Code Similarity Detection by Computing Block Level Feature Metrics

Sudhamani M^{#1}, Lalitha Rangarajan^{#1}

^{#1} Department of Studies in Computer Science, University of Mysore, Mysuru – 570 006

ABSTRACT

All software systems are subject to continuous evolution and maintenance activities in order to eliminate defects and extend their functionalities. We need to deal with code duplication in order to prevent some problems that will appear when trying to adapt to the changes that are imminent in a real software system. In this paper, source code is divided into different types of blocks to detect duplicate segments. The proposed method aims at detecting segments that have similar functional blocks. We have experimentally validated the proposed technique in terms of accuracy. The method can efficiently identify structural and functional similar clones.

Key words: Code clones, Structural similarity, Block feature metric, Similarity value

Corresponding Author: Sudhamani M

INTRODUCTION

Code plagiarism is one of the realistic problems in academic area especially in programming courses as it does not require much effort to do copy paste and a little modification [10]. As software is a key resource of present complex systems, the size of software systems is continuously increasing. Code duplication increases the size of code, fault occurrence and resource requirements. Current research shows that, source code in many large scale applications contain code clones. Poor structure may occur due to poor initial design and lack of necessary features in the programming language, but is often due to repeated, poorly planned changes made to programs. Programs undergoing development and maintenance often have a lot of duplicate code. Software is a collection of program files. These programs are collection of different types of program statements like declarative statements, computational statements, logical statements, input-output statements and control statements [16]. Execution of program depends on structure of control statements present in the program [5]. We divide program file into several blocks like, loop block, condition block, function definition block and function call. Presence of different types of blocks for the programs in figure 1 is shown in figure 2. We compute block-wise features like number and type of statements present in the blocks and types and count of operators present in the blocks. Then these features are compared to find similar code fragments.

<pre> #include <stdio.h> int main() { long array[100], n, c, d, swap; printf("Enter number of elements\n"); scanf("%ld", &n); printf("Enter %ld integers\n", n); for (c = 0; c < n; c++) scanf("%ld", &array[c]); bubble_sort(array, n); printf("Sorted list in ascending order:\n"); for (c = 0; c < n; c++) printf("%ld\n", array[c]); return 0; } void bubble_sort (long list[], long n) { long c, d, t; for (c = 0; c < (n - 1); c++) { for (d = 0; d < n - c - 1; d++) { if (list[d] > list[d+1]) { /* Swapping */ t = list[d]; list[d] = list[d+1]; list[d+1] = t; } } } } </pre>	<pre> /* Bubble sort code */ #include <stdio.h> int main() { int array[100], n, c, d, swap; printf("Enter number of elements\n"); scanf("%d", &n); printf("Enter %d integers\n", n); for (c = 0; c < n; c++) scanf("%d", &array[c]); for (c = 0; c < (n - 1); c++) { for (d = 0; d < n - c - 1; d++) { if (array[d] > array[d+1]) { swap = array[d]; array[d] = array[d+1]; array[d+1] = swap; } } } printf("Sorted list in ascending order:\n"); for (c = 0; c < n; c++) printf("%d\n", array[c]); return 0; } </pre>
---	---

Fig 1: Different versions of sorting

Fig 2: Block wise diagram of programs in figure 1

Duplication is usually generated by copy-paste activities, as it is easy and fast. A new feature that is same as an existing feature is implemented by copying and pasting code fragments, followed by some modifications. Programmers also introduce duplication by reimplementing functionality that already exists in the system, but of which they are not aware. Several clones in real programs have approximately identical functionality, have many matching lines, but nevertheless differ in the order of the matching lines. Duplication in source code, irrespective of the reason why it exists, is the cause for many difficulties. It increases the size of the code and makes software maintenance more strenuous. If an improvement or bug fix is done on a code fragment, it may be necessary to search for clones of the fragment to perform the corresponding modification. Other possible reasons for code clones other than copy paste are listed below.

- Code reutilization by copying existing solution
- Failure to identify or use abstract data types
- Performance enhancement
- Accidents [2]

In the literature, different types of clones are represented based on the level of similarity between the code fragments.

Type I: Exactly similar code fragments with changes only in comments and spaces.

Type II: Exactly identical code fragments with changes in variables and literals.

