Code clone detection: An approach based on statement level features

[1]Dr Sudhamani M and [2]Lalitha Rangarajan

[1] Department of Computer Science, MMK & SDM MMV Mysuru, [2]Department of Studies in Computer Science, University of Mysore, Mysore - 570006, Karnataka, INDIA

Abstract

Software systems are gradually evolving with the addition of new features, functions or modification of the existing modules. Code clones are copied or near-copied portions of other code segments. Duplicate codes weaken the software quality and results in complexities, such as, software maintenance, readability, and understand ability. In this paper, we make an attempt to detect code clones, both at syntactic and semantic levels, by proposing an approach that exploits program statement level features. For example, type of statement, operators and operands, and their counts. For corroboration of the efficacy of the proposed approach, we conducted experiments on standard C projects of Bellon's benchmarking and student lab program (SLP) datasets. Moreover, we compared the performance of the proposed approach with the NICAD (Accurate Detection of Near-Miss Intentional Clones), CLAN and clone manager tools, also. Extensive experiments demonstrate the promising findings, which can be used in future investigations.

Keywords: Software systems, Code clones, Program statements, Duplicate code, Program fragments, Software quality, Program syntactic and semantics.

1. Introduction

Due to the constant modification and frequent copy-paste/duplication of code fragments viz. code clones in software systems, manual inspection cannot be done. Over the years, an enormous amount of research has been carried out for automating the identification/detection of code clones [1–4]. However, a full-fledged automatic identification system is still a distant dream due to the challenges being posed by constant development of software systems. As a matter of fact, detection of code clones are indispensable for many software engineering tasks, such as, program understanding, refactoring, optimization, code searching, and bug detection [1, 5–8], etc. Hence, identification of code clones carries an utmost importance in software engineering and has been widely studied.

Two major issues concerning the detection of code clones is the large size and complexity of software systems. These issues have made identification of code clones as an indispensable task for software

maintenance [9], and has led to the development of many automated tools[10, 11]. Furthermore, it has been found that code fragments are not copied as it is, so their detection process must ignore this discrepancy/inconsistency. And, concentrate on similarities for duplicate detection. As stated, to eliminate defects/bugs and extend their functionalities software systems are undergoing continuous modification, and maintenance. While performing these activities intentionally or unintentionally code fragments get copied/duplicated. Moreover, various studies revealed that about 5% to 20% of a software may contain code clones [12]. Therefore, we need to deal with this situation so that to prevent problems that might appear when software tries to adapt the changes that are imminent in a real-world systems.

In literature, depending upon the level of similarity, code clones are divided into four broad categories [12]. **Type-I:** identical code fragments except for variations in whitespace, layouts, and comments. **Type-II:** structurally and syntactically identical fragments except for variations in identifiers, literals and function names. **Type-III:** copied fragments with modifications, e.g. statements added or removed. **Type-IV:** code fragments which perform the identical functions, but implemented using different syntactic variants, also referred as *semantic clones*. There exist various approaches for Type-I, Type-II, and Type-III code clones [13, 14]. However, presence of syntactic and semantic flexibility in the source code make Type-IV a complex process. In other words, both syntax, and structure/relationships between code fragments need to be taken into account. For this reason, there are various unresolved issues in this type.

From literature, we found that many approaches try to use syntax and semantics based information separately to represent source code. Like, for instance, abstract syntax tree (AST) [15, 16], program dependence graph (PDG) [17, 18], text [19, 20], and token based [21, 22], etc. However, these approaches on one hand, represent one aspect but on the other, lack the capability of representing other important aspect. For example, AST represents syntax tree but incapable to represent control flow statements. Similarly, approaches based on PDG are computationally extensive (NP-hard). With this motivation, in this paper, we present a metric based approach to identify Type-IV code clones i.e., detection of similar functional statements. Broadly speaking, our approach falls in machine learning paradigm which have achieved substantial performance in number of studies. The basic idea of our approach is to extract statement level features such as type of operators, operands and their counts, from functional statements present in a program. Thereafter, a dissimilarity matrix is constructed and finally, after matching, code clones are detected. In brief the main contributions of this study are: (*i*) Detects functionally similar code clones, (*ii*) identifies structurally identical code fragments, and (*iii*) detects syntactic and semantically identical code fragments also. The remainder of this paper is organized as follows. Detailed related work about code clone detection is described in section 2. Section 3 explains the proposed approach. Experimental results and their performance, comparative analysis is carried in section.4. Finally, conclusion and future work is drawn in section 5.

2. Related works

A number of researchers have attempted to detect the code clones. As reuse of existing code is key factor in software development. Moreover, if it is not detected, it downgrades the design, structure, quality, readability, changeability, and maintainability of software systems [23] . The earlier attempts mainly differ in representation and approach used.

2.1. String based

String based approaches perform comparison based on source code characters. Comparison of two strings is usually done with calculating some form of edit distance [5, 24]. Baker's Dup tool fixates duplication or near-duplication in a large software system. And, transforms program text into sequence of tokens. Later, line based string matching algorithm is used to detect duplicates[5]. Dup identifies parameterized matches and it emphasizes results in the form of both report and scatter plots. This tool does not support exploration and navigation in the duplicated code. The tool fails to detect code written in different styles. Johnson used incremental hash function combined with sliding window method to find clones of different lengths and Karp-Rabin fingerprint algorithm is applied to detect duplicate code [9, 24].

Latent Semantic Indexing (LSI) is an information retrieval technique used to find semantic similarities in source code [25]. Dynamic pattern matching (DPM) is applied on normalized source lines to compare strings [6]. Nearest-neighbor (NN) algorithm is used to detect near miss clones in similar data detection (SDD) [26]. Cordy et al. [27], used an island grammar to find near miss clones in HTML web pages. String based approach are unable to detect variable rename and reorder statements. It does not perform any syntactic or semantic analysis on source code. This method can detect Type-I clones.

2.2. Token based

Source lines are transformed into sequence of token streams using respective scanner or lexer [6, 28]. Later, these sequence of tokens are scanned to detect code clones. Afterwards, these token sequences are compared by following either suffix tree algorithm (STA) or hashing algorithm (HA) [29]. This technique is slightly slower than text based method, because of the tokenization process. And, is capable to detect both Type-I and Type-II clones. Tools based on token based technique are CCFinder, CPminer, JPlag, and CPD11 [9, 30].

Brenda et al., [5] developed an effective token-based clone detection tool referred as Dup. In Dup, lexer is used to tokenize the source code. Thereafter, these token sequences are compared using STA. CCFinder, CP-Miner, Gemini, and RTF are prominent token based tools. Kamiya et al.[31] used source normalization in CCFinder. Gemini et al., [32] visualizes near miss clones using scatter plots and RTF tokenization after using STA. Furthermore, frequent subsequence data mining technique is adopted in CP-Miner to find similar stream of tokens [33]. Winnowing, JPlag and SIM [30, 34] are familiar token based plagiarism detection tools. However, CCFinder and Dup are unreliable to handle reordered statements but CP-Miner handles this situation efficiently. Compared to text-based approaches token-based approaches are usually resistant against code changes such as

formatting, spacing, and variable renaming. Token based approaches are capable to detect Type-1 and Type-2 clones but finds it difficult to detect near miss clones.

2.3. Tree based

Source code is transformed into AST with appropriate parser. Then, tree matching technique is used to search similar sub-ASTs [30]. The parse tree/AST contains the complete information about the source code. When any match is found corresponding source code of the similar sub trees are reported as clone pairs. The results drawn by tree comparison are reasonably efficient but difficult and complex to generate, thanks to transformation into AST and their poor scalability. AST-based approaches can identify exact, near miss, and gap clones. However, disregards the information about variable rename. AST does not show the data flow information, therefore, it cannot handle statement reorder and control replacement.

CloneDR is the prime AST based tool developed by Baxter et al [35]. 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 [36], the ASTs are represented in IML (Intermediate Language) and comparison is done on IMLs rather than ASTs. Yang [37] 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.[38] found exact and parameterized clones by transforming AST of a program to an XML representation and then a data mining frequent item set technique is applied to the XML representation to detect clones. Finally, we also come across Evans et al.[39], method that identifies exact and near miss clones with gaps.

2.4. Program dependence graph (PDG) based

Source code is transformed into graph called Program dependency graph (PDG). Which contains the control and data flow information of a program and hence carries semantic information [18]. Then isomorphic sub-graph matching is applied to find similar sub graphs that are reported as clones. PDG based approaches are robust in detecting functional similarities. Like, for example in [30], source code is represented with high abstraction. PDG-based approaches, on one hand, can effectively deal with reordered statements, insertion and deletion of code, intertwined code, and non-contiguous code, but, on the other hand, not scalable to large size programs (due to finding isomorphic sub-graphs that is with NP-hard complexity [30, 40, 41]).

One of the leading PDG-based clone detection approaches presented is Komondoor et al [42]. And, finds isomorphic PDG sub-graphs using program slicing. Krinke et al. [43] used k-length patch matching iterative approach for detecting maximal similar sub-graphs [43]. GPLAG [44] is a PDG-based plagiarism detection tool. Chen at al.[45], proposed a PDG-based technique for code compaction taking into account syntactic structure and data flow.

2.5. Metric based

Metric-based approaches gather different metrics for code fragments and compares metric values to check similarity between two code fragments [46]. 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 are in capable to detect segment-based copy-paste clones. However, is scalable and straight forward.

Mayrand et al.[47], calculated metrics, like the number of lines of source, number of function calls contained, and 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. Patenaude et al. [48] 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 [49].

Kontogiannis et al. [50], design an abstract pattern matching tool to identify probable matches using markov models (MM) to compute similarity between programs. They proposed two ways to detect clones, first, is direct comparison of metrics computed from AST and computes the ratio of input / output variable to the fan-out (number of function calls), McCabe cyclomatic complexity, modified Albercht's function point metric and modified Henry-Kafaura's information flow quality metric). The cecond 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 [47]. Metric-based approach is also used to find duplicated web pages. Di Ducca et al.[46] 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.[51] proposed a semi-automated method to detect clone script functions, using eMetrics tool to retrieve the potential function clones. Davey et al. [52], detects exact, parameterized and near-miss clones by using neural networks on features retrieved.

2.6. Hybrid-based approach

Hybrid techniques are combination of other techniques. There are various alternative code clone techniques that use a hybrid approach. These techniques integrate syntactic and semantic characteristics. New languages can be added by the specification of their syntaxes. It is also possible to explore other language features by including new specialized comparison functions.

Leitao et al [25], provides a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques using call graph method in combination with specialized comparison functions [52]. In the approach by Koschke et al.[47], the tokens of the AST-nodes are compared using a STA 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 STA. Greenan et al.,[48], proposed an algorithm to find method level clones on transformed AST using sequence matching algorithm. Jiang et al.[30] proposed a new method to find similar trees by comparing the structural information of AST in Euclidean space and Locality sensitive HA was used to cluster the clones.

Overall, from the literature we observe that each technique comes with its pros and cons. Further from the different artificial intelligence and machine learning fields, we observe that efficient feature representation is an important aspect for object characterization/detection. Therefore, in this paper, we extracted program statement level features from source code. We observe from the literature there are few attempts in this direction. Which is striking as the program flow is governed by entirely these statements.

3. Proposed Method

The proposed approach for code clone detection extracts the features from functional statements in the source code. Figure 1 illustrates the different steps/stages of the proposed methodology.

