Computers, Materials & Continua
DOI:10.32604/cmc.2022.022659
Article

Tech Science Press

SSA-HIAST: A Novel Framework for Code Clone Detection

Neha Saini* and Sukhdip Singh

Deenbandhu Chhotu Ram University of Science and Technology, Murthal, 131001, India
*Corresponding Author: Neha Saini. Email: neha3998akalacademy@gmail.com
Received: 14 August 2021; Accepted: 29 September 2021

Abstract: In the recent era of software development, reusing software is one of the major activities that is widely used to save time. To reuse software, the copy and paste method is used and this whole process is known as code cloning. This activity leads to problems like difficulty in debugging, increase in time to debug and manage software code. In the literature, various algorithms have been developed to find out the clones but it takes too much time as well as more space to figure out the clones. Unfortunately, most of them are not scalable. This problem has been targeted upon in this paper. In the proposed framework, authors have proposed a new method of identifying clones that takes lesser time to find out clones as compared with many popular code clone detection algorithms. The proposed framework has also addressed one of the key issues in code clone detection i.e., detection of near-miss (Type-3) and semantic clones (Type-4) with significant accuracy of 95.52% and 92.80% respectively. The present study is divided into two phases, the first method converts any code into an intermediate representation form i.e., Hash-inspired abstract syntax trees. In the second phase, these abstract syntax trees are passed to a novel approach "Similarity-based self-adjusting hash inspired abstract syntax tree" algorithm that helps in knowing the similarity level of codes. The proposed method has shown a lot of improvement over the existing code clones identification methods.

Keywords: Code cloning; clone detection; hash inspired abstract syntax tree; rotations; hybrid framework

1 Introduction

Authors are Computer Industry has grown significantly over the past years. High-quality software and operating systems have a major role in driving this growth. Present software and operating systems are composed of millions of lines of code (LOC) that work to achieve a common objective with high efficiency and effectiveness. Software's are written using different programming languages like C, C++, JAVA, Python, etc. and the life cycle of software has multiple phases in it. Starting with detailed requirement analysis to designing, coding, testing, and ending with maintaining it. Research [1,2] has shown that out of the above-mentioned components of the life cycle of software development maintenance is the costliest part in terms of money and man-hours involved to carry it out.

Software maintenance is highly dependent on the practices that were used to build the software. One such practice those programmers use to write codes for software is code cloning. Code cloning is the process of using similar code fragments repeatedly in an application with some or no modifications at all. Research points out that 7–23% of codes are cloned in large-scale systems [3]. No Line breaks between paragraphs belonging to the same section.

1.1 Benefits of Code Cloning

Apart from ease of maintenance in the future, code cloning offers several other benefits like improvement in software metrics, low compilation time, less cognitive load, less human error, and fewer code fragments that are forgotten or missed. Code cloning has its roots in changing paradigms of programming languages i.e., higher use of templates in programming [2].

1.2 Drawbacks of Code Cloning

To begin with, code cloning makes it extremely hard to perform modifications in codes for maintenance purposes. In a high code cloned system, for a certain modification to be done a programmer has to carefully perform the modifications in all the cloned sub-systems. This phenomenon is also known as "bug propagation" [4].

1.3 Types of Code Similarity

Designing an effective code clone detection system requires an understanding of principles on which two codes are considered to be similar or clones of each other.

1.3.1 Syntactic Similarity

Two codes are said to be similar syntax-wise if they are similar textually.

- Type 1 Clones:

Also known as "exact clones", these are code clones that differ only in terms of white spaces and or addition/deletion of comments [5].

- Type 2 Clones:

Also known as "parameterized clones" these are code clones that are slightly modified by changing variables, methods, or class names. For example, code fragments such as "a=b+2" & "d=e+2" are Type 2 clones [6].

- Type 3 Clones:

Also known as "gapped clones" are code clones that differ at the statement level. Here code fragments have statements either added, edited/modified, and or deleted in addition to Type 2 differences [7].

1.3.2 Semantic Similarity

Two codes are said to be similar semantically if they are similar on a functional level while completely different textually. These are Type 4 clones and are the hardest to find. For example, Tab. 1 shows a sample python program to find the factorial of a number using recursion, and a python program to find factorial of a number using for loop (without recursion) can be considered as Type 4 clones.

Table 1: Example Type-4 Clone

Factorial using recursion	Factorial using for loop

def recur_factorial(n):
    if n == 1:
        return n
    else:
return n∗recur_factorial(n-1)
num = 7
recur_factorial(num))
	
	num = 7
	factorial = 1
	for i in range(1, num + 1):
	factorial = factorial∗i

1.4 Issues with Existing Work

Present code clone detection techniques have the following limitations:

1.4.1 Sensitivity to Type 3 & 4 Clones

Most of the literature [8–10] studied by us focuses on Type 1 & 2 clones.

1.4.2 Runtime Complexity

Techniques like CCFinder [11], VUDDY [12], SeClone [13], TwinFinder [14], Deckard [15] have high complexities in terms of usage of memory and processing power.

2 Related Work

This section details the state-of-the-art techniques for Type 3 & 4 code clones' detectors along with work done in the code clone detection using machine learning.

According to work done by Urak, most of the code plagiarism detection is limited by a variety of source codes that they can process [16]. Furthermore, most of the techniques used for semantic code clone detection are unable to provide a heuristic solution for problems varying from statement reordering, inversion of control predicates, insertion of non-useful statements. All these could cause a bottleneck in the environment. To handle these issues tekchandani proposed a novel approach that uses data flow analysis based on liveness analysis & reaching definition for detecting semantic clones in a procedure or a program [17].

In [18] Keivanloo et al. suggested the k-means clustering method as a replacement for the threshold-based cutoff phase in the clone identification process. Previous work on clone detection solved the scalability issue. As a result, they suggest a technique to aid practitioners in the use of scalable Type-3 clone detection algorithms across software systems. They are particularly concerned with enhancing performance and usability. As part of the setup, k-means is used to calculate the number of anticipated clusters. The testing results suggest that using the k-means algorithm boosts performance by 12 percent.

Anil et al. [19] described a simple and effective approach for detecting precise and near-miss clones in program source code using AST. The identification of code clones is useful not only for creating more organized code fragments but also for finding domain concepts and their idiomatic implementations.