Type III: Copied segments with further modifications such as reordered statements added or removed statements.

Type IV: Two program instances that perform the same function but are implemented by different syntactic variants [6].

The proposed method can identify functionally similar control structures. Our method is evaluated on two open-source systems by manually inspecting the detected clones. We compute source code features that are being used to detect the extent of similarity between source code fragments. The four different types of source code features are: (1) Textual features that are on the level of textual representation of the program like length of the statements, length of identifiers, number of identifiers and number of source lines. (2) Structural features that defines structural properties of the program. (3) Syntactic structural features that are computed by transforming the source code into abstract syntax trees and (4) Semantic structural features, obtained from detailed code representations like call-graph features, program dependence graph features like value and reference dependencies. The method, proposed method is close to feature (4). However the computation of metric in the proposed method considers structural features of blocks of code. Program blocks categorization is shown in figure 2. The proposed method doesn't require any transformation. Control blocks are of three types - iterative blocks (set of statements included in iterative statements like for, while, do-while and repeat-until [4]) and decision blocks or condition blocks (the statements included in conditional statements like if, if-else, switch-case) and function definition block. We also consider function call as a block of single statement (execution flow depends on function call) even though it is a single statement and count for the number of operators and operands used in actual parameter. The paper is organized into five sections. Section 2 discusses the related work. Proposed system is described in section 3. Section 4 gives experimental analysis. Conclusion and future work are discussed in the last section.

RELATED WORK

In recent years, the detection of clones has drawn considerable attention. An abundant number of clone detectors have been proposed in the literature. Based on the comparison techniques, clone detection techniques are categorized into the following categories such as string based, token based, tree based and graph based to detect code clones [7][8].

String based approach: In string / text based approach source code lines are considered as a sequence of characters and are compared with each other. In some cases raw source codes are directly compared, and in some other cases transformation and normalization techniques are used before comparison. Baker's Dup tool focuses on locating duplication or near-duplication in a large software system. Dup converts source code into tokens and line based string matching algorithm is applied to find duplicates [13, 14]. Dup detects parameterized matches and it brings out results in the form of both report and scatter plots. This tool does not support exploration and navigation through the duplicated code. The tool cannot detect code written in different styles and accuracy of the tool is low. Johnson used incremental hash function combined with sliding window method to find clones of different lengths. He applied Karp-Rabin fingerprint algorithm on source code [7, 9]. Ducasse et al. normalized the source lines and used dynamic pattern matching to compare strings. SDD is a string-based tool which uses n-neighbor approach in finding near-miss clones. Marcus and Maletic uses latent semantic indexing (LSI), an information retrieval technique to statistically analyze the software systems and find semantic similarities in source code [16, 17]. Cordy et al. used an island grammar to find near miss clones in HTML web pages. Text based approaches are fast and less expensive compared to other approaches. However, string based approach is less expensive but cannot detect variable rename and reorder statements.

Token based approach: In token-based approach, source text is transformed / lexed / parsed into stream of tokens using lex / yacc scanners. The sequence of tokens is scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones [7]. Brenda Baker developed an effective token-based clone detection tool Dup [14, 16, and 18]. In Dup, lexer is used to tokenize the source code. Then these token sequences are compared using suffix tree algorithm. CCFinder, CP-Miner, Gemini and RTF are prominent token based tools. Kamiya et al. used source normalization in CCFinder [18]. Gemini visualizes near miss clones using scatter plots and RTF tokenization with suffix tree method to detect code clones [17]. Frequent subsequence data mining technique is adopted in CP-Miner to find similar stream of tokens [25]. Winnowing, JPlag and SIM [5, 22] are familiar token based plagiarism detection tools. CCFinder and Dup are unreliable to handle reordered statements. CP-Miner handles this situation. Compared to text-based approaches, a token-based approach is usually resistant against code changes such as formatting, spacing and variable renaming. The technique detects Type-1 and Type-2 clones but found it difficult to find near miss clones.