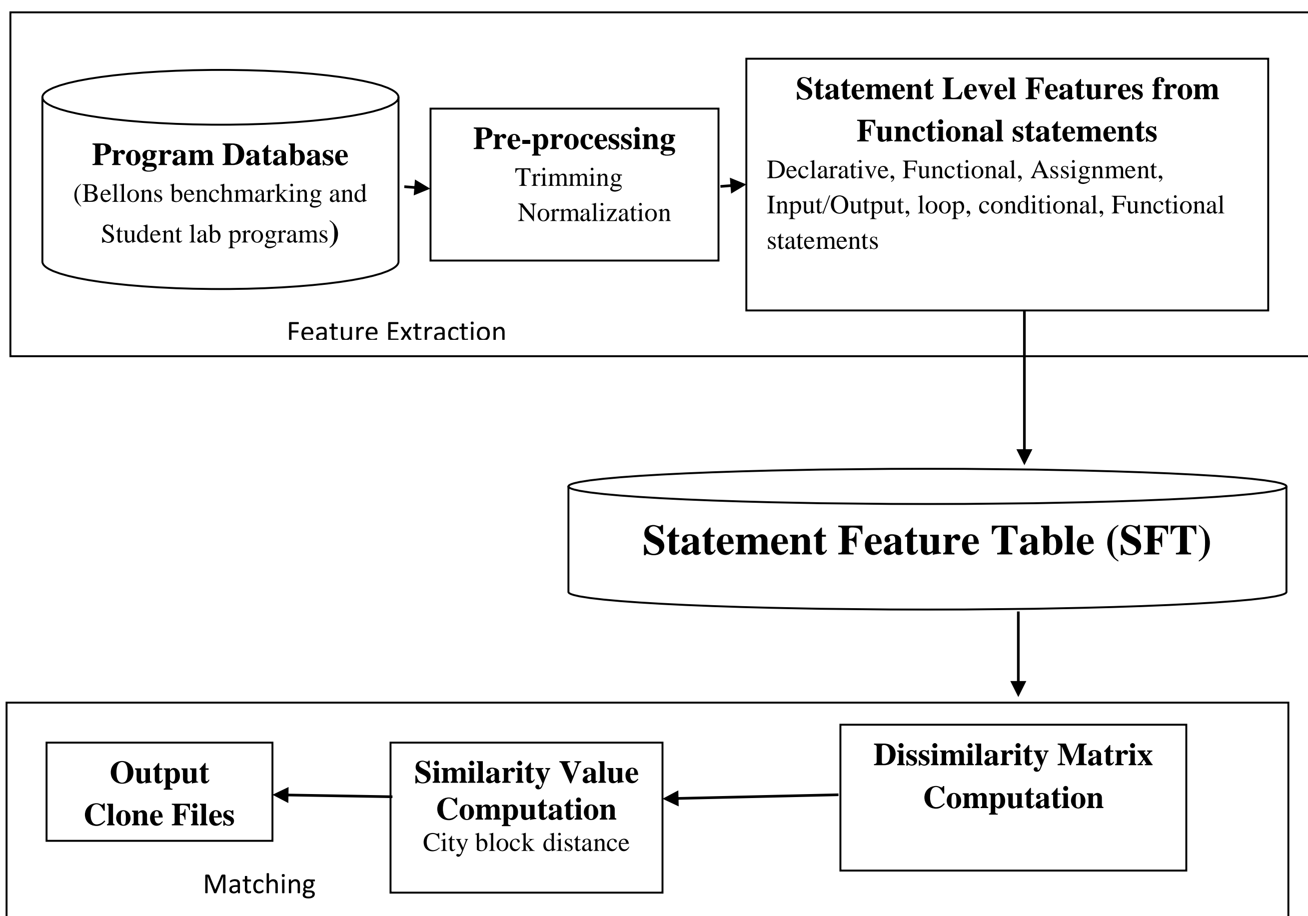


Figure 1. Illustration of proposed approach

3.1. Preprocessing

At the outset, we followed state-of-the-art pre-processing steps [53, 54], to the input programs in the database. That ranges from trimming (removing extra spaces, comments, and un-executable statements) to normalization.

3.2. Feature computation and Dissimilarity matrix construction

We extracted statement level features from the following: (*i*) Declarative statements (DS), (*ii*) Function definition statements (FDS), (*iii*) Assignment statements (AS), (*iv*) Input statements (IS), (*v*) Output statements (OS), (*vi*) Loop statements (LS) (*vii*) Conditional statements (CS),

and (*viii*) Function call statements (FC). For the sake of understanding, consider two versions of a sorting program (Figure 2) after extraction, features are stored in separate tables referred as statement feature tables (SFTs) and are shown in Table 1 and Table 2, respectively. Thereafter, dissimilarity matrix (DM) is constructed from both the SFTs. Corresponding to each statement, a row of program-I is compared with each row of program-II. Note, we have used city block distance [55], for comparison. For example, distance between 12th statement ('for' loop from Table 1) and 13th statement ('while' loop from Table 2) of program-I and program-II (see Figure 2) is computed from corresponding rows in SFTs as: |0-0| + |1-1| + |0-0| + |1-0|+ |0-0| + |0-0| + |0-0| + |0-0| + |0-0| + |1-1| + |0-0| +|0-0| |0-0| + |0-0| + |0-0| + |1-1|+|0-0| + |0-0| + |0-0| + |4-3|+|0-0| + |1-0| = 3 and is stored in DM (12th row and 13th column), as shown in Figure 3. Note that, presence of zero indicates that program statements are similar. Table 3 shows the numbers of probable similar statements between two programs.

3.3. Similarity value computation

Similarity between two programs is computed using Equation 1.

$$S = \begin{cases} n & \textit{if } (\mathrm{r}_1 == \mathrm{r}_2) \\ \dfrac{n}{|r_1 - r_2|} & \textit{Otherwise} \end{cases} \qquad (1)$$

Here 'n' is the number of zero's (i.e. number of similar statements), r_1 and r_2 are number of executable statements of corresponding programs/versions, upon computation, which equals to 39 / (|19-21|) = 19.5 (see Figure 3). Table 3 is used to find similar code segments, for instance, consecutive statements of program-(a) (13 to 16) and program-(b) (14 to 17) are in successive rows(27 to 30) in Table 3. Further, these statements are inside conditional blocks in both the programs, therefore, are similar blocks. As a result, statements 1 to 11 are similar blocks/segments of two programs and these entries appear at different rows of the Table 3. It may be noted that, any rear-

rangement of the Table 3 shall indicate similar blocks. However, Table 3 is also used to find overall similarity between two versions of a program. Each step shown in Figure 1 is repeated for each time whenever we have to establish similarity between two programs or we have to detect code clones between any two programs.

```
1.#include < stdio.h >
2.Void main () {
3.int n,i, j,a[25],temp;
4.printf ("\n Enter the range ");
5.scanf ("%d",&n);
6.printf ("\n Enter the numbers ");
7.for (j = 1; j <= n; j++)
8.scanf ("%d", &a[i]);
9.printf ("\n Numbers before sort");
10.for (j = 1; j <= n; j++)
11.Printf ("%d", a[i]);
12.for (i = 1;i <= n;i++)
13.{
14.for (j = 1; j <= n; j++)
15.{
16.if (a[j] > a[j]+1)
17.{
18.temp = a[j];
19.a[j] = a[j+1];
20.a[j+1] = temp;
21.}
22.}
23.}
24. printf ("The sorted array is");
25.for (i = 0;i < n;i++)
26.printf ("%d",a[i]);
27.}
```