The author has presented a novel work that performs code clone genealogy evolution on OpenMRS, an e-health system based on git. The model is based on transitive closure computation using the Hadoop ecosystem [8]. The authors presented a parse tree kernel-based code plagiarism detection method. In terms of parse tree similarity, the parse tree kernel produces a similarity value between two source codes [20]. The system successfully handles structural information because parse trees include the key syntactic structure of source codes. This article makes two important contributions. First, they suggest a program source code-optimized parse tree kernel. This system, which is based on this kernel, outperforms well-known baseline systems, according to the evaluation. Second, they gathered a large number of real-world Java source codes from a programming class at a university. Two separate human annotators manually evaluated and labeled this test set to identify plagiarized codes. A code clone detection framework for detecting both code obfuscation & cloning using machine learning has been given by the authors. They use features extracted from Java Bytecode dependency graphs, program dependency graphs & abstract syntax trees [1].

In this paper, they focus on improving the scalability of code clone detection, relative to the current state-of-the-art techniques. Their adaptive prefix filtering technique improves the performance of code clone detection for many common execution parameters when tested on common benchmarks. The experimental results exhibit improvements for commonly used similarity thresholds of between 40% and 80%, in the best case decreasing the execution time up to 11% and increasing the number of filtered candidates up to 63% [21].

A DeepCRM was proposed by the authors, which is a deep learning-based model for code readability and classification. DeepCRM firstly transforms source codes into integer matrices as the input to ConvNets. DeepCRM consists of three separate ConvNets with identical structures that are trained on data pre-processed in different ways. DeepCRM shows an increase of 2.4% to 17.2% from previous approaches [22].

2.1 State of the Art for Type 3 Clones

LVMapper was developed to detect large variance codes i.e., clones with relatively more differences in large source code repositories. It specifically considers the modifications that are more scattered in large codes. LVMapper makes use of seeds (small windows of continuous lines) to located and filter the candidate pairs of code clones [9]. SourcererCC is a technique based on token level granularity that uses an index to achieve scalability. SourcererCC has a precision of 86% and a recall rate of (86% – 100%) on 250MLOC [23]. CloneWorks has direct application in large-scale clone detection experiments. It can be fully customized to the user's need for representation of source code for clone detection [24]. NICAD is a lightweight clone detection approach that uses flexible pretty-printing and code normalization techniques. It uses agile parsing to remove noise and is-land grammars to select potential clones [25]. Deckard is based on the characterization of subtrees with numerical vectors and an algorithm w.r.t Euclidean distance matrix to cluster above said vectors [15].

2.2 State of the Art for Type 4 Clones

Jiang proposed a random number input approach to detect semantic clones. The key used by Jiang is of reducing code by using all possible consecutive subsequences of a code fragment [26]. Gabel proposed a scalable clone detection technique that reduces the difficult graph similarity problem to a tree similarity problem by carefully matching the Program dependency Graph(PDG) to their related structured syntax [27].

2.3 Latest Work on Code Clone Detection

Twin-Finder proposed a novel closed-loop clone detection approach that uses symbolic execution and machine learning techniques to get better results. For reducing false positives TwinFinder uses a feedback loop for formal loops to tune the machine learning algorithm. It lays special focus on false positives and was able to eliminate 99.32, 89 & 86.74% of false positives in bzip2, thttpd & Links respectively [14].

Oreo is a novel technique specifically designed for Type 4 clones that also exhibit some similarities syntax-wise. This category of clones is said to be in the Twilight zone. Oreo uses machine learning & size similarity sharding to perform clone detection [10].

Clonmel proposed a solution to code clone detection problems via learning supervised deep features [28].

3 SSA-HIAST Framework

As per the literature studied by us, most of the code clone detection techniques are comprised of two major phases. Firstly, they use a suitable technique to convert the code fragments into a suitable representation state. And secondly, they deploy an appropriate code similarity detection algorithm to detect code clones as shown in Fig 1.

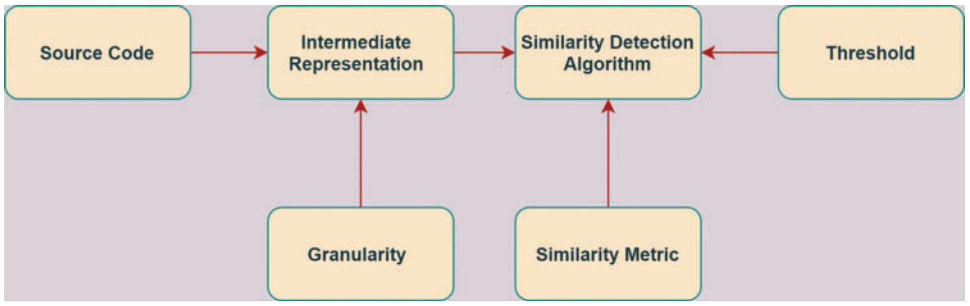

Figure 1: Overall Process followed by existing techniques

Our work is a first of its kind i.e., hybrid framework SSA-HIAST (Similarity-based self-adjusting Hash-inspired Abstract Syntax Tree) for code clone detection of Type 1, 2, 3 & 4 clones of Python programming language.

Code Repository

For the implementation of the said framework, we have used 153 open-source codes from GitHub from different repositories. We then injected Type 1, 2, 3, and 4 code clones in these 153 codes manually. Three python programmers manually injected these code clones of different lengths and logic. The three programmers were given training before injecting the clones. Also, the results of the evaluations of one programmer are cross verified by the other two programmers. To check the accuracy of the detected clones, some of the clones that were put by the programmer were selected randomly to check out if they were detected by the programmer. The entire process took 18 months.

3.1 Phase 1. Intermediate Code Representation

We use Abstract Syntax trees as the basic structure for intermediate code representation. ASTs represent the logical structure of source code and are created from a token stream. Fig. 2 represents a basic AST for a sample python code. According to the best of our knowledge, there have not been any advancements to the core structure of the AST's. Hence, we introduce Hash-inspired AST (HIAST).