Tree-based approach: Abstract syntax tree based approach converts source code into an abstract syntax tree and traverses the tree for finding similar sub-trees [21]. The parse tree or AST contains the complete information about the source code. The result obtained through this comparison is quite efficient but it is very difficult and complex to create an abstract syntax tree and the scalability is also not good [21]. CloneDR is the prime AST based tool developed by Baxter et al. A compiler generator is used to generate an annotated parse tree and compares its sub-trees by characterization metrics based on a hash function through tree matching to detect gapped clones and reordered statements. ccdiml tool fails to handle

variable rename whereas CloneDR does. In Bauhaus, the ASTs are represented in IML (Intermediate Language) [24] and comparison is done on IMLs rather than ASTs. Yang [23, 5] proposed a similar approach to find the syntactic differences between two versions of the same programs by generating a variant of parse tree for both the versions and then applied dynamic programming approach to search similar sub-trees. Wahler et al. found exact and parameterized clones by converting AST of a program to an XML representation [20, 5] and then a data mining frequent itemset technique was applied in the XML representation of the AST to find clones. Evans and Fraser proposed a method to identify exact and near miss clones with gaps. AST-based approach can identify exact, near miss and gap clones apart from disregards to the information about variable rename. AST does not show the data flow information, therefore it cannot handle statement reorder and control replacement. PDG-based techniques can overthrow these drawbacks [5].

PDG-based approaches: Program Dependence Graph (PDG) [16] contains the control flow and data flow information of a program and hence carries semantic information. Once a set of PDGs are obtained from a subject program, isomorphic sub-graph matching algorithm is applied for finding similar sub-graphs which are returned as clones. PDG based approaches are robust in detecting functional similarities like [5] source code representation of high abstraction than other approaches by considering the semantic information of the source code. One of the leading PDG-based clone detection approaches is Komondoor and Horwitz's PDG-DUP which finds isomorphic PDG sub-graphs using program slicing. Krinke used k-length patch matching iterative approach for detecting maximal similar sub-graphs [30]. GPLAG [31] is a PDG-based plagiarism detection tool. Chen et al. proposed a PDG-based technique for code compaction taking into account syntactic structure and data flow. PDG-based approaches can effectively handle reordered statements, insertion and deletion of code, intertwined code, and non-contiguous code, but this method is not scalable to large size programs and finding isomorphic sub-graphs itself is a NP-hard complexity [33, 5].

Metrics-based approaches: Metrics-based approaches gather different metrics from source code and compare these metrics instead of comparing code directly. There are several clone detection techniques that use various software metrics for detecting similar codes. Mayrand et al. [32] calculated metrics like the number of lines of source, number of function calls contained, number of control flow graph (CFG) edges etc, for each function unit of a program. Function units with similar metric values are identified as code clones. Source code is represented as Intermediate Representation Language (IRL) and metrics are calculated from names, layout, expression and control flow of functions. A clone is characterized only as a pair of whole function bodies that have similar metric values. This approach detects function based clones but fails to detect segment-based copy-paste, where frequency of segment based copy-paste is more than function based copy-paste. Patenaude et al. proposed a method to find method-level similarity by comparing metrics such as number of calls from a method, number of statements, McCabe's cyclomatic complexity, number of global and local variables. They define these metrics for Java language and extend the IBM Datrix tool to support Java in software quality assessment.

Kontogiannis et al. built an abstract pattern matching tool to identify probable matches using Markov models to compute similarity between programs. He proposed two ways to detect clones, one is direct comparison of metric computed from AST tree (number of fanout (functions called), the ratio of input / output variable to the fanout, McCabe cyclomatic complexity, modified Albercht's function point metric and modified Henry-Kafaura's

information flow quality metric). Second method uses a Dynamic Programming (DP) technique at statement level. In dynamic programming (DP) approach, the distance between the pair of segments is measured by the least expensive sequence of insert, delete and edit steps required to make one segment similar to the other [32]. Metric-based approach is also used to find duplicated web pages. Di Ducca et al. proposed a method to identify similar static HTML pages by comparing the Levenshtein distances between items in web pages and calculating their degree of similarity. Lanubile et al. proposed a semi automated method to detect clone script functions, using eMetrics tool to retrieve the potential function clones. Davey et al. detects exact, parameterized and near-miss clones by using neural networks on features retrieved [5, 26].