(*i*)

```
1.#include < stdio.h >
2.Void main () {
3.int n,i, j,a[25],temp;
4.printf ("\n Enter the range ");
5.scanf ("%d",&n);
6.printf ("\n Enter the numbers ");
7.for (j = 1; j <= n; j++)
8.scanf ("%d", &a[i]);
9.printf ("\n Numbers before sort");
10.for (i = 1;i <= n;i++)
11.Printf ("%d", a[i]);
12.for (j = 1; j <= n; j++)
13.{
14.j = 1;
15while (j<= n)
16.{
17.if (a[j] > a[j]+1)
18.{
19.temp = a[j];
20.a[j] = a[j+1];
21.a[j+1] = temp;
22.}
23.}
24.j++;
25.}
26. printf ("The sorted array is");
27.for (i = 0;i < n;i++)
28.printf ("%d",a[i]);
29.}
```

(*ii*)

Figure 2. Two versions of sorting. (*i*) Program-I (*ii*) Program-II

Table 1. Statement feature table (SFT) for program-I in Figure. 2

Line No	Statement type	Control lines			Assignment and Arithmetic operators								Relational operators						Logical operators		Operands	
		L	C	OC	=	+	-	*	/	mod	++	--	==	!=	<	>	<=	>=	&&	||	variables	constants
1	(FDS)	5	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
2	(DS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	1
3	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	(IS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
5	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
7	(IS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
8	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
10	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
11	(LS)	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
12	(LS)	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
13	(CS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	4	1
14	(AS)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0
15	(AS)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	1
16	(AS)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1
17	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
19	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0

Table 2. Statement feature table (SFT) for program-II in Figure. 2

Line No	Statement type	Control lines			Assignment and Arithmetic operators								Relational operators						Logical operators		Operands	
		L	C	OC	=	+	-	*	/	mod	++	--	==	!=	<	>	<=	>=	&&	||	variables	Constants
1	(FDS)	5	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
2	(DS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	1
3	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	(IS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
5	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
7	(IS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
8	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
10	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
11	(LS)	1	1	0	2	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	5	1
12	(AS)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
13	(LS)	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	3	1
14	(CS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	4	1
15	(AS)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0
16	(AS)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	1
17	(AS)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1
18	(AS)	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
19	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	(LS)	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	4	1
21	(OS)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1		(FDS)	(DS)	(OS)	(IS)	(OS)	(LS)	(IS)	(OS)	(LS)	(OS)	(LS)	(AS)	(LS)	(CS)	(AS)	(AS)	(AS)	(AS)	(OS)	(LS)	(OS)
2	(FDS)	0	14	8	10	8	14	10	8	14	10	12	11	10	14	12	15	14	8	8	14	10
3	(DS)	14	0	6	4	6	4	4	6	4	4	6	5	6	2	4	3	4	6	6	4	4
4	(OS)	8	6	0	2	0	8	2	0	8	2	10	3	6	6	4	7	6	2	0	8	2
5	(IS)	10	4	2	0	2	6	0	2	6	0	8	3	4	4	2	5	4	2	2	6	0
6	(OS)	8	6	0	2	0	8	2	0	8	2	10	3	6	6	4	7	6	2	0	8	2
7	(LS)	14	4	8	6	8	0	6	8	0	6	2	5	4	4	4	3	4	6	8	0	6
8	(IS)	10	4	2	0	2	6	0	2	6	0	8	3	4	4	2	5	4	2	2	6	0
9	(OS)	8	6	0	2	0	8	2	0	8	2	10	3	6	6	4	7	6	2	0	8	2
10	(LS)	14	4	8	6	8	0	6	8	0	6	2	5	4	4	4	3	4	6	8	0	6
11	(OS)	10	4	2	0	2	6	0	2	6	0	8	3	4	4	2	5	4	2	2	6	0
12	(LS)	12	6	10	8	10	2	8	10	2	8	0	7	4	6	6	5	6	8	10	2	8
13	(LS)	13	5	9	7	9	1	7	9	1	7	1	6	3	5	5	4	5	7	9	1	7
14	(CS)	14	2	6	4	6	4	4	6	4	4	6	5	6	0	4	3	4	6	6	4	4
15	(AS)	12	4	4	2	4	4	2	4	4	2	6	3	4	4	0	3	2	4	4	4	2
16	(AS)	15	3	7	5	7	3	5	7	3	5	5	4	7	3	3	0	1	7	7	3	5
17	(AS)	14	4	6	4	6	4	4	6	4	4	6	3	6	4	2	1	0	6	6	4	4
18	(OS)	8	6	0	2	0	8	2	0	8	2	10	3	6	6	4	7	6	2	0	8	2
19	(LS)	14	4	8	6	8	0	6	8	0	6	2	5	4	4	4	3	4	6	8	0	6
20	(OS)	10	4	2	0	2	6	0	2	6	0	8	3	4	4	2	5	4	2	2	6	0

Figure 3. Distance matrix computed using Table 1 and Table 2

Table 3. Index table (probable similar statements between program-I and program-II)

Sl. No	Statement no in Table	Statement no in Table
1	1	1
2	2	2
3	3	3
4	5	3
5	8	3
6	17	3
7	4	4
8	7	4
9	3	5
10	5	5
11	8	5
12	17	5
13	6	6
14	9	6
15	18	6
16	4	7
17	7	7
18	3	8
19	5	8
20	8	8
21	17	8
22	6	9
23	9	9
24	18	9
25	10	10
26	19	10
27	13	14
28	14	15

29	15	16
30	16	17
31	3	19
32	5	19
33	17	19
34	6	20
35	9	20
36	18	20
37	10	21
38	19	21
39	11	11

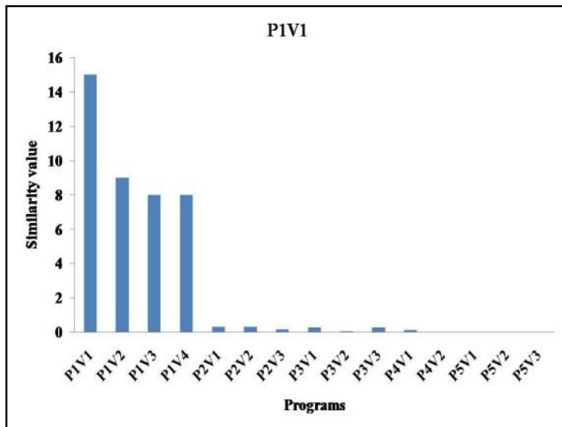
4. Experimentation

4.1. Experimental results

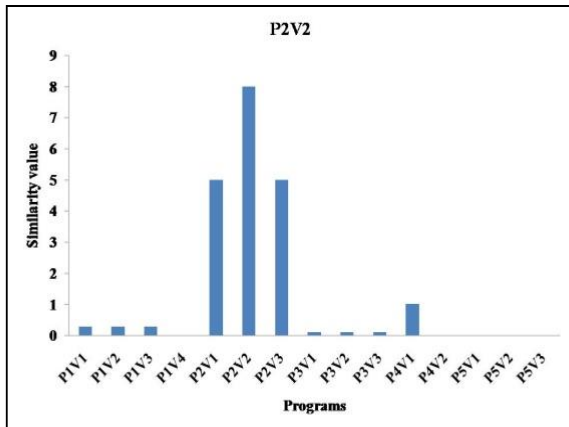
We conducted experiments on fifteen (15) versions of five programs of the Bellon's benchmarking [56], dataset and student lab programs dataset created in our lab comprising of 93 programs designed by students. We can observe from Table 5 each program shows highest similarity with its version as compared to other programs. The Figure 4 shows the similarity values, for example, Figure 4a shows the similarity value of version 1 of program-1 (P1V1) with other program versions. Figure 4f shows the total similarity values.

Table 5. Obtained similarity values

	P1V1	P1V2	P1V3	P1V4	P2V1	P2V2	P2V3	P3V1	P3V2	P3V3	P4V1	P4V2	P5V1	P5V2	P5V3
P1V1	554	318.5	44.583	42.5	3.2647	1.631	2.4117	0.525	0.6315	0.6315	14.467	14.75	22.153	22.154	22.154
P1V2	318.5	890	45.646	43.571	3.9375	1.611	2.7187	0	0.1388	0.1388	18.1562	18.3620	19.066	19.066	19.066
P1V3	44.583	45.643	650	625	3.1521	1.8	2.3043	0.9230	1.06	1.06	20.12	21.1590	390	390	390
P1V4	42.5	43.571	625	618	3.0652	1.82	2.2608	0.9423	1.08	1.08	19.5	20.523	386	386	386
P2V1	3.2647	3.937	3.1524	3.0652	42	7	25	2.6666	4.5	4.5	2.2812	2.2555	2.1702	2.1702	2.1702
P2V2	1.6315	1.6111	1.8	1.82	7	26	5.25	8	16	16	1.2	1.2128	1.45098	1.4509	1.4509
P2V3	2.412	2.7187	2.3043	2.2609	25	5.25	34	1.8333	2.75	2.75	1.5416	1.5444	1.4042	1.4042	1.4042
P3V1	0.525	0	0.9231	0.9423	2.6666	8	1.8333	26	12	12	0.4803	0.5104	1.3585	1.3584	1.3585
P3V2	0.6315	0.1388	1.06	1.08	4.5	16	2.75	12	28	28	0.55	0.5638	1.4313	1.4313	1.4314
P3V3	0.6315	0.1388	1.06	1.08	4.5	16	2.75	12	28	28	0.55	0.5638	1.4313	1.4313	1.4314
P4V1	14.467	18.156	20.12	19.5	2.2812	1.2	1.54166	0.4803	0.55	0.55	1830	281	11.877	11.877	11.877
P4V2	14.75	18.362	21.159	20.522	2.2555	1.2126	1.5444	0.5104	0.563	0.563	281	1566	12.67	12.67	12.674
P5V1	22.153	19.066	390	386	2.1702	1.4509	1.4042	1.3584	1.4313	1.4313	11.877	12.674	431	431	431
P5V2	22.153	19.066	390	386	2.1702	1.4509	1.4042	1.3584	1.4313	1.4313	11.877	12.674	431	431	431
P5V3	22.153	19.066	390	386	2.1702	1.4509	1.4042	1.3584	1.4313	1.4313	11.877	12.674	431	431	431



(a)



(b)