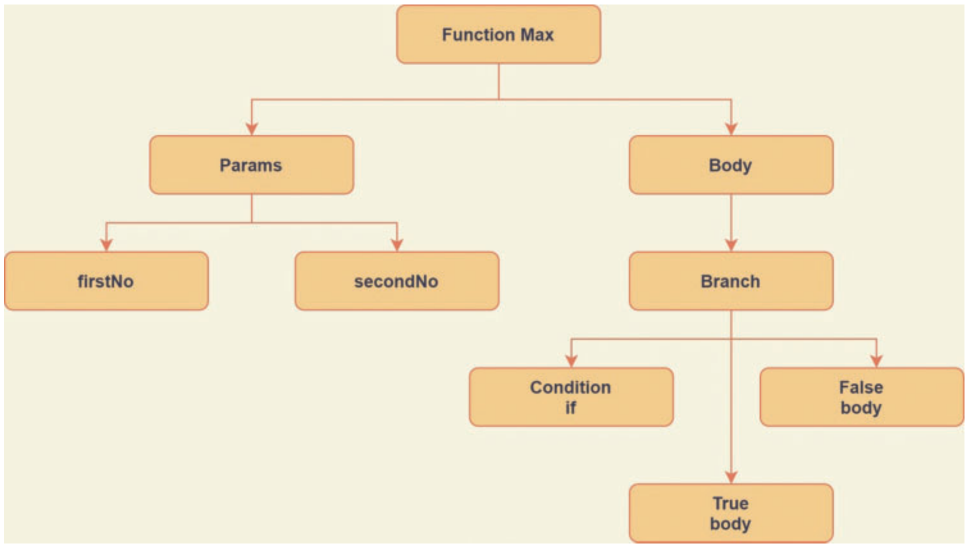

Figure 2: AST for a sample Python Code

Firstly, every source code is transformed into a parse tree representation by using appropriate syntax grammar. The HIAST helps to process trees of the Python abstract syntax grammar. The algorithm helps to programmatically what the current grammar looks like. HIAST computes a hash(object) at every stage and stores the computed hash along with the node for further input to the code matching algorithm. It also builds a hash table of the hashes which would be later used to tune the HIAST in terms of height. Also, by using HIAST, the hash of the entire code will be generated. We just need to keep track of hash values and not the entire code. This in turn will help in better management of both the software and software clones. Each node of AST is traversed in preorder for attaching hash value to it. The pseudocode for the generation of HIAST is given below:

Pseudocode 1: HIAST Generation

```
    1.  Function tranformTree(node)
    2.  {if (node == None)
    3.  return
    4.  node.hashId(attach(hash(node))
    5.  preOrder(node.left)
    6.  preOrder(node.right)
}
```

Sample python code:

```
def max(firstNo, secondNo):
    if (firstNo, secondNo):
            return firstNo
    else
            return secondNo
```

3.2 Limitation of AST

For codes expanding to millions of lines, the height of an AST can be high which can lead to higher runtimes for code similarity detection. To achieve faster processing, we have used the concept of indexing every node in AST using a dedicated hash.

3.3 Benefits of Self Adjust Feature in HIAST

Using HIAST along with rotations inspired from the AVL tree, the height of the tree will remain maintained at a certain level and will not increase unnecessarily. Due to this, the memory consumed will be lesser as compared to simple AST. Also, the time for comparisons will be less as unnecessary comparisons will reduce due to the reduction in height of the tree.

3.4 Rules for Generating HIAST

Each node in a statement from a code fragment can be represented as a record in the following way:

- operators: one field for an operator, remaining fields pointers to operands
- mknode(operator, leftOperand, rightOperand) as shown in Fig. 3.

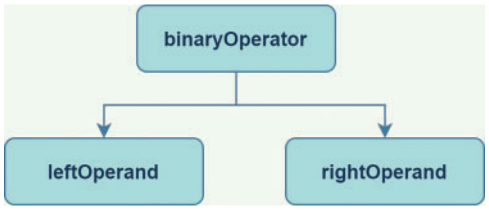

Figure 3: AST representation for operators

- Number/String: one field with label "num"/"str" and a pointer to keep the value of the number mkleaf(num/str, val) as mentioned in Fig. 4.

Number
Value

Figure 4: AST representations for literals

- Looping construct (for, while): one field for the type of looping construct and remaining field pointers for number assignment, condition & operator as in Fig. 5.

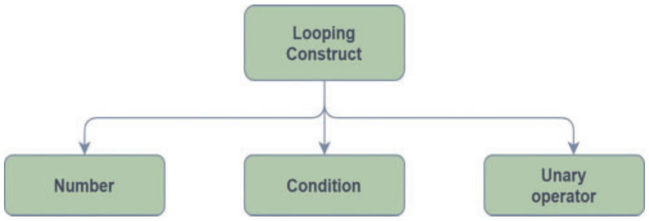

Figure 5: AST representation for looping constructs

- Condition (if, else): one field for condition and remaining field pointers to condition variables as shown in Fig. 6.

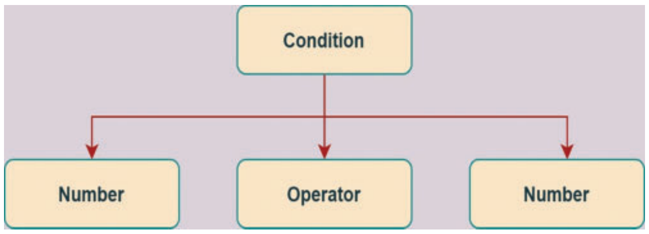

Figure 6: AST representation for conditions

- Data Structure (List, Set, Tuple, Dictionary & Array): One field for size (n) and next (n) fields for values.
- Function call: one field for the function name and remaining pointers for all the arguments as mentioned in Fig. 7

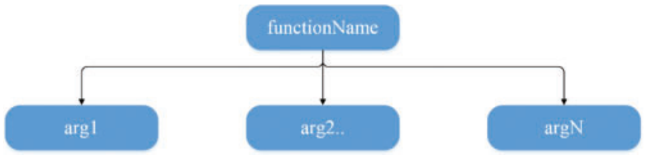

Figure 7: AST representation for function calls

- Class/Object: One field containing the object of the class.
- hashId: one field with label "hashId" and pointer to store the hash of the node currently being created: mkleaf(hashId, hashval).

For example, the statement "expression = 6 + 8" would be first converted to tokens as shown in Fig. 8 when passed to HIAST algorithm will give the AST tree as:

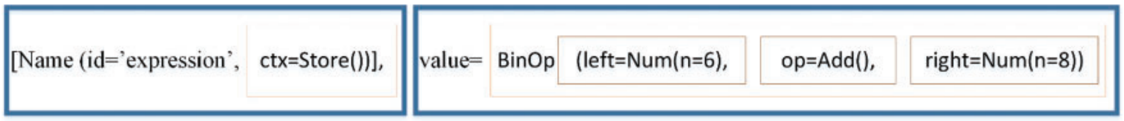
[Name (id='expression', ctx=Store())],
value= BinOp (left=Num(n=6), op=Add(), right=Num(n=8))

Figure 8: Token representation for "expression = 6 + 8

The following sequence of function calls creates AST for - expression = 6 + 8 as shown in Fig. 9.