Hybrid-based approach: These techniques use a composite / fusion of syntactic and semantic characteristics. Leitao [17] provides a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques using call graphs method in combination with specialized comparison functions [1]. In the approach by Koschke et al. the tokens of the AST-nodes are compared using a suffix tree-based algorithm. Hence clones are detected in linear time compared to AST based approach [35]. A function-level clone detection technique is proposed for the Microsoft's new Phoenix framework using AST and suffix trees. Greenan [34] proposed an algorithm to find method level clones on transformed AST using sequence matching algorithm. Jiang et al. proposed a new method to find similar trees by comparing the structural information of AST in Euclidean space and Locality sensitive hashing algorithm was used to cluster the clones [5].

PROPOSED METHODOLOGY

Code clone in a software system refers to similar copies of same instance of source code in the software. The objective of the system is to find similar blocks present in the source code. Similarity is computed by comparing the features of blocks. Features we consider here are structure, order and types of program statements present in the program file. Figure 3 represents the architectural design and steps of the proposed method where each step is explained subsequently.

The different stages in the proposed method are preprocessing, block wise feature metric computation, block level difference matrix computation, similarity value calculation and clustering.

Fig 3: Architecture of the proposed method

Input to the method is a folder containing program files. We have not considered header files, so we separate C files from header files by comparing the file extension and store C files in a separate folder to process. In pre-processing, extra space and comments are removed. Features computed are type of block (condition/ loop/ function definition/general), structure of nested blocks, type of statements (such as input output statements, computation statements etc) and number, type of operators (arithmetic operators, relational operators and logical operators) contained in the block. This information are stored in a table and named as Block Feature Metric Table (BFMT). BFMT for code segments in figure 1 are shown in table 1 and 2 respectively.

Table1. Block Feature Metric Table for first code segment in figure 1

Block_no	Block_type	block_id	L	C	Computation statements	no_AO	no_RO	no_LO	No_fun_call
1	Fun_def	3	2	0	0	0	0	0	1
2	Loop	1	0	0	0	2	1	0	0
3	F_call	4	0	0	0	0	0	0	0
4	Loop	1	0	0	0	2	1	0	0
5	Fun_def	3	2	1	0	0	0	0	0
6	Loop	1	1	1	0	3	1	0	0
7	Loop	1	0	1	0	4	1	0	0
8	Cond	2	0	0	3	6	1	0	0

Table 2. Block Feature Metric Table for second code segment in figure 1

Block_no	Block_type	block_id	L	C	Computation statements	no_AO	no_RO	no_LO	No_fun_call
1	Fun_def	3	5	1	0	0	0	0	0
2	Loop	1	0	0	0	2	1	0	0
3	Loop	1	1	1	0	3	1	0	0
4	Loop	1	0	1	0	4	1	0	0
5	Cond	2	0	0	3	6	1	0	0
6	Loop	1	0	0	0	2	1	0	0

Block Distance Matrix (D) is computed from two Block Feature Metric tables. A row of program 1 (corresponding to a block instance) is compared with every row of program 2. Rows i and j of the program are compared using city block distance formula as $|x_1 - x_2| + |y_1 - y_2|$. For instance, distance between block 4 of program 1 and block 2 of program 2 is computed from the respective rows of corresponding BFMT as: $|0-0| + |0-0| + |0-0| + |2-2| + |1-1| + |0-0| + |0-0| = 0$ and is stored in BDM (4, 2). From the table we compute similar blocks present in two programs. Presence of zero in the cell indicates possible similarity of corresponding blocks.

Similarity between two programs is computed using the formula

$$S = n / (r_1 - r_2) \dots \dots (1)$$

where, s is the similarity value, n is the number of zeros in block distance matrix and r1, r2 are number of blocks in program 1 and program 2 respectively.

Table 2. Block Feature Metric Table for second code segment in figure 1

	FD	LB	LB	LB	CB	LB
FD	5	6	7	9	13	6
LB	9	0	3	3	7	0
FC	6	3	6	6	10	3
LB	9	0	3	3	7	0
FD	3	6	5	7	13	6
LB	8	6	0	5	5	6
LB	10	6	5	0	3	6
CB	16	7	8	6	0	7

Time complexity

Suppose two programs have SL_1 and SL_2 source lines and B_1 and B_2 blocks. Note that the number of blocks (segments) in a program will be far less than the number of source lines ($B \ll SL$). Table 4 shows the detail of major steps in the computation of similarity and the corresponding complexities.