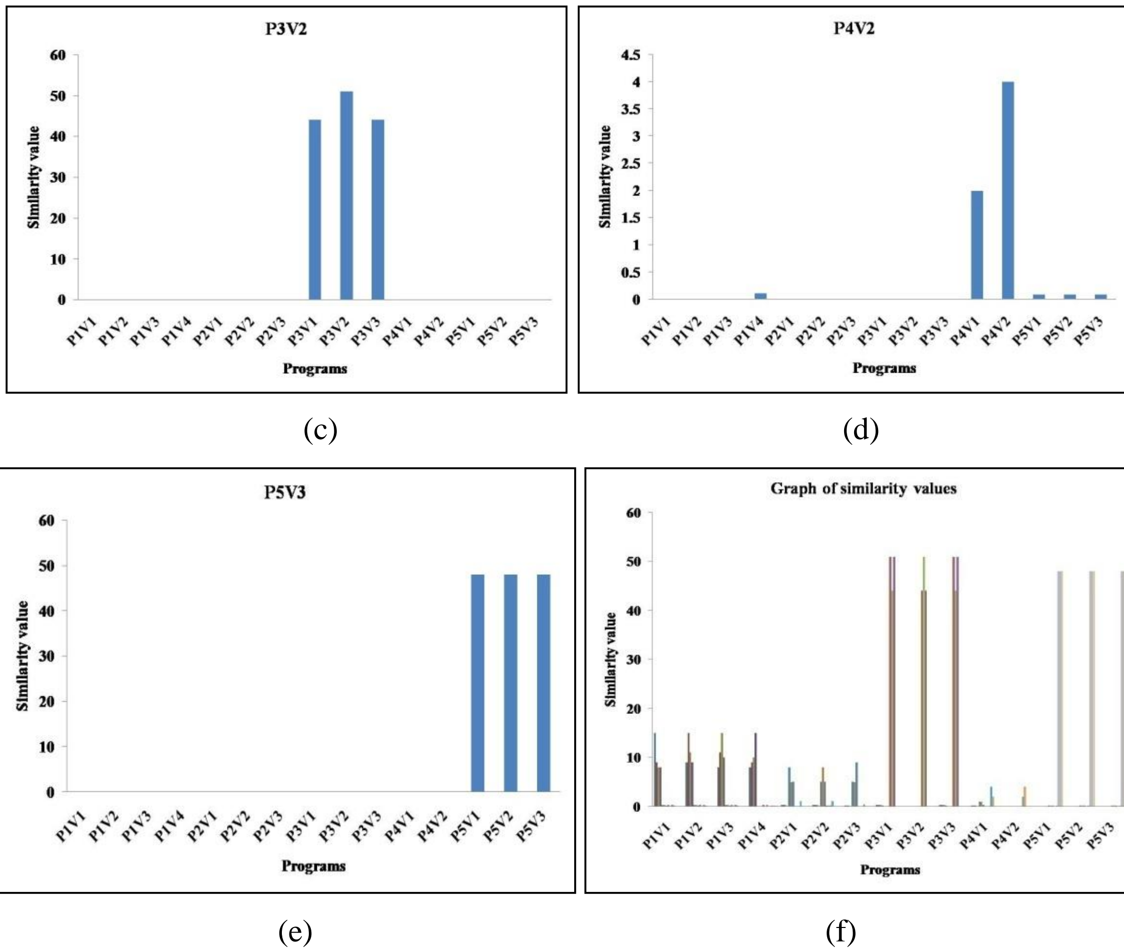


Figure 4. Illustration of similarity values of a program with its versions.

In order to find whether only two versions of a particular program show higher similarity when compared to similarities between other programs. K- Means clustering [57], (with $K = 2$) is performed on similarity values obtained in Table 5 and the results are shown in Table 6. The rationale for using $K = 2$ comes from the fact that the either the program is similar or not. Note, clustering is performed on a set of similarity values corresponding to one version of a program (see column of Table 6).

Table 6. K- Means (with $K = 2$) clustering on the similarity values obtained in Table 5

	P1V1	P1V2	P1V3	P1V4	P2V1	P2V2	P2V3	P3V1	P3V2	P3V3	P4V1	P4V2	P5V1	P5V2	P5V3
P1V1	2	1	1	2	1	1	1	2	2	1	1	2	1	1	1
P1V2	2	2	1	2	1	1	1	2	2	1	1	2	1	1	1
P1V3	1	1	2	1	1	1	1	2	2	1	1	2	2	2	2
P1V4	1	1	2	1	1	1	1	2	2	1	1	2	2	2	2
P2V1	1	1	1	2	2	1	2	2	2	1	1	2	1	1	1
P2V2	1	1	1	2	1	2	1	2	1	2	1	2	1	1	1
P2V3	1	1	1	2	2	1	2	2	2	1	1	2	1	1	1
P3V1	1	1	1	2	1	1	1	1	2	1	1	2	1	1	1

P3V 2	1	1	1	2	1	2	1	2	1	2	1	2	1	1	1
P3V 3	1	1	1	2	1	2	1	2	1	2	1	2	1	1	1
P4V 1	1	1	1	2	1	1	1	2	2	1	2	2	1	1	1
P4V 2	1	1	1	2	1	1	1	2	2	1	1	1	1	1	1
P5V 1	1	1	2	1	1	1	1	2	2	1	1	2	2	2	2
P5V 2	1	1	2	1	1	1	1	2	2	1	1	2	2	2	2
P5V 3	1	1	2	1	1	1	1	2	2	1	1	2	2	2	2

We observe that there are some errors in clone detection. For instance, version 3 and version 4 of program 1 are not detected as clones (i.e., false negative). Overall, percentage of accuracy is 86.22 %.

4.2. Performance Analysis

For the evaluation of the performance of the proposed method. We conducted experiments on the source code of four (4) Bellons benchmarking dataset, and 93 lab student lab programs (SLP), From Bellons dataset we used cook, postgresql, snns, and wetlab projects. Complete results of Bellons experimentation can be found available at https://bauhaus-stuttgart.de/clones/ . However, for coherence, we provided details of projects used in this paper in Table 7. Table 8 gives the results obtained on used projects.

Table 7. Open source projects used for experimentation

S.No.	Project name	Details
1	Cook	Cook is a program tool for constructing files. It is given a set files to create, and instructions in detail how to create them.
2	PostgreSQL	It is a database that runs on many different operating systems.
3	Weltab	It is a vote tabulation system.
4	SNNS	Stuttgart Neural Network Simulator is a neural network simulator originally developed at the university of Stuttgart.
5	SLP	Student Lab Programs (collection of different versions of programs)

Table 8. Performance analysis

S.N	Projects	Precision	Recall	F-measure	Accuracy
1	Cook	0.9153	0,85421	0.8837	0.96137
2	Postgresql	0.8965	0.9134	0.90487	0.9746
3	Snns	0.9261	0.9408	0.9334	0.989
4	Weltab	0.9386	0.80183	0.82595	0.9665
5	SLP	0.7164	0.81993	0.73811	0.96703

4.3. Comparative Analysis

Proposed approach is compared with the results obtained with existing tools namely CLAN, NICAD and Clone Manager. Accuracy is calculated for each column and later average accuracy is reported shown in Table 9.