P1= mkleaf(num, 6)

P2= mkleaf(num, 8)

P3= mknode(+, P1, P2)

P4= mkleaf(store,P3)

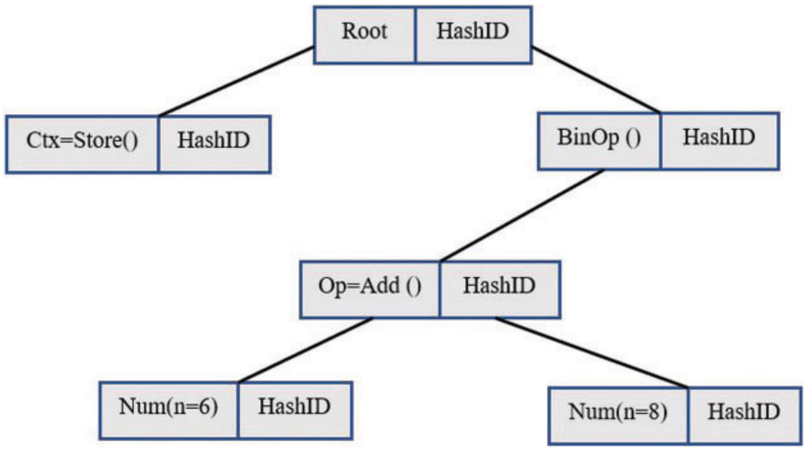

Figure 9: HIAST representation for "expression = 6 + 8"

The pseudocode for the generation of AST is given below. It scans the code and inserts a node in the tree depending on the type of token.

Pseudocode 2: Pseudocode for AST generation

```
Input: Token Sequence
Output: AST
Define a function insert_node(obj):
  If obj.type == 'operator'
        Tree.makeNode(obj, hash(obj))
elif obj.type == 'loop'
          Tree.makeNode(obj.type, hash(obj))
       Tree.makeLeaf(obj.number)
           Tree.makeLeaf(obj.condition)
          Tree.makeLeaf(obj.operator)
elif obj.type == 'condition'
          Tree.makeNode(obj.type, hash(obj))
      Tree.makeLeaf(obj.number)
             Tree.makeLeaf(obj.operator)
         Tree.makeLeaf(obj.number)
elif obj.type == 'number'
        Tree.makeNode(obj.type, hash(obj))
       Tree.makeLeaf(obj.value)
elif obj.type == 'function'
              Tree.makeNode(obj.type, hash(obj))
          For every argument in obj.type:
         Tree.makeLeaf(obj.argument)
```

3.5 Phase 2: Code Similarity Detection

Once the codes to be checked are in suitable representation, we then apply an effective (high recall & precision) code similarity detection algorithm. To the best of our knowledge, we are the first ones to use the "Self-adjusting" feature in AST's using a similarity score. Large codes can generate AST's that are consume a lot of memory due to the high depth of the tree generated.

Large code systems have the same code clones used in multiple parts of the codes. With this as a motivation, we decided to restructure a code by adjusting similar code fragments up or down the order in the original file. Most of the previous work applied similarity detection on code fragments individually. We here introduce a novel technique to apply similarity detection of a code file instead of a code fragment. Also, there are few existing techniques on code file similarity, but the proposed work focuses on large code systems with an efficient similarity detection approach.

3.5.1 Similarity Metric

Given two code files c1 & c2, the similarity between the two code fragments representing a subtree of AST is defined in Eq. (1).

$$\mathrm{Sim}(\mathrm{Subtree\ c1},\ \mathrm{Subtree\ c2}) = 2\mathrm{SN}/(2\mathrm{SN} + \mathrm{L} + \mathrm{R}) \tag{1}$$

where "SN" is the number of shared nodes in subtree T1 & T2 ;{L:[t1, t2 .... tn] , R[t1, t2........tn]}

3.5.2 Syntactic Similarity

A similarity detection algorithm requires a Threshold function along with a similarity detection algorithm. The threshold function is used to decide the optimum level for similarity check as shown in Eq. (2).

$$\mathrm{Th} = \sqrt{}(\min(\mathrm{Sim}(\mathrm{Ci}), \mathrm{Sim}(\mathrm{Cj}) * 2(1 - \mathrm{Sim}))) \tag{2}$$

Our self-tree readjusting algorithm performs necessary rotations on the HIAST to effectively compare two subtrees for clones. Rotations are performed based on values of threshold (Th). An upper limit has been set for the number of rotations to be performed to prevent resource exhaustion. With "label" fields in the node structure, we can bring two subtrees under comparison to the same level by performing certain rotations.

We have used Latent Semantic Indexing [29] based on the Euclidean distance between two vectors to cluster a vector group given a set of characterstic vectors. Assume two feature vectors FVeci and FVecj each represents two code snippets CSi and CSj. Size(CSi) and Size(CSj) represent the code size (the total number of AST nodes). E is the euclidean distance between FVeci and FVecj ([FVeci; FVecj]). Given a feature vector group VG, the threshold may be reduced to $\sqrt{}$(min$f$()(Sim(Ci), Sim(Cj)∗2(1-Sim))), where vector sizes are used to estimate tree sizes. The Sim is the code similarity measure given by Eq. (1). Thus, if E([Veci; Vecj]<=Th, code fragments CSi and CSj will be grouped as code clones under a certain code similarity Sim. There are four types of rotations that are used in the framework. The pseudocode for the rotation set used is given below.

Pseudocode 3: Rotation Set used in SSA-HIAST

```
function RotateToRight (current)
N_Root = c → left
c → left = N_Root->right
N_Root → right = c
return N_Root
    end function
function RotateToLeft (c)
N_Root = c → left
c →right = N_Root →left
N_Root → left = c
return N_Root
    end function
function RotateToLeftRight(c)
c→ left = leftRotate(c → left)
return rightRotate(c)
    end function
function RotateToRightLeft(c)
current → right = rightRotate(c → right)
return leftRotate(c)
    end function
```

The pseudocode for the similarity algorithm used is given below. First code files are scanned. Then they are tokenized. After tokenization, the HIAST of the two code files is generated and compared based on similarity metric and threshold.

Pseudocode 4: Similarity-based Self Adjust in HIAST

```
Step 1. Input two code files Fi, Fj,
Similarity_Threshold=$\alpha$
Step 2. Initialize Python tokenizer <T>
Step 3. For (linei, linej) in (Fi,Fj):
         TokenSet Tsi = T < linei>
         TokenSet Tsj = T < linej>
Step 4.   Set Sim = 0
Step 5. For (obji, objj) in (Tsi, Tsj):
T1= Call insert_node(obji)
T2 = Call inert_node(objj)
Sim = Calculate_Similarity(T1,T2)
Step 6. Check( If(T1==T2) ) as per Similarity_Threshold i.e., (Sim> $\alpha$)
Return Clone Found
else
Move to Step 7
```

(Continued)

```
Step 7. For rotation in Rotation set:
Apply (rotation, T1, T2)
Update Sim
Move to Step 6
Step 8. Return No Clone Found
```

The example of two codes for comparison is given in Tab. 2. The similarity of code 1 and code 2 is explained with the help of rotations in Fig. 10.