Table 4. Time complexity table

Steps	Complexities
Preprocessing	$\theta(SL_1) + \theta(SL_2)$
BFMT	$\theta(SL_1) + \theta(SL_2)$
BDT	$\theta(B_1 \times B_2)$
Similarity computation	$\theta(B_1 \times B_2)$

Hence total time complexity is maximum $((\theta(SL_1) + \theta(SL_2)), \theta(B_1 \times B_2))$. However, this is polynomial time complexity.

EXPERIMENTAL RESULTS

We conducted experiments on a data set of size 93 programs, which are downloaded from 'flsourcecode' for which versions are created by reordering and changing independent statements, reordering and changing control statements wherever possible, so that outputs of the programs are not affected. Experiments are conducted and shown for similarity values of 7 programs and 19 instances for the proposed method. Table 5 shows the similarity measurements of the programs and figure 4 shows the similarity graph. All programs show higher similarities with only its versions.

We also conducted experiment on our own data set of 93 Students' Lab Programs (SLP) and two projects from Bellon's benchmark dataset for detection of code clones and performance is measured by observing the accuracy. Details of the data set used for experimentation are in table 6. Bellon has intentionally injected 2% of clones to the projects. We manually verified those clones and results are tabulated in table 7.

Table 5. Similarity table for 7 programs and 19 variants

	P1V1	P1V2	P1V3	P2V1	P3V1	P4V1	P4V2	P4V3	P4V4	P4V5	P4V6	P5V1	P6V1	P6V2	P6V3	P6V4	P6V5	P7V1	P7V2
P1V1	8	5	3	0	0	0.1666	0.1333	0.1333	0	0.1538	0	0	0	0	0	0	0	1	0
P1V2	5	8	5	0	0	0.1666	0.1333	0.1333	0	0.1538	0	0	0	0	0	0	0	1	0
P1V3	3	5	9	0	0	0.0909	0.0714	0.0714	0	0.0833	0	0	0	0	0	0	0	0.3333	0
P2V1	0	0	0	41	1.0588	0	0.125	0.125	0.1	0.1	0.2857	0.7407	0	0	0	2	0	0	0
P3V1	0	0	0	1.0588	169	0	0	0	0.0454	0	0.048	1.7142	0	0	0	0	0	0	0
P4V1	0.1666	0.1666	0.0909	0	0	26	7.3333	7.3333	11	18	3	0	0	0	0	0	0	0.0714	0
P4V2	0.1333	0.1333	0.0714	0.125	0	7.3333	29	29	8.5	9.5	17	0	0	0	0	0.25	0	0.0588	0
P4V3	0.1333	0.1333	0.0714	0.125	0	7.3333	29	29	8.5	9.5	17	0	0	0	0	0.25	0	0.0588	0
P4V4	0	0	0	0.1	0.0454	11	8.5	8.5	21	12	6.6666	0	0	0	0	0.2	0	0	0
P4V5	0.1538	0.1538	0.0833	0.1	0	18	9.5	9.5	12	21	5	0	0	0	0	0.2	0	0.0667	0
P4V6	0	0	0	0.2857	0.0487	3	17	17	6.6666	5	30	0	0	0	0	0.571	0	0	0
P5V1	0	0	0	0.7407	1.7142	0	0	0	0	0	0	530	0	0	0	0	0	0	0
P6V1	0	0	0	0	0	0	0	0	0	0	0	0	44	44	41	36	44	0	0
P6V2	0	0	0	0	0	0	0	0	0	0	0	0	44	44	41	36	44	0	0
P6V3	0	0	0	0	0	0	0	0	0	0	0	0	41	41	45	33	41	0	0
P6V4	0	0	0	2	0	0	0.25	0.25	0.2	0.2	0.5714	0	36	36	33	41	36	0	0
P6V5	0	0	0	0	0	0	0	0	0	0	0	0	44	44	41	36	44	0	0
P7V1	1	1	0.3333	0	0	0.0714	0.0588	0.0588	0	0.0666	0	0	0	0	0	0	0	4	2
P7V2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	4

Table 6. Details of projects used for analysis

Sl. No	Project name	Details	KLOC	Number of files processed by our method
1	Cook	Cook is a tool for constructing files from a given set of files.	70	133
2	Snns	Stuttgart Neural Network Simulator is a neural network simulator developed by University of Stuttgart.	34	35
3	SLP	Student lab assignment programs	13	93

Fig 4: Similarity graphs. a) Third instance of P1 program, b) Second instance of P6 program and c) similarity graph for 7 programs and 19 instances

Table 7. Details of injected clones

Sl. No	Projects	Injected T1	Injected T2	Injected T3	Sum of injected clones	Sum of clones detected
1	Cook	1	1	1	3	3
2	Snns	1	0	1	2	2

Clustering

To find whether only versions of the same programs show higher similarity when compared to similarities with other programs, we performed clustering of similarity values using k-means clustering algorithm with k=2. Table 8 shows the clustering results of table 5. The clustering is done on a set of similarity values corresponding to one version of a program (available in a column). Then these clustered results are compared with standard results given by Bellon and team. Then accuracy of the proposed method is calculated using formula (2). For instance, in column 'P4V1', true positive is 5 and true negative is 13. Hence accuracy is computed as $(5+13)/19 = 0.9473$. Same way accuracy is calculated for every column and average accuracy is computed. Table 9 shows accuracy table for data sets used for experimentation.

$$Accuracy = \frac{True\ positive + True\ negative}{Total\ number\ of\ program\ files} \dots (2)$$

where, True positive indicates correctly identified program files and True negative indicates correctly rejected program files.

Table 8. Clustering results for similarity values in table 5

	P1V1	P1V2	P1V3	P2V1	P3V1	P4V1	P4V2	P4V3	P4V4	P4V5	P4V6	P5V1	P6V1	P6V2	P6V3	P6V4	P6V5	P7V1	P7V2
P1V1	2	1	2	1	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1
P1V2	2	1	2	1	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1
P1V3	2	1	2	1	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1
P2V1	1	2	1	2	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1
P3V1	1	2	1	1	2	1	1	1	2	1	1	1	2	1	1	1	1	1	1
P4V1	1	2	1	1	1	2	2	2	1	2	1	1	2	1	1	1	1	1	1
P4V2	1	2	1	1	1	2	2	2	1	2	2	1	2	1	1	1	1	1	1
P4V3	1	2	1	1	1	2	2	2	1	2	2	1	2	1	1	1	1	1	1
P4V4	1	2	1	1	1	2	2	2	1	2	2	1	2	1	1	1	1	1	1
P4V5	1	2	1	1	1	2	2	2	1	2	2	1	2	1	1	1	1	1	1
P4V6	1	2	1	1	1	1	2	2	1	2	2	1	2	1	1	1	1	1	1
P5V1	1	2	1	1	1	1	1	1	2	1	1	2	2	1	1	1	1	1	1
P6V1	1	2	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2	1	1
P6V2	1	2	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2	1	1
P6V3	1	2	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2	1	1
P6V4	1	2	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2	1	1
P6V5	1	2	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2	1	1
P7V1	1	2	1	1	1	1	1	1	2	1	1	1	2	1	1	1	1	2	2
P7V2	1	2	1	1	1	1	1	1	2	1	1	1	2	1	1	1	1	2	2

Table 9. Resultant table

Sl.no	Project name	Accuracy
1	Cook	0.9615
2	Snns	0.991837
3	SLP	0.9846

CONCLUSION AND FUTURE WORK

The proposed method takes a set of program files as input, compares these programs pair wise and computes similarity value. Proposed method efficiently detects clones in polynomial time complexity and without any transformation of source code (without using lexer / scanner or AST, PDG generator). In this work we have just considered all blocks of various types in the programs. In the future work, Block flow graph (similar to control flow graph), which indicates the execution sequence of blocks may be generated for the programs and these can be compared. Perhaps this can increase the efficiency of clone detection.

REFERENCE

[1] L. A. Belady and M. M. Lehman. A model of large program development. IBM Systems Journal, 15(3):225–252, 1976.

