Clone detection using abstract syntax trees. Proceedings of the International Conference on Software Maintenance, November 16-20, 1998, Bethesda, MD., USA., pp: 368-377.
14. Jiang, L., G. Misherghi, Z. Su and S. Glondu, 2007. Deckard: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th International Conference on Software Engineering, May 20-26, 2007, Minneapolis, MN., USA., pp: 96-105.
15. Koschke, R., R. Falke and P. Frenzel, 2006. Clone detection using abstract syntax suffix trees. Proceedings of the 13th Working Conference on Reverse Engineering, October 23-27, 2006, Benevento, Italy, pp: 253-262.
16. Higo, Y. and S. Kusumoto, 2011. Code clone detection on specialized PDGs with heuristics. Proceedings of the 15th European Conference on Software Maintenance and Reengineering, March 1-4, 2011, Oldenburg, Germany, pp: 75-84.
17. Krinke, J., 2001. Identifying similar code with program dependence graphs. Proceedings of the 8th Working Conference on Reverse Engineering, October 2-5, 2001, Stuttgart, Germany, pp: 301-309.
18. Sargsyan, S., S. Kurmangaleev, A. Belevantsev and A. Avetisyan, 2016. Scalable and accurate detection of code clones. Program. Comput. Software, 42: 27-33.
19. Mayrand, J., C. Leblanc and E.M. Merlo, 1996. Experiment on the automatic detection of function clones in a software system using metrics. Proceedings of the International Conference on Software Maintenance, November 4-8, 1996, Washington, DC., USA., pp: 244-253.
20. Singh, M. and V. Sharma, 2015. Detection of file level clone for high level cloning. Procedia Comput. Sci., 57: 915-922.
21. Liu, Y., T.T. Tran, F. Lauenroth and B. Schmidt, 2014. SWAPHI-LS: Smith-waterman algorithm on Xeon phi coprocessors for long DNA sequences. Proceedings of the IEEE International Conference on Cluster Computing, September 22-26, 2014, Madrid, Spain, pp: 257-265.
22. Sandes, E.F.D.O. and A.C.M.A. de Melo, 2013. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. IEEE Trans. Parallel Distrib. Syst., 24: 1009-1021.
23. Zhang, Z., P. Berman, T. Wiehe and W. Miller, 1999. Post-processing long pairwise alignments. Bioinformatics, 15: 1012-1019.
24. Arslan, A.N., O. Egecioglu and P.A. Pevzner, 2001. A new approach to sequence comparison: Normalized sequence alignment. Bioinformatics, 17: 327-337.