Table 2: Example Type-4 Clone

Code 1: a - 4 + c	Code 2: a + c - 4
P1= mkleaf(id, a)	P1= mkleaf(id, a)
P2= mkleaf(num, 4)	P2= mkleaf(id, c)
P3= mknode(-,P1,P2)	P3= mknode(+,P1,P2)
P4= mkleaf(id, c)	P4= mkleaf(num, 4)
P5= mknode(+,P3,P4)	P5= mknode(-,P3,P4)

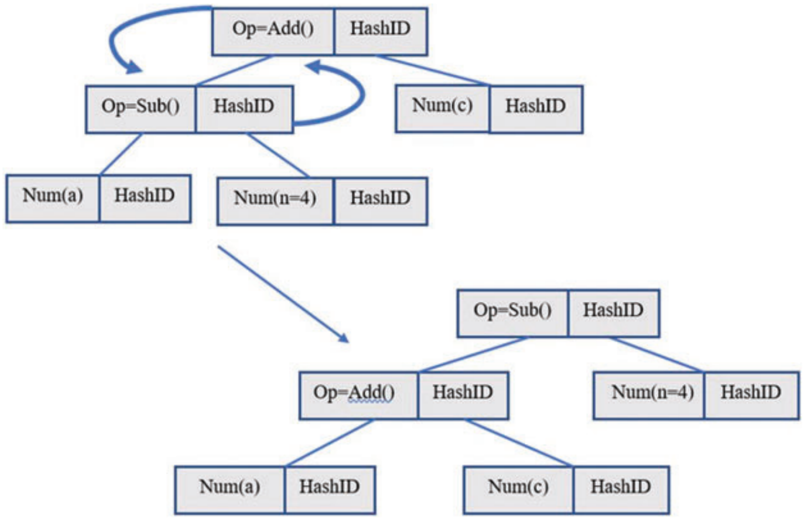

Figure 10: Rotation being performed in a HIAST

4 Experimentation and Results

For experimentation, we have used 153 python codes from different publicly available repositories on GitHub. This section presents a detailed analysis of the results of the SSA-HIAST approach. A similarity detection algorithm requires a threshold value along with a similarity detection algorithm. The threshold value is used to decide the optimum level for similarity check. Similarity detection algorithm requires a Similarity threshold for comparing with similarity metric Sim. If the value of Sim comes out to be greater than Similarity Threshold, then we have a match otherwise we don't have a match. In our framework, the two codes are clones if they are 90% similar. Also, Rotation Threshold is required to prevent the algorithm from going into an infinite loop. Our self-tree readjusting algorithm performs necessary rotations on the HIAST to effectively compare two subtrees for clones. With "label" fields in the node structure, we can bring two subtrees under comparison to the same level by performing certain rotations. Our self-tree readjusting algorithm performs necessary rotations on the HIAST to effectively compare two subtrees for clones. Rotations are performed based on values of threshold (Th). An upper limit has been set for the number of rotations to be performed to prevent resource exhaustion.

4.1 Experimental Setup

In all the 153 python codes, three different python programmers injected code clone fragments that had a variable function and class name modifications. Furthermore, for, while & if statements were changed to their synonym's expressions and useless line (600–2000) were added. The system used to detect clones was the Intel i5(2.7Ghz) based machine with 16GB of RAM running Ubuntu 18.04 LTS.

4.2 Evaluation Criteria

The framework is evaluated on the basis of various parameters discussed below:

- **Clone Quantity**: No of detected clones
- **Clone Quality**: No of false positives
- **Precision**: The ratio of true positives to all positives is known as precision. In our case, it is the number of clones that are correctly identified out of all the clones present.
  Precision=True Positive/ (True Positive +False Positive)
- **Recall:** It is a test of how well our model detects True Positives. In our case, it is all the clones present, to the how many correctly identified clones are there.
  Recall=True Positive/ (True Positive +False Negative)

Based on evaluation criteria, the results are depicted in Figs. 11–14, as the comparison of injected clone with detected clone in a program is shown in the case of Type 1, Type 2, Type 3 and Type 4 clones respectively, which directly depicts the accuracy and efficiency of the model in detecting the clones in a program. The results obtained after the application of our framework for software clone detection for type 1 clones can be seen in Fig. 11. There is a slight difference between the peaks of injected and detected clones as can be seen in Fig. 11. Our proposed framework has shown impressive results for Type-1 clone detection. The accuracy achieved for Type-1 clone detection is 97.23%. The framework can detect most of the type-1clones correctly. Similarly, the results for Type-2 clone detection are summarized in Fig. 12. Although the proposed framework has shown good results for Type-2 clone detection also, the accuracy achieved for Type-2 clone detection is lesser as compared to Type-1 clone detection. The accuracy achieved for Type-2 clone detection is 96.74%.

Type-3 clones, also known as near-miss are a bit difficult to identify as compared to Type-1 and Type-2 clones. Result analysis for Type-3 clone detection is shown in Fig. 13. The accuracy achieved is also lesser compared to the detection of Type-1 and Type-2 clones. The accuracy achieved is 95.52%. The Type-4 clones are the most difficult to detect and manage as they are based on semantic similarity of code. The accuracy achieved by using the proposed framework for Type-4 clone detection is 92.80%. The difference between the peaks of injected and detected clones is also largest for Type-4 clone detection as evident from Fig. 14.