- [2] Automated Detection of Code Duplication Clusters By Richard Wettel Diploma Thesis Faculty of Automatics and Computer Science of the "Politehnica" University of Timișoara Timișoara, June 2004.
- [3] Kevin Greenan, "Method – Level Code Clone Detection on Transformed Abstract Syntax Trees Using Sequences Matching Algorithms" March 2015.
- [4] M Sudhamani and L Rangarajan, "Structural Similarity Detection using Control Statements", Proc. International Conference on Information and Communication Technology vol 46, (ICICT) 2015 892-899.
- [5] C.K. Roy, J.R. Cordy, A survey on software clone detection research, Queen's Technical Report: 541, 2007, p. 115.
- [6] M Sudhamani and L Rangarajan, "Code Clone Detection Based on Order and Content of Control Statements", 2nd International Conference on Contemporary Computing and Informatics (ic3i) 2016 (59-64) DOI: 10.1109/IC3I.2016.7917935
- [7] C. K. Roy, J. R. Cordy, R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Comparison and Evaluation of Code Clone Detection Techniques, Science of Computer Programming, 74, 470-495, (2009).
- [8] Ali El-Matarawy, Mohammad El-Ramly, Reem Bahgat, "Code Clone Detection using Sequential Pattern Mining", International Journal of Computer Applications (0975 – 8887) Volume 127 – No.2, October 2015.
- [9] J. Johnson, Substring matching for clone detection and change tracking, in: Proceedings of the 10th International Conference on Software Maintenance, ICSM 1994, pp. 120_126.
- [10] M. Rieger, Effective clone detection without language barriers, Ph.D. Thesis, University of Bern, Switzerland, 2005.
- [11] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, 1999, pp. 109_118.
- [12] S. Lee, I. Jeong, SDD: High performance code clone detection system for large scale source code, in: Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA Companion 2005, 2005, pp. 140_141.
- [13] Brenda S. Baker. Finding Clones with Dup: Analysis of an Experiment. IEEE Transactions on Software Engineering, Vol. 33(9): 608-621, September 2007.
- [14] Brenda S. Baker. A Program for Identifying Duplicated Code. In Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:4957, March 1992.
- [15] Raghavan Komondoor. Automated Duplicated-Code Detection and Procedure Extraction. Ph.D. Thesis, 2003.
- [16] M Sudhamani and L Rangarajan, "Duplicate Code Detection using Control Statements", International Journal of Computer Applications Technology and Research Vol 4, Issue 10, IJCATR 2015 (728-736) .

- [17] A. Marcus, J. Maletic, Identification of high-level concept clones in source code, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, 2001, pp. 107-114.
- [18] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654-670.
- [19] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, on detection of gapped code clones using gap locations, in: Proceedings 9th Asia-Pacific Software Engineering Conference, APSEC 2002, 2002, pp. 327-336.
- [20] V. Wahler, D. Seipel, J. Gudenberg, G. Fischer, Clone detection in source code by frequent itemset techniques, in: Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, SCAM 2004, 2004, pp. 128-135.
- [21] Yu, D.Peng, X.Zhao, W., "An Automatic Refactoring Method of Cloned Code Using Abstract Syntax Tree and Static Analysis", Journal of Chinese Computer Systems, Vol.28, 2008.
- [22] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. In Journal of Universal Computer Science, 8(11):10161038, November 2002.
- [23] Wu Yang. Identifying syntactic differences between two programs. In Software Practice and Experience, 21(7):739755, July 1991.
- [24] Rainer Koschke, J.-F. Girard, M. Wrthner. An Intermediate Representation for Reverse Engineering Analyzes. In Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98), pp. 241-250, Honolulu, Hawaii, USA, October 1998.
- [25] Brenda S. Baker. On Finding Duplication in Strings and Software. Journal of Algorithms, 1993.
- [26] Neil Davey, Paul Barson, Simon Field, Ray J Frank. The Development of a Software Clone Detector. International Journal of Applied Software Technology, Vol. 1(3/4):219-236, 1995
- [27] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 96-105, Minnesota, USA, May 2007.
- [28] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.
- [29] Raghavan Komondoor. Automated Duplicated-Code Detection and Procedure Extraction. Ph.D. Thesis, 2003.
- [30] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), pp.301-309, Stuttgart, Germany, October 2001.
- [31] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In the Proceedings of the 12th ACM

SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06), pp. 872-881, Philadelphia, USA, August 2006

[32] Jean Mayrand, Claude Leblanc, Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proceedings of the 12th, International Conference on Software Maintenance (ICSM'96), pp. 244-253, Monterey, CA, USA, November 1996.

[33] M. Weiser. Program Slicing. In IEEE Transactions on Software Engineering, Vol.10(4):352-357, July 1984.