Table 9. Comparative analysis with the existing tools

Projects	CLAN	NICAD	Clone Manager	proposed
Cook	0.825	0.675	0.9525	0.96137
Postsql	0.62	0.585	0.97	0.9746
Snns	0.5753	0.932	0.9392	0.989
Weltab	0.469	1	0.980	0.9664

4.4. Time complexity

Let SL_1 and SL_2 be the lines of code in any two programs. Table 10 shows the major steps for the computation and their corresponding complexities.

Table 10. Time complexity

Steps	Complexities
Preprocessing	$q(SL_1) + q(SL_2)$
SF	$q(SL_1) + q(SL_2)$
DSFT	$q\ (SL_1 \text{ x } SL_2)$
Similarity computation	$q\ (SL_1 \text{x } SL_2)$

Hence total number of steps is:

$$((q\ (SL1)\ +\ q\ (SL2))\ +\ (q\ (SL1)\ +\ q\ (SL2))\ +\ (q\ (SL1) \text{ x } q\ (SL2))\ +\ (q\ (SL1) \text{ x } q\ (SL2))$$

This is $O\ (SL1 + SL2)$. Therefore, proposed approach is polynomial time complexity.

5. Conclusion and Future work

In general, the syntactic representation and semantics of a program is extremely difficult to establish and, in particular, for code clone detection. There exist a number of studies for detection of code clones. In this paper, we attempted to capture both the representation and semantics by extracting statement level features from program statements. As we know the flow of any program depends on these statements, such as, control, conditional, iterative, and computational statements. Extensive experimentation have been performed on standard benchmarking dataset (C projects of Bellon's dataset) and Student Lab Programs (SLP). Afterwards, comparison is performed with the results obtained using NICAD, CLAN, and Clone Manager tools. We observe that, performance of proposed approach is far better than CLAN, marginally better than Clone Manager and NICAD except for weltab project. In contrast to previous studies, proposed approach extracts features in one scan after pre-processing. More importantly, time complexity of the proposed approach is polynomial that too without any transformation of code. That is, without any usage of lexer/scanner or AST/PDG generators. Although, from the experimentation, proposed approach has demonstrated high performance, however, current framework is limited to C- programs only. There seems to be room for exploring the proposed approach for other programing languages as well i.e., to make it generalize.

References

1. Mens T, Demeyer S (2008) Identifying and Removing Software Clones. Softw Evol 1–347. https://doi.org/10.1007/978-3-540-76440-3
2. Sheneamer A, Roy S, Kalita J (2018) A detection framework for semantic code clones and obfuscated code. Expert Syst Appl 97:405–420. https://doi.org/10.1016/j.eswa.2017.12.040
3. Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Blasco J (2014) Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Syst Appl 41:1104–1117. https://doi.org/10.1016/j.eswa.2013.07.106
4. Sudhamani M, Rangarajan L (2019) Code similarity detection through control statement and program features. Expert Syst Appl 132:63–75. https://doi.org/10.1016/j.eswa.2019.04.045
5. Baker BS (1992) A Program for Identifying Duplicated Code. Comput Sci Stat 24:49–57
6. Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. Proc - Work Conf Reverse Eng WCRE 253–262. https://doi.org/10.1109/WCRE.2006.18
7. Ducasse S, Rieger M, Demeyer S (1999) A Language independent approach for detecting duplicated code. Conf Softw Maint 109–118. https://doi.org/10.1109/icsm.1999.792593
8. Komondoor R, Horwitz S (2001) Using slicing to identify duplication in source code. Lect Notes Comput Sci (including Subser Lect Notes Artif Intell Lect Notes Bioinformatics) 2126 LNCS:40–56. https://doi.org/10.1007/3-540-47764-0_3
9. Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques

and tools: A qualitative approach. Sci Comput Program 74:470–495. https://doi.org/10.1016/j.scico.2009.02.007

10. Dang S, Wani SA (2015) Performance Evaluation of Clone Detection Tools. Int J Sci Res 4:2013–2016
11. Gautam P, Saini H (2016) Various code clone detection techniques and tools: A comprehensive survey. Commun Comput Inf Sci 628 CCIS:655–667. https://doi.org/10.1007/978-981-10-3433-6_79
12. Roy CK, Cordy JR (2007) A Survey on Software Clone Detection Research. 541:115
13. Roy CK, Cordy JR (2018) Benchmarks for software clone detection: A ten-year retrospective. 25th IEEE Int Conf Softw Anal Evol Reengineering, SANER 2018 - Proc 2018–March:26–37. https://doi.org/10.1109/SANER.2018.8330194
14. Rattan D, Bhatia R, Singh M (2013) Software clone detection: A systematic review. Elsevier B.V.
15. Perez D, Chiba S (2019) Cross-language clone detection by learning over abstract syntax trees
16. Jiang L, Misherghi G, Su Z, Glondu S (2007) DECKARD: Scalable and accurate tree-based detection of code clones. Proc - Int Conf Softw Eng 96–105. https://doi.org/10.1109/ICSE.2007.30
17. Liu C, Chen C, Han J, Yu PS (2006) GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. 872. https://doi.org/10.1145/1150402.1150522
18. Krinke J (2001) Identifying similar code with program dependence graphs. Reverse Eng - Work Conf Proc 301–309. https://doi.org/10.1109/wcre.2001.957835
19. Jia Y, King CS (2007) Clone Detection Using Dependence Analysis and Lexical Analysis
20. Roopam, Singh G (2017) To enhance the code clone detection algorithm by using hybrid approach for detection of code clones. Proc 2017 Int Conf Intell Comput Control Syst ICICCS 2017 2018–Janua:192–198. https://doi.org/10.1109/ICCONS.2017.8250708
21. Kamiya T, Kusumoto S, Inoue K (2001) A Token based Code Clone Detection Technique and Its Evaluation
22. Elkhail AA, Svacina J, Cerny T (2019) Intelligent token-based code clone detection system for large scale source code. Proc 2019 Res Adapt Converg Syst RACS 2019 256–260. https://doi.org/10.1145/3338840.3355654
23. Yoshida N, Higo Y, Kusumoto S, Inoue K (2012) An experience report on analyzing industrial software systems using code clone detection techniques. Proc - Asia-Pacific Softw Eng Conf APSEC 1:310–313. https://doi.org/10.1109/APSEC.2012.98
24. Johnson JH (1994) Substring matching for clone detection and change tracking. Proc - 1994 Int Conf Softw Maintenance, ICSM 1994 120–126. https://doi.org/10.1109/ICSM.1994.336783
25. Marcus A, Maletic JI (2005) Identification of high-level concept clones in source code. 107–114. https://doi.org/10.1109/ase.2001.989796
26. Lee S, Jeong I (2005) SDD: High Performance Code Clone Detection System for Large Scale Source Code. Companion to 20th Annu ACM SIGPLAN Conf Object-oriented Program Syst Lang Appl

140–141

27. Cordy JR, Dean TR (2004) Practical Language-Independent Detection of Near-Miss Clones
28. Sudhamani M, Rangarajan L (2016) Code clone detection based on order and content of control statements. Proc 2016 2nd Int Conf Contemp Comput Informatics, IC3I 2016 59–64. https://doi.org/10.1109/IC3I.2016.7917935
29. Tairas R, Gray J (2006) Phoenix-based clone detection using suffix trees. Proc Annu Southeast Conf 2006:679–684. https://doi.org/10.1145/1185448.1185597
30. Min H, Ping ZL (2019) Survey on software clone detection research. ACM Int Conf Proceeding Ser 9–16. https://doi.org/10.1145/3312662.3312707
31. Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. 28:3
32. Al-Saffar Z, Sarhan S, Elmougy S (2016) Automatic Detecting and Removal Duplicate Codes Clones. Int J Intell Comput Inf Sci 16:81–93. https://doi.org/10.21608/ijicis.2016.19841
33. Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans Softw Eng 32:176–192. https://doi.org/10.1109/TSE.2006.28
34. Sadowski C, Levin G (2007) SimHash : Hash-based Similarity Detection. Techreport 1–10
35. Baxter ID, Yahin A, Moura L, et al (1998) Clone detection using abstract syntax trees. Conf Softw Maint 368–377. https://doi.org/10.1109/icsm.1998.738528
36. Koschke R, Girard JF, Wuerthner M (1998) Intermediate representation for integrating reverse engineering analyses. Reverse Eng - Work Conf Proc 241–250. https://doi.org/10.1109/wcre.1998.723194
37. Yang W (1991) Identifying syntactic differences between two programs. Softw Pract Exp 21:739–755. https://doi.org/10.1002/spe.4380210706
38. Wahler V, Seipel D, Wolff V. Gudenberg J, Fischer G (2004) Clone detection in source code by frequent itemset techniques. Proc - Fourth IEEE Int Work Source Code Anal Manip 128–135. https://doi.org/10.1109/SCAM.2004.6
39. Evans WS, Fraser CW, Ma F (2007) Clone detection via structural abstraction. Proc - Work Conf Reverse Eng WCRE 150–159. https://doi.org/10.1109/WCRE.2007.15
40. Akmaliyah M (2013) Program slicing. J Chem Inf Model 53:1689–1699
41. Bhat MI, Sharada B (2016) Recognition of Handwritten Devanagiri Numerals by Graph Representation and SVM. Intl Conf Adv Comput Commun Informatics 1930–1935. https://doi.org/10.1109/ICACCI.2016.7732333
42. Komondoor R, Horwitz S (2000) Semantics-preserving procedure extraction. Conf Rec Annu ACM Symp Princ Program Lang 155–169. https://doi.org/10.1145/325694.325713
43. Hirschberg DS (1975) A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun ACM 18:341–343. https://doi.org/10.1145/360825.360861

44. Chen J, Dean TR, Alalfi MH (2014) How Accurate Is Coarse-grained Clone Detection?: Comparision with Fine-grained Detectors. 63:
45. Chen W, Li B, Gupta R (1390) Code composition of Machine single-entry multiple-exit regions. 99-117 ص ; 8 شماره
46. Ducasse S, Rieger M, Demeyer S (1999) Language independent approach for detecting duplicated code. Conf Softw Maint 109–118. https://doi.org/10.1109/icsm.1999.792593
47. Mayrand J, Leblanc C, Merlo EM (1996) Experiment on the automatic detection of function clones in a software system using metrics. Conf Softw Maint 244–253. https://doi.org/10.1109/icsm.1996.565012
48. Greenan K (2005) Method-Level Code Clone Detection on Transformed Abstract Syntax Trees Using Sequence Matching Algorithms. Transformation 1–17
49. Saini V, Sajnani H, Lopes C (2018) Cloned and non-cloned Java methods: a comparative study. Empir Softw Eng 23:2232–2278. https://doi.org/10.1007/s10664-017-9572-7
50. Kontogiannis KA, Demori R, Merlo E, et al (1996) Pattern matching for clone and concept detection. Autom Softw Eng 3:77–108. https://doi.org/10.1007/BF00126960
51. Lanubile F, Mallardo T (2002) Tool support for distributed inspection. Proc - IEEE Comput Soc Int Comput Softw Appl Conf 1071–1076. https://doi.org/10.1109/CMPSAC.2002.1045151
52. Davey N, Barson P, Field S, Frank RJ (1995) The Development of a Software Clone Detector. Int J Appl Softw Technol 1:219–236
53. Choi E, Yoshida N, Higo Y, Inoue K (2015) Proposing and evaluating clone detection approaches with preprocessing input source files. IEICE Trans Inf Syst E98D:325–333. https://doi.org/10.1587/transinf.2014EDP7292
54. Li D, Piao M, Shon HS, et al (2014) One pass preprocessing for token-based source code clone detection. 2014 IEEE 6th Int Conf Aware Sci Technol iCAST 2014. https://doi.org/10.1109/ICAwST.2014.6981824
55. Elen A, Avuçlu E (2021) Standardized Variable Distances: A distance-based machine learning method. Appl Soft Comput 98:106855. https://doi.org/10.1016/j.asoc.2020.106855
56. Bellon S, Koschke R, Antoniol G, et al (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33:577–591. https://doi.org/10.1109/TSE.2007.70725
57. Jain, Anil K. RCD (2000) Algorithms for Clustering Data