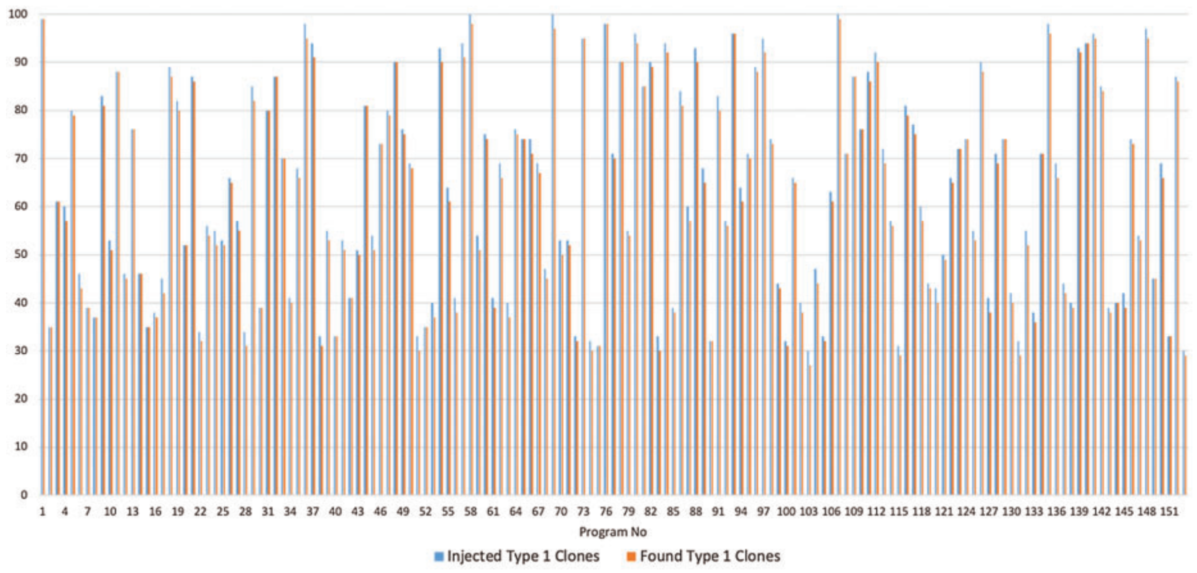

Figure 11: Comparison Injected v/s Detected Type 1 clones

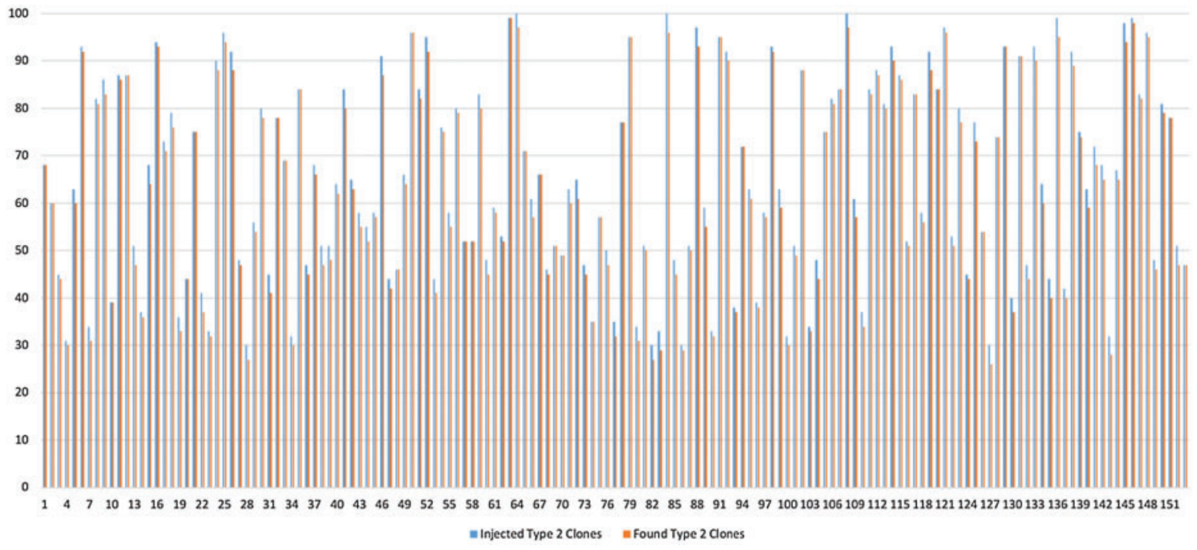

Figure 12: Comparison of Injected v/s Detected Type 2 clones

Moreover, the proposed algorithm is compared with the classic algorithms in Tab. 3 in terms of space and time complexity. It can be seen from the table that the space complexity of CP-Miner is directly proportional to the number of lines of code, whereas the space complexity of the proposed SSA-HIAST algorithm is directly dependent on the number of nodes of the tree. Hence the proposed algorithm is better than CP-Miner in terms of space utilization. Also, CP-Miner has quadratic run time complexity, whereas SSA-HIAST has linear run time complexity.

