Comparison and Evaluation of Clone Detection Techniques with Different Code Representations

Yuekun Wang[1], Yuhang Ye[1], Yueming Wu[2,*], Weiwei Zhang[1], Yinxing Xue[1,*], Yang Liu[2]
[1] University of Science and Technology of China, China
[2] Nanyang Technological University, Singapore

***Abstract*—As one of bad smells in code, code clones may increase the cost of software maintenance and the risk of vulnerability propagation. In the past two decades, numerous clone detection technologies have been proposed. They can be divided into text-based, token-based, tree-based, and graph-based approaches according to their code representations. Different code representations abstract the code details from different perspectives. However, it is unclear which code representation is more effective in detecting code clones and how to combine different code representations to achieve ideal performance.**

In this paper, we present an empirical study to compare the clone detection ability of different code representations. Specifically, we reproduce 12 clone detection algorithms and divide them into different groups according to their code representations. After analyzing the empirical results, we find that token and tree representations can perform better than graph representation when detecting simple code clones. However, when the code complexity of a code pair increases, graph representation becomes more effective. To make our findings more practical, we perform manual analysis on open-source projects to seek a possible distribution of different clone types in the open-source community. Through the results, we observe that most clone pairs belong to simple code clones. Based on this observation, we discard heavyweight graph-based clone detection algorithms and conduct combination experiments to find out a suitable combination of token-based and tree-based approaches for achieving scalable and effective code clone detection. We develop the suitable combination into a tool called TACC and evaluate it with other state-of-the-art code clone detectors. Experimental results indicate that TACC performs better and has the ability to detect large-scale code clones.

***Index Terms*—Clone Detection, Empirical Study, Code Representation, Large Scale**

I. Introduction

Nowadays, copying existing code fragments and pasting them into other parts of the code is a common behavior in software development. The reusing code, which is identical or similar to the source code, is called *code clone* [1]. Although previous research [2] has demonstrated that code clone does have some potential benefits, code clones have been shown to be bad smells: the growth in code clones can lead to the propagation of potential software defects that reduces the reliability and maintainability of software systems [3]. Therefore, the common wisdom is that code clones need to be detected and managed.

To implement clone management and reduce the risks caused by code clones, clone detection has been active in the field of software engineering [4]. In practice, there have been proposed many code clone detection methods which can be classified into five main categories according to different code representations. They are text-based, token-based, *abstract syntax tree* (AST-based), *control flow graph* (CFG-based), and *program dependency graph* (PDG-based). For example, Johnson *et al.* [5] directly hash substrings of source text content and use a sliding window to find similar files. CCFinder [6] extracts token sequences from source codes through lexical analysis and normalizes the code using predefined rules to detect clones. Koschke *et al.* [7] find syntactic clones in linear time by serializing AST subtrees and building suffix trees. CCsharp [8] first simplifies PDGs and then identifies clone pairs by computing subgraph isomorphisms. Different code representations embody different levels of abstraction of the source code, which affect the capability of clone detection due to the retention of different program information. However, it is unclear which code representation is more effective in detecting code clones. In other words, there is a lack of clarity on the capabilities of clone detection techniques based on different representations.

Code clone is generally classified into four generally accepted types in the literature [1], [9]. The first three types are syntactic code clