The Nucleus ***54****, No. 4 (2017) 197-204*

www.thenucleuspak.org.pk

The Nucleus
ISSN 0029-5698 (Print)
ISSN 2306-6539 (Online)

A Technique of Code Clone Detection based on Defined Mechanism for Threshold Calculation

R. Mehboob, S. Shabbir and A. Javed*

Software Engineering Department, UET, Taxila, Pakistan

ARTICLE INFO

Article history :
Received : 11 June, 2017
Accepted : 09 November, 2017
Published : 31 December, 2017

Keywords:
Clone detection
Condition
Statements
Threshold
Tokenization

ABSTRACT

Over the past few years the revolution in the technology and use of programming languages for product development has made code reusability a common practice. Consequently the problem of code cloning is also increasing leading to redundancy and increase in the maintenancet. The real motivation of the proposed research work is to identify code clones from pair of codes that are going to be utilized for a project under consideration. Existing practices such as control flow graphs (CFGs) and abstract syntax tree (AST) promote a high level of abstraction by masking the inner details of the code. Therefore, a strategy is needed to define the mechanism for calculation of the threshold value to identify clones by considering the inner details of the code. This paper presents a defined mechanism for the computation of threshold for code clone detection. Moreover, the inner details of the code are examined by performing the comparative analysis of tokens and conditional clauses. The proposed technique eliminates high level of abstraction caused by the use of CFGs. The proposed method is tested on custom dataset of different sorting algorithms. Experimental results indicate the effectiveness of the proposed method for code clone detection.

1. Introduction

In software engineering, it is essential practice to manage the complexity and evolution of software systems. Code clone occurs when developers systematically copy the previously existing code which solved a similar problem to the one they are currently trying to solve. Developers commonly adopt this practice to minimize the development time and effort. However, such kind of practices may degrade the quality, violate the intellectual property rights and make the code redundant. In code cloning, the copied code is either modified by changing the names of variables or operations. Sometimes the whole statement of the copied program are deleted and relocated. Such clones are hardest to detect. The standards for software quality are raised as the level of complexity and abstraction of software system has increased. Code clone detection is a quality assurance technique that aims to detect those code fragments that are similar to one another in any respect. The duplication of code fragments in the software systems increase the size of the code and hinders the maintainability. The quality of the software systems can be improved by detecting the code clones effectively.

Code clone detection techniques have been proposed to address the issues associated with code duplication. Pradhan et al. [1] implied a graph isomorphism technique to identify the similar design patterns and their frequency. However, there might be a possibility for the existence of various algorithms of same programming problem with different implementation. Li et al. [2] proposed an effective vector based detection mechanism for code cloning through function call that used the structure and properties of the program. The features of the program based on function calls were extracted and compared. The node of each structure was identified and compared through two function calling trees to identify the similarities in the two codes. This method [2] is efficient, simple and less expensive but it has been tested in C++ only. Gupta and Gupta [3] presented a hybrid approach using Abstract Syntax Tree, Program Dependence and Control Flow Graph techniques. Li et al. [4] proposed a technique to measure the resemblance in different programs; this technique consists of two stages. In the first stage, program was converted into a labeled graph. Weisfieler graph kernel was then computed for the labeled graph in the second stage. The subsequent results were eventually compared with the computed kernels for other programs stored in the repository. The abstraction property of this approach creates a limitation of declaring only the isomorphic graphs as clones. In addition, the hard coded threshold value also reduces the accuracy of code clone detection. Tekchandani et al. [5] proposed a method to extract the domain information from source code. Abstract syntax tree was then created by tokenizing and parsing the source code along with generating the data and control flow graphs. Tokenization resolved the problem of graph isomorphism. The complex modifications in the code statements were detected effectively. However, it is unable to handle the frequency of the identifiers as well as similar code fragments in enormous amount of data. Koschke [6] used the suffix tree for clone detection in large scale systems. The similarity lying in the control flow graphs (CFGs) and abstract syntax trees (ASTs) owing to their control flow restricts the clone detection process. The inner details of the code are masked by the techniques [3-6].

*Corresponding author : ali.javed@uettaxila.edu.pk

Sheneamer and Kalita [7] proposed a hybrid code clone detection technique using coarse-grained and fine-grained features to detect the gapped clone. The coarse grain approach aimed at improving the precision and the fine grain approach improved the recall. However, complex modifications in two source codes cannot be detected except white spaces, variable and function names. Keivanloo et al. [8] proposed a threshold free approach for detecting the gapped clones at method granularity. Ashish [9] proposed a technique to categorize the clones into separate clusters. K-mean clustering technique was used to separate the clones automatically in each cluster. However, there exists a possibility of clone presence in other clusters. Higo and Kusumoto [10] presented a similar approach for automatic clone detection in the fragments of codes during code enhancement or bug fixing. However, two different code files were compared line by line to detect the clones. Kanagalakshmi and Suguna [11] proposed a technique of code clones detection based on levenshtein distance for static and dynamic websites. Singh and Kaur [12] proposed a system based on Euclidean distance to uncover structural clones of type-4 among the several simple clones. Weighted frequent item set mining was exploited to find similar code fragments. Clones having weight support greater than the predefined threshold were selected. Similar fragments were then clustered together to indicate high level clones. This technique [12] is beneficial in terms of reducing the number of repeated patterns, maintenance cost redundancy, code and model refactoring. Kamiya [13] exploited frequent item set mining algorithm upon the sequence of execution of functions in order to detect the gapped clones. The presence of similar code patterns in those items sets having weighted support count less than predefined threshold limits the detection of clones. Rajakumari and Jebarajan [14] proposed a method to identify and evaluate the code clones through data mining techniques. The evaluation stage successfully filtered the valuable clones while discarding the risky ones. Baxter et al. [15] proposed a tool for detecting code clones by finding similar abstract syntax trees (AST). Gplag method [16] based on program dependent graphs was used to improve the plagiarism detection.

Abdel-Aziz et al. [17] presented a technique to extract precise clones using the differential file comparison algorithm. Qu et al. [18] proposed a framework of pattern mining to detect the code clones in large scale software systems. Graph based mining was used in combination with spatial space analysis to accommodate the simple changes and complex modifications effectively. Moreover, pattern analysis approach improved the performance while reducing the computational cost and search complexity. However, this technique only ensures manual switching between the lossy and lossless spatial search although the underlying redundancies should be detected automatically. Su et al. [19] presented a method to identify the codes having similar behavior on the execution time. Okutan and Yildiz [20] exploited a kernel matrix to compute the relationship between code similarity and failure that shows the degree of similarity in the structural clones.

Tian et al. [21] proposed a thread related system calls based approach for plagiarism detection in multithreaded programs using the dynamic birth marking. Maskeri et al. [22] emphasized on the violation of copyrights via detecting plagiarism in the copyright code through mining the software store houses. Flores et al. [23] applied the latent semantic analysis approach to address the issue of cross language duplication in reused code. Stojanovic et al. [24] proposed a metric based approach that consider the procedures from two codes for similarity detection. Software metrics based on the language constructs were defined through high level language and compiler optimization techniques. However, the use of compiler optimization metric based approach in binary codes lead to low recall. Yuan et al. [25] designed a tool for the analysis and representation of detected clones to facilitate the developers for effective management and understanding of clones. In addition, this tool also assists in clones refactoring. Vale et al. [26] presented a repository of bad smells and related refactoring methods for software product line. Ouni et al. [27] presented a search based approach to improve the automation process in code refactoring. The development history of the software along with the semantics and the structural information was used for process improvement. Yamashita and Moonen [28] examined how code smelling reflects the parameters affecting the maintainability of the software based on an empirical study. Hermans et al. [29] proposed an algorithm for clone detection in the spreadsheet that leads to error generation and data loss. Hauptmann et al. [30] introduced a mechanism for the detection of code smells in the tests written in the natural language. The extent of code smell was also determined using the specified measures. Li et al. [31] detected the copy and paste code fragments in the code.

This paper presents a code clone detection method based on a defined mechanism for threshold computation. The proposed technique is based on a similar assumption that codes having the same control flow may differ logically. The motivation is to minimize the level of abstraction to zero while considering the inner details of the codes. The redundancy prevailing due to code clones increase the computational time and cost to a considerable factor. Similarly, the presence of code clones make reusability and maintainability a difficult practice to pursue. The proposed method consists of two stages. The threshold value is computed by subtracting the number of lines containing the unique identifiers from the number of lines of the entire code. The block of code free of any unique identifier, i.e., variables and keywords are retained for further processing. Inner details of the code are considered

by extracting the tokens from the block of code based upon the defined delimiters. Tokens of two source code files are compared on the basis of lexical analysis (CLA). Similar tokens are regarded as clones and error rate is computed. Comparison on the basis of conditional block alleviates the problem of considering the code fragments having same control flow as clones. For comparison on the basis of conditional block (CCB), conditional clauses are extracted from the block of code based on two relational operators (i.e. < or >). Threshold value is computed on the basis of number of conditions. Statements within the conditional block are extracted and their count is maintained. Statements of the two source files are compared according to the similarity of conditions. Two conditional blocks are regarded as clones if their conditional clauses contain the same operator (< or >) as well as similar number of statements which are logically equivalent. The main contributions of the proposed technique are:

1. Elimination of the high level abstraction while unmasking inner details of the codes.
2. Uncover clones irrespective of control flow.
3. Comparison on the results of Lexical Analysis (CLA)
4. Comparison of Conditional Blocks (CCB)
5. Threshold value calculation for both phases.

The rest of the paper is organized as follows: Section 2 presents the proposed method of code clone detection. Experimental results and discussion are provided in Section 3. Finally, Section 4 concludes the proposed work along-with possible future extension.

2. Proposed Method

The proposed method consists of two phases. The first phase provides a comparison on the results of lexical analysis (CLA). CLA aims to unmask the inner details of the code, thereby removing the abstraction. CLA compares the extracted tokens and evaluates the results based on a defined threshold. The second phase "Comparison of the conditional blocks" (CCB) intends to alleviate the problem of declaring the code fragments as clones on the basis of similar condition in control flow graph (CFG). CCB aims to compare statements on the basis of similarity of conditions and evaluate the results according to the defined threshold. Shown in Fig. 1 is the block diagram of the proposed method.

2.1. Threshold Calculation

Automatic threshold computation is performed in the proposed method to address the limitation of existing code clone detection methods that are unable to provide any defined mechanism for threshold calculation. Existing approaches [4, 13, 18, 32, 33] use either random or any hard coded threshold value. The performance of the existing methods adopting these two criteria's for threshold selection degrades significantly on a large and diverse dataset of codes. The proposed method resolves this issue by automatically computing the threshold value as follows:

$$Th_A = loc_A - ui_A \quad (1)$$

Where Th_A is the threshold value computed for the code, loc_A refers to the total number of lines in the source code and ui_A represents the total number of unique identifiers including the keyword and variables extracted from the imported code file.

2.1.1. Unique identifiers

Unique identifiers are characterized as the specified keywords of a programming language which are used as reserved words within the code. The keywords cannot be manipulated while solving any programming problem. Reserve words may vary from language to language but their utility is same across multiple languages. Identifiers of Java language specified here include: (java, static, void, public, private, protected, int, char, string, float, double, long, unsigned, flag, args, class, extend, system, out, println, print, scanner, nextInt, next, new, array list, if, do, for, array, return, switch, case, default, break, continue). The total count of extracted identifiers (both keywords and variables) is also stored and maintained throughout the process.

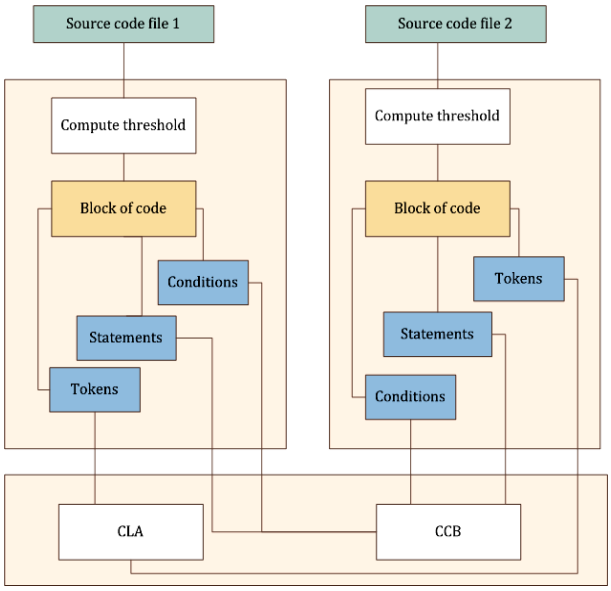

Fig. 1: Block diagram of proposed code clone detection method

2.1.2. Variables

Variables are the user-defined terms that are used in the code to perform a certain task. Variables can be any letter or word defined in the code. The collection of unique identifiers and variables are summed up and subtracted from the total lines of code to calculate the threshold. The computed threshold defines the size of the code block free of all unique identifiers i.e. keywords and variables. The final comparison for clone detection is based on those lines

of code within the code block extracted on the basis of defined threshold.

2.2 Phase 1: Comparison on the results of Lexical Analysis (CLA)

CLA includes the lexical analysis which results in the extraction of tokens. Each step of lexical analysis is discussed in this section.

2.2.1 Tokenization

The first step involves creating tokens on the basis of predefined delimiters. The delimiters can be semi colon (;), Brackets ( ), { }, [ ], Special characters (!, &, ^, $, #, @, \) and Operators (<, >, =, -, +, *, /, %), etc. The process of extracting tokens from the imported code is called tokenization. The total count of tokens is stored and maintained throughout the process. The major step involved in the lexical analysis is the extraction of tokens according to the pre-defined delimiters during the process of tokenization.

2.2.2 Clone detection comparison

A comparison of the two source code files in terms of their similarity and difference is performed on a defined mechanism. The extracted tokens for both files and their counts are used for comparison. Clone detection comparison is performed token by token with the two imported files. If the two codes under consideration are similar then clone is declared otherwise the two codes differ and cannot be regarded as clones. Moreover, the total number of detections is also maintained.

2.3 Phase 2: Comparison of Conditional Blocks (CCB)

CCB is the comparison of the extracted conditional clauses. The entire block of conditions is extracted and comparison is performed on each clause. The two extracted conditional blocks from the two intended codes are then further processed to extract the conditions and statements.

2.3.1 Conditional block

Conditional blocks consist of chunk of statements used for decision making. In the first stage the entire conditional block is stored in a new file and then retrieved for further processing and evaluations when required. The conditional block dictates the flow of the program as well. Two programs may carry the same conditional blocks and their flow may be similar. However, some conditions in the same conditional block may be identical even though statements differ. Similarly, some conditions may differ in the same conditional blocks. The count of the conditions is maintained and threshold is set with count greater than zero.

2.3.2 Conditions

Conditions are the decision points in a program that control the flow of the program. If a condition is true it may enter in another loop containing further conditions. Within a conditional block "if" and "else" are the driving sources which may proceed or terminate a program depending on the obtained results. Conditional clauses are accessed by checking the presence of operators i.e. "<" or ">". The count of the conditions is stored and maintained as a threshold for further processing. If the conditional clauses of two conditional blocks are matched on the basis of *less than* (<) or *greater than* (>) operators, then comparison is performed on the basis of statements.

2.3.3 Statements

Statements are the actions performed on the validity or absurdity of any condition. The comparison is performed statement by statement in the two conditions of the imported code files. The results of comparison are evaluated in terms of clone detection. The conditions are compared against the statements. If the number of the two entities differs, then a sequential comparison is performed against each statement. If two conditional clauses are same then statements are compared. In case the statements of the two programs are identical then it is declared as a clone. If the statements differ as a result of comparison then it is not regarded as a clone.

Major steps involved in the comparison of conditional clauses are summarized as follows:

Step 1: Extraction of conditional block from the two imported code files.

Step 2: Extraction of conditions from the two conditional blocks.

Step 3: Extraction of statements from the extracted conditions.

2.3.4 Threshold calculation

Threshold of the CCB phase is dependent upon the number of conditions. The comparison on the extracted block of conditions yields individual conditions. The threshold is based on the number of conditions and statements lie above zero. Total count of conditions is maintained and further comparison is performed on the defined threshold computed as follows:

$$th = \sum n_C \qquad (2)$$

Where *th* is the threshold, and n_c is the number of extracted conditions.

If *th* is greater than or equal to 1, then predicate clauses of the conditional blocks are compared. However, conditions are not compared if *th* is equivalent to zero.

2.3.5 Clone detection comparison

A comparison of the two source code files in terms of their similarity and difference is performed on the block of conditions. Comparison is performed statement by statement along the two codes after the approval of the

validity of the conditional clauses. If the statements of the two codes and their number of statements under consideration are similar then it is declared as a clone otherwise the two codes differ. The number of similar tokens from Phase 1: CLA and similar statements from Phase 2: CCB are maintained for computation of error rates for both phases. Clones are maintained and further used to calculate the error rate. The algorithm of the proposed method is provided in Table 1.

Table 1: Defined mechanism for threshold calculation

Input: Source Code
Output: Clone Detections (Average Error rate)
Count lines of code; loc_A, loc_B
Extract Unique identifiers
Count unique identifiers; ui_A, ui_B
Calculate threshold
$Th_A = loc_A - ui_A$
$Th_B = loc_B - ui_B$
Extract rest of the lines of code uptill the $Th_A$and Th_B
Phase 1: CLA
Extract tokens on the basis of lexical analysis
Count total number of tokens, n_t_A and n_t_B
if $n_t\_A > n_t\_B$
Compare the remaining tokens till n_t_A
if $tokens_A == tokens_B$
Tokens are similar
else
Tokens are not similar
else Compare the remaining tokens till n_t_B
if $tokens_A == tokens_B$
Tokens are similar
else
Tokens are not similar
Phase 2: CCB
Extract conditional blocks
CB_A and CB_B
Extract conditional clauses C_A and C_B from CB_A and CB_B,
Calculate threshold
$thresh_A = n\_C_A$
$thresh_B = n\_C_B$
$Count\_S_A = statements_A$
$Count\_S_B statements_B$
if $thresh_A \&\& thresh_B > 0$
if $C_A == C_B$
if $Count\_S_A > Count\_S_B$
Compare statements of the blocks till $Count\_S_A$
If $statements_A == statements_B$
Cloned
Else
Not cloned
Else
Compare statements till $Count\_S_B$
else
Conditions are not same

3. Results and Discussion

This section provides a discussion on the experiment designed to evaluate the performance of the proposed method. The results of the experiment are also reported and compared with the existing methods. Moreover, the details of the dataset is also provided.

Sorting is considered as a fundamental process for algorithm design. Sorting is defined as the process of rearranging elements in specific sequence either increasing or decreasing. The sorting mechanism holds significant importance in terms of cost reduction for the purpose of accessing data. A customized dataset of different sorting algorithms is used to evaluate the performance of the proposed method. The dataset consists of sorting algorithms such as bubble, selection, insertion, shell, merge, quick, counting, bucket and heap sort. Since each sorting algorithm provides a sorted list of elements in either increasing or decreasing order but the way that each algorithm sorts the elements can vary. In general sorting techniques can be classified into two major categories i.e., comparison based sorting and non-comparison based sorting. Comparison-based techniques [34] include sorting the results achieved after comparison or a couple of iterative comparisons as in selection sort, bubble sort quick sort, heap sort, insertion sort and merge sort. Non comparison-based techniques [34] include counting sort, bucket sort, etc.

3.1 Performance Evaluation

Performance evaluation of the proposed algorithm is tested on different sorting algorithms to measure the effectiveness of our method.

An experiment is designed to process the two codes of sorting algorithms for code clone detection at the same time. Bubble sort and selection sort have the same control flow but they differ in terms of their logic for sorting purpose. Moreover, both are comparison-based algorithms. File 1 contains source code of the bubble sort algorithm, whereas, file 2 contains the source code of selection sort algorithm. In bubble sort algorithm, each element is compared to the rest of the elements in the array and swapped if they are not in order. Selection sort algorithm performs array sorting by repeatedly finding the minimum or maximum element from the unsorted part and placing it at the beginning. The algorithm maintains two sub arrays in a given array.

Total number of lines of code is counted for both files. File 1 contains 32 lines of code of bubble sort algorithm, whereas, file 2 contains 28 lines of code of selection sort algorithm. In the next step unique identifiers are extracted from the file 1 that is 24 in this case. Threshold value for file 1 is 8. Threshold value decides that these lines of code are left which do not contain any unique identifier i.e., keywords and variables. The code blocks within the computed threshold contains the functionality for sorting of elements. Similarly, unique identifies from file 2 is also extracted that are 13 in the present study. Threshold value for file 2 is 15. These code blocks are further used for comparison on the basis of lexical analysis (CLA) and

Table 2: Average error rate of detections for both phases

File 1	File 2 Sorting Algorithms	Detection (CLA)	Detection (CCB)	Average Error Rate (%)
Bubble Sort	Selection Sort	0	1	8.33
	Quick Sort	3	0	0.25
	Merge Sort	0	0	0
	Insertion Sort	40	1	5.55
	Shell Sort	2	1	5.35
Selection Sort	Quick Sort	0	0	0
	Merge Sort	0	0	0
	Insertion Sort	0	1	4.165
	Shell Sort	0	2	8.33
Insertion Sort	Merge Sort	0	0	0
	Quick Sort	0	0	0
	Shell Sort	0	1	5.55
Bucket sort	Bubble Sort	0	0	0
	Selection Sort	0	0	0
	Merge Sort	0	0	0
	Quick Sort	0	0	0
Counting sort	Bubble Sort	0	0	0
	Selection Sort	0	0	0
	Insertion Sort	0	0	0
Heap sort	Bubble Sort	0	2	11.11
	Selection Sort	0	2	6.25
	Insertion Sort	0	2	6.25

comparison on the basis of conditional block (CCB). For CLA all the extracted tokens based on delimiters are compared for both files and number of detections is maintained. Since two files have same conditional clauses, so statements extracted for both blocks are compared and number of detections is maintained for CCB. Tokens are extracted from both files i.e. file 1 and file 2 according to the predefined delimiters. All the generated tokens of both files are compared with each other. No similar tokens are detected so similarity is considered as false. Further comparison is performed on the extracted conditional blocks from which conditional clauses and statements are extracted. Both conditional blocks i.e. CB_1 and CB_2 extracted from file 1 and file 2, respectively. Conditional clauses extracted from both conditional blocks contain the same operator. Threshold value is calculated according to the number of conditions. So, threshold value for both conditional blocks is greater than 1. Hence statements within the conditional blocks are compared with each other and number of similar statements is recorded. Finally error rate is calculated for both phases using equations (3), (4) and (5). No detections are observed in any of the phase, therefore, average error rate is 0.

$$err\_rate_{CLA} = 100 - \frac{detection_{CLA}}{total\_no\_of\_comparisons} * 100 \quad (3)$$

$$err\_rate_{CCB} = 100 - \frac{detection_{CCB}}{total\_no\_of\_comparisons} * 100 \quad (4)$$

$$avg\_err = \frac{err\_rate_{CLA} + err\_rate_{CCB}}{2} \quad (5)$$

Where $err\_rate_{CLA,}$ err_rate_{CCB} and *avg_err* represents the error rate of CLA, CCB, and average respectively.

The frequency of tokens can limit the proposed technique to detect the repeated tokens with better accuracy. The tokens that repeat frequently will be displayed once in the output. Table 2 depicts the average error rate of comparison based sorting algorithm files. Fig. 2 shows the graphical representation of the results of comparison.

3.2 Performance Comparison

Performance of the proposed method is compared against existing state-of-the-art methods [4, 7, 16, 18, 24, 31]. Results of comparison with the existing techniques are provided in Table 3. It can be observed from Table 3 that the proposed method outperforms the existing state-of-the-art methods of code clone detection.

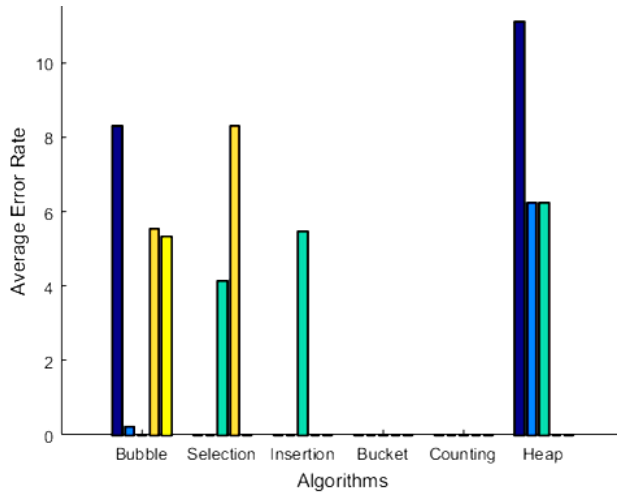

Fig. 2: Performance Evaluation of proposed method on sorting algorithms

Table 3: Performance comparison with existing techniques

Code Clone Detection Methods	Accuracy
Li et al. [4]	87.1%
Sheneamer and Kalita [7]	48.89%
Liu et al. [16]	91.5%
Qu et al. [18]	79%
Stojanović et al. [24]	43%
Li et al. [31]	83.5%
Proposed Method	97.22%

4. Conclusions

This paper presents a defined mechanism for the calculation of threshold in clone detection. The proposed technique is divided into two major phases. First phase involves the comparison on the results of lexical analysis i.e. tokenization. The extracted tokens and unique identifiers are used for threshold calculation. The final comparison of the two codes is performed on the basis of threshold. The second phase is the comparison of the conditional blocks in which conditions and statements are compared based on a defined threshold. The results of the two phases indicate the confirmation of either existence or non-existence of the clone. The proposed technique is significant in terms of eliminating high level of abstraction, detecting the clones irrespective of their control flows. The defined mechanism for threshold computation guarantees the contribution of every single token in clone detection. The average accuracy of 97.2% signifies the effectiveness of the proposed method for code clone detection. The proposed technique can be further extended by keeping track of the frequency of particular token that appears in the source code file.

References

[1] P. Pradhan, A. K. Dwivedi and S. K. Rath, "Detection of design pattern using graph isomorphism and normalized cross correlation", 8th Int. Conf. Contemp. Comput., pp. 208–213, 2015.

[2] W. Li, D. Li, C. Qiu and J. Hou, "Efficient metric vector-based code clone detection using function-calling tree", Int. J. Hybrid Inf. Technol., vol. 8, no. 11, pp. 139–150, 2015.

[3] S. Gupta and P. C. Gupta, "A novel approach to detect duplicate code blocks to reduce maintenance effort", Int. J. Adv. Comput. Sci. Appl., vol. 7, no. 4, pp. 311–314, 2016.

[4] W. Li, H. Saidi, H. Sanchez and M. Sch, "Detecting similar programs via the Weisfeiler-Leman graph kernel", Proc. of 15th Int. Conf. on Software Reuse, pp. 315–330, 2016.

[5] R. Tekchandani, R. Bhatia and M. Singh, "Semantic code clone detection for Internet of things applications using reaching definition and liveness analysis", J. Supercomput., pp. 1–28, 2016.

[6] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing", J. Softw. Evol. Process, vol. 26, no. 8, pp. 747–769, 2014.

[7] A. Sheneamer and J. Kalita, "Code clone detection using coarse and fine-grained hybrid approaches", IEEE 7th Int. Conf. Intell. Comput. Inf. Syst. ICICIS 2015, pp. 472–480, 2016.

[8] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous Java repository", IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, Proc. SANER 2015, pp. 201–210, 2015.

[9] A. Ashish, "Clones clustering using K-means", 10th Int. Conf. Intell. Syst. Control, pp. 1–6, 2016.

[10] Y. Higo and S. Kusumoto, "How often do unintended inconsistencies happen? Deriving modification patterns and detecting overlooked code fragments", IEEE Int. Conf. Softw. Maintenance, pp. 222–231, 2012.

[11] K. Kanagalakshmi and R. Suguna, "Software refactoring technique for code clone detection of static and dynamic website", Int. J. Comp. Applications", vol. 107, no. 12, pp. 1–10, 2014.

[12] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software", Ain Shams Eng. J., 2016.

[13] T. Kamiya, "An Execution-Semantic and Content-and-Context-Based Code-Clone Detection and Analysis", Software Clones (IWSC), IEEE 9th Int. Workshop, pp. 1–7, 2015.

[14] K. E. Rajakumari and T. Jebarajan, "A novel approach to effective detection and analysis of code clones", Third Int. Conf. Innov. Comput. Technol., pp. 287–290, 2013.

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, "Clone detection using abstract syntax trees", Proc. Int. Conf. Softw. Maint. (Cat. No. 98CB36272), pp. 368–377, 1998.

[16] C. Liu, C. Chen, J. Han and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis", Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. data Min., pp. 872–881, 2006.

[17] R. M. Abdel-Aziz, A. E. Aboutabl and M. S. Mostafa, "Clone detection using DIFF algorithm for aspect mining", Int. J. Adv. Comput. Sci. Appl., vol. 3, no. 8, pp. 137–140, 2012.

[18] W. Qu, Y. Jia and M. Jiang, "Pattern mining of cloned codes in software systems", Inf. Sci. (Ny)., vol. 259, pp. 544–554, 2014.

[19] F. H. Su, J. Bell and G. Kaiser, "Challenges in behavioral code clone detection", IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2016, vol. 2, pp. 21–22, 2016.

[20] A. Okutan and O. Taner Yildiz, "A novel kernel to predict software defectiveness", J. Syst. Softw., vol. 119, pp. 109–121, 2016.

[21] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs", J. Syst. Softw., vol. 119, pp. 136–148, 2016.

[22] G. Maskeri, D. Karnam, S. A. Viswanathan and S. Padmanabhuni, "Version history based source code plagiarism detection in proprietary systems", IEEE Int. Conf. Softw. Maintenance, ICSM, pp. 609–612, 2012.

[23] E. Flores, A. Barron-Cedeno, L. Moreno and P. Rosso, "Cross-language source code re-use detection using latent semantic analysis", J. Univers. Comput. Sci., vol. 21, no. 13, pp. 1708–1725, 2015.

[24] S. Stojanović, Z. Radivojević, and M. Cvetanović, "Approach for estimating similarity between procedures in differently compiled binaries", Inf. Softw. Technol., vol. 58, pp. 259–271, 2015.

[25] Y. Yuan, F. Zhang, and X. Su, "CloneAyz : An approach for clone representation and analysis", Inf. Sci. Control Engg. (IEEE), pp. 252-256, 2016.

[26] G. Vale, E. Figueiredo, R. Abilio, and H. Costa, "Bad smells in software product lines: A systematic review", Proc. of 8th Brazilian Symp. Softw. Components, Archit. Reuse, pp. 84–94, 2014.

[27] A. Ouni, M. Kessentini, S. Bechikh and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," Software Quality Journal , vol. 23, no. 2. 2015.

[28] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?", IEEE Int. Conf. Softw. Maintenance, ICSM, pp. 306–315, 2012.

[29] F. Hermans, M. Pinzger and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas", Empir. Softw. Eng., vol. 20, no. 2, pp. 549–575, 2015.

[30] B. Hauptmann, M. Junker, S. Eder, L. Heinemann, R. Vaas and P. Braun, "Hunting for Smells in Natural Language Tests", Proc. of Int. Conf. on Software Engg., no. 1, pp. 4–7, 2013.

[31] Z. Li, S. Lu, S. Myagmar and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code", IEEE Trans. Softw. Eng., vol. 32, no. 3, pp. 176–192, 2006.

[32] H. Kaur and R. Maini, "Identification of recurring patterns of code to detect structural clones", Proc. of 6th Int. Adv. Comput. Conf., pp. 398–403, 2016.

[33] M. Abdelkader and M. Mimoun, "Clone detection using time series and dynamic time warping techniques", Third World Conf. Complex Syst., pp. 1–6, 2015.

[34] J. Alnihoud and R. Mansi, "An enhancement of major sorting algorithms", Int. Arab J. Inf. Technol., vol. 7, no. 1, pp. 55–62, 2010.