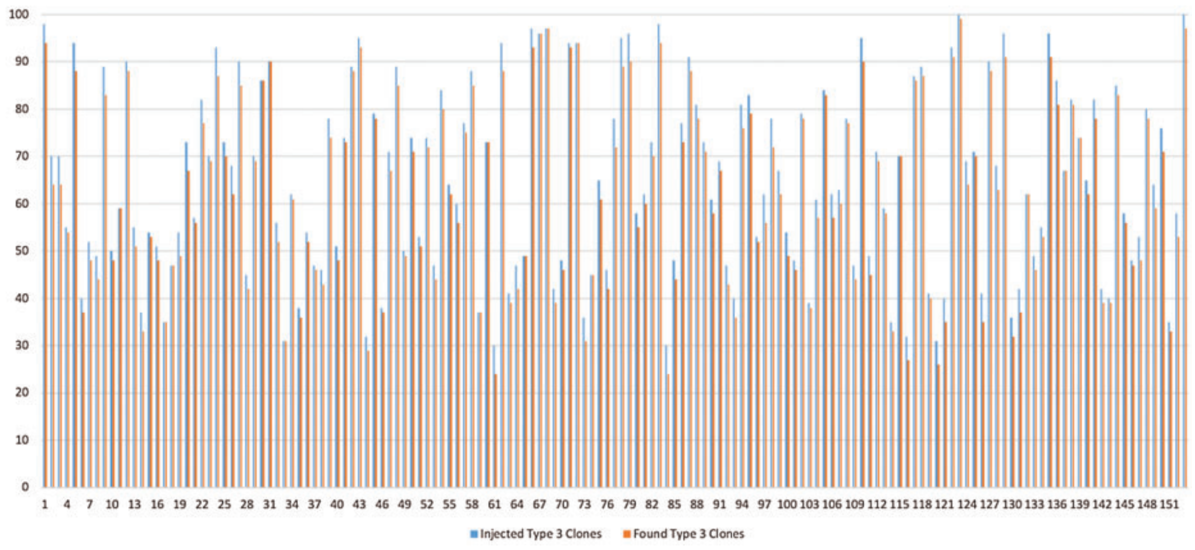

Figure 13: Comparison of Injected v/s Detected Type 3 clones

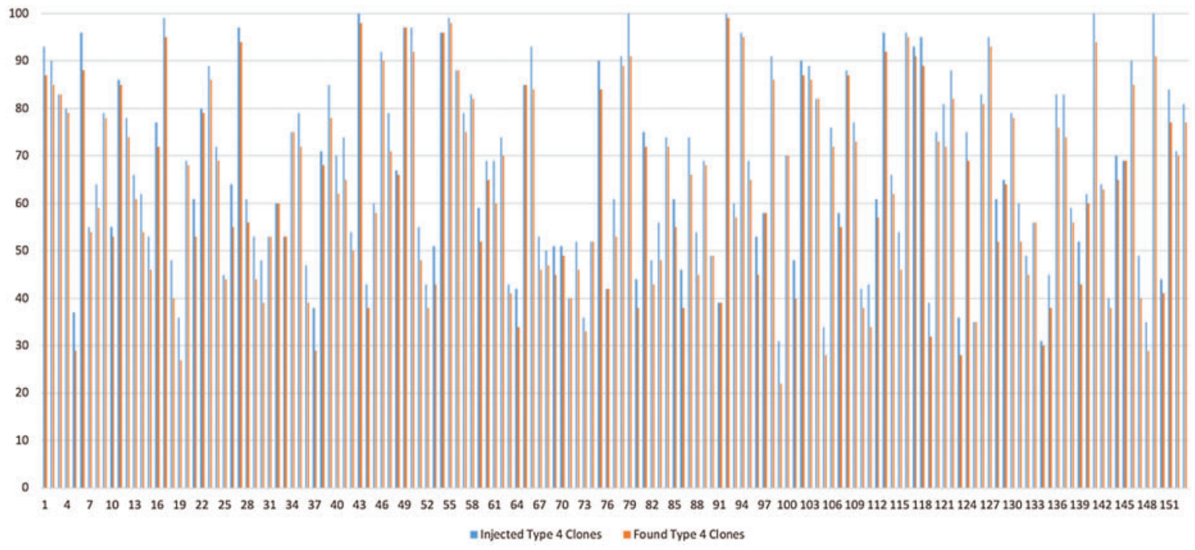

Figure 14: Comparison of Injected v/s Detected Type 4 clones

The precision and recall values for different clone types are given in Fig. 15. The precision and recall for type-1 clone detection are 97.52% and 94.93%, for type -2 precision and recall values are 96% and 92.8% respectively which are comparatively lesser as compared to type-1. For type-3 clone detection, the proposed framework has achieved precision and recall of 95.9% and 91.2% respectively. The least precision and recall values have been achieved for type-4 clone detection which is 94.5% and 87.6% respectively.

Table 3: Comparative Analysis

Algorithm	Space complexity	Run time complexity
CP-Miner [30]	O(r)	$O(r^2)$
Clone DR [31]	O(s)	$O(r^2/\|Buckets\|)$
LSH [32]	$O(s^{p+1} + ks)$	$O(ks^p \log s)$
LSH w/grouping [33]	$O(\max_{v\in G\,\|v\|}{}^{p+1} + k\|v\|\ )$	$O(k\sum v\in G\ \|v\|^p\ log\|v\|)$
DECKARD w/Post-Processing [15]	$max\{O(c\|rcAN\|),\ O\ _{v\in G}\ ,(_{\|v\|}{}^{p+1} + k\|v\|)\}$	$O(s + k\sum v\in G\|v\|^{\rho+1} \log\|v\| + c\|rcAN\|^2)$
DP-matching [34]	$O(\max_{v\in G\,\|v\|}{}^{p+1} + k\|v\|\ )$	$O(k\sum v\in G\ \|v\|^p\ log\|v\|)$
Event checking [35]	$O(s^{p+1} + ks)$	$O(ks^p \log s)$
Normalisation pipeline [36]	$O(s^{p+1} + ks)$	$O(ks^p \log s)$
Context-sensitive pointer analysis [37]	$s\ O(n\alpha(n, n))$	O(n) s
SourcererCC [38]	$O(n^2)$	$O(n^2)$
Autoencode [38]	$O(n^2)$	$O(n^2)$
SSA-HIAST	O(s)	$O(s + \log\|Buckets\|)$

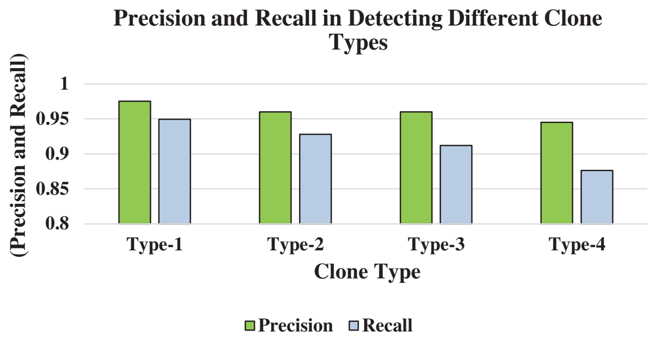

Figure 15: Precision *vs.* Recall in Detecting Different Clone Types

Worst-case complexities of CloneDR, CP-Miner, and DECKARD (r is the number of lines of code, s is the size of a parse tree, $|Buckets|$ is the number of hash tables used in CloneDR, k is the number of node kinds, $|v|$ is the size of a vector group, $0 < \rho < 1$, c is the number of clone classes reported, and $|rcAN|$ is the average size of the clone classes).

4.3 Benchmarking Against the State of the Art

The benchmark SSA-HIAST is compared with the state of art and the results are shown in Tab. 4. Along with this, a comparison table has also been developed for comparing performance metrics of the proposed algorithm with some of the pre-existing models as shown in Tab. 5 and found that the proposed model outperforms all other models in terms of performance metrics also.

Table 4: Benchmarking SSA-HIAST

Benchmark	Deckard %age clone detection	Twin-Finder %age clone detection	SSA- HIAST %age clone detection
bzip2	32.5	62.15	68.75
sphinx3	32.28	75.89	77.85
hmmer	27.19	59.55	62.24
Thhtpd	29.13	51.64	64.78
Gzip	9.57	40.15	39.84
Man	14.74	49.08	55.47
Links	22.69	64.71	64.5

Table 5: Comparison of the proposed model with pre-existing models in terms of performance metrics

Study	KLOC	Precision(%)	Recall(%)
Dup [39]	27	80	80
CCFinder [40]	21	99	93
Duploc [39]	23	90	86
DP matching [34]	28	87	83
SourcererCC [38]	26	82	79
Autoencode [38]	30	81	76
Proposed algorithm	35	99	95

5 Conclusion and Future Scope

The proposed system SSA-HIAST has achieved higher clone detection rates than the popular and established Deckard and Twin Finder clone detection techniques. For comparison of the clone detection rate, various projects have been considered. The proposed system has surpassed clone detection techniques in almost all projects. Moreover, the proposed algorithm has been tested on 153 python codes that have been publicly taken from the GitHub repositories. The results have been evaluated on the criteria of the number of false positives as well as the number of clones detected. The proposed algorithm can detect type-4 clones with an accuracy of 92.8%. The space complexity of the proposed algorithm is O(s) where s is the number of nodes of HIAST and the runtime complexity of our algorithm is (O(r+slog(buckets)). The proposed Framework outperforms other works like Dup & Duploc in terms of precision and recall and CCFinder in terms of recall. In the future, we are planning to conduct the same experiment using a hybrid deep learning approach, by combining two or more techniques for code clone detection and management.

Along with this, we will focus to extend this work to some other programming languages like Java, R, and C as this framework works for python language only. After detecting and prioritizing true clones, this framework can be further strengthened by employing clone eradication strategies.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. Sheneamer, S. Roy and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 405–420, 2018.

[2] D. Rattan, R. Bhatia and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[3] J. Akram, Z. Shi, M. Mumtaz and P. Luo, "DCCD: An efficient and scalable distributed code clone detection technique for Big code," in *30th Int. Conf. on Software Engineering and Knowledge Engineering*, Redwood City,California, USA, pp. 354–360, 2018.

[4] E. Kodhai and S. Kanmani, "Method-level code clone detection through LWH (Light weight hybrid) approach," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, pp. 1–29, 2014.

[5] Q. U. Ain, F. A. W. Haider, M. W. Butt, B. Anwar and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, pp. 86121–86144, 2019.

[6] A. Cuomo, U. Villano and A. Santone, "A novel approach based on formal methods for clone detection," in *Int. Workshop on Software Clones (IWSC)*, Zurich, Switzerland, pp. 8–14, 2012.

[7] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *31st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Singapore, pp. 87–98, 2016.

[8] R. Tekchandani, R. Bhatia and M. Singh, "Code clone genealogy detection on e-health system using hadoop," *Computers and Electrical Engineering*, vol. 61, pp. 15–30, 2017.

[9] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu *et al.*, "LVMapper: A large-variance clone detector using sequencing alignment approach," *IEEE Access*, vol. 8, pp. 27986–27997, 2020.

[10] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *26th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, New York, United States, pp. 354–365, 2018.

[11] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transcations on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[12] S. Kim, S. Woo, H. Lee and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, pp. 595–614, 2017

[13] I. Keivanloo, J. Rilling and P. Charland, "Seclone - A hybrid approach to internet-scale real-time code clone search," in *19th IEEE Int. Conf. on Program Comprehension*, Kingston, Ontario, Canada, pp. 223–224, 2011.

[14] H. Xue, G. Venkataramani and T. Lan, "Twin-finder: Integrated reasoning engine for pointer-related code clone detection," *in ArXiv Prep*, pp. 1–7, 2019.

[15] L. Jiang, G. Misherghi, Z. Su and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *29th Int. Conf. on Software Engineering (ICSE'07)*, Minneapolis, MN, pp. 96–105, 2007.

[16] M. Duračík, E. Kršák and P. Hrkút, "Current trends in source code analysis, plagiarism detection and issues of analysisšBig datasets," *Procedia Engineering*, vol. 192, pp. 136–141, 2017.

[17] R. Tekchandani, R. Bhatia and M. Singh, "Semantic code clone detection for internet of things applications using reaching definition and liveness analysis," *Journal of Supercomputing*, vol. 74, no. 9, pp. 4199–4226, 2016.

[18] I. Keivanloo, F. Zhang and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *22nd Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, Montreal, QC, Canada, pp. 201–210, 2015.

[19] I. G. Anil, C. Reddy and A. Govardhan, "Software code clone detection using ast," *International Journal of P2P Network Trends and Technology*, vol. 4, no. 3, pp. 33–39, June 2014.

[20] J. W. Son, T. G. Noh, H. J. Song and S. B. Park, "An application for plagiarized source code detection based on a parse tree kernel," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 8, pp. 1911–1918, 2013.
[21] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, pp. 130–142, 2018.
[22] Q. Mi, J. Keung, Y. Xiao, S. Mensah and Y. Gao, "Improving code readability classification using convolutional neural networks," *Information and Software Technology*, vol. 104, pp. 60–71, 2018.
[23] V. Saini, H. Sajnani, J. Kim and C. Lopes, "SourcererCC and SourcererCC-i: Tools to detect clones in batch mode and during software development," in *38th Int. Conf. on Software Engineering Companion (ICSE-C)*, Austin, TX, USA, pp. 597–600, 2016.
[24] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *IEEE/ACM 39th Int. Conf. on Software Engineering Companion*, Buenos, Aires, Argentina, pp. 27–30, 2017.
[25] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *IEEE Int. Conf. on Program Comprehension*, Amsterdam, Netherlands, pp. 172–181, 2008
[26] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *18th Int. Symposium on Software Testing and Analysis*, Amsterdam, pp. 81–91, 2009.
[27] M. Gabel, L. Jiang and Z. Su, "Scalable detection of semantic clones," in *Proc. of the 30th Int. Conf. on Software Engineering*, Leipzig, Germany, pp. 321–330, 2008.
[28] H. H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Int. Joint Conf. on Artificial Intelligence (IJCAI-17)*, Melbourne, Australia, pp. 3034–3040, 2017.
[29] S. Dumais, "Latent semantic indexing (LSI) and TREC-2," *Nist Special Publication Sp*, pp. 105–105, 1994.
[30] Z. Li, S. Lu, S. Myagmar and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.
[31] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna and L. Bier, "Clone detection using abstract syntax suffix trees," in *13th Working Conf. on Reverse Engineering*, Benevento, Italy, pp. 253–262, 2006.
[32] M. Datar, P. Indyk, N. Immorlica and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Twentieth Annual Symposium on Computational Geometry*, Brooklyn, New York, USA, pp. 253–262, 2004.
[33] A. Gionis, P. Indyk and R. Motwani, "Similarity search in high dimensions via hashing," in *25th Int. Conf. on Very Large Data Bases*, Edinburgh, Scotland, pp. 518–529, 1999.
[34] T. Lavoie, M. Eilers-Smith and E. Merlo, "Challenging cloning related problems with gpu-based algorithms," in *Int. Workshop on Software Clones*, Cape Town, South Africa, pp. 25–32, 2010.
[35] D. Chen, S. J. Turner, W. Cai, B. P. Gan and M. Y. H. Low, "Algorithms for HLA-based distributed simulation cloning," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 15, pp. 316–345, 2005.
[36] E. Juergens, B. Hummel and F. Deissenboeck, "Clonedetective-a workbench for clone detection research," in *Int. Conf. on Software Engineering*, Vancouver, BC, Canada, pp. 603–606, 2009.
[37] C. Lattner, A. Lenharth and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 6, no. 42, pp. 278–289, 2007.
[38] W. Rahman, "Clone detection on large scale codebases," in *Int. Workshop on Software Clones*, Ontario, Canada, pp. 38–44, 2020.
[39] H. Liu, Z. Ma, L. Zhang and W. Shao, "Detecting duplications in sequence diagrams based on suffix trees," in *Asia Pacific Software Engineering Conf.*, Bangalore, India, pp. 269–276, 2016.
[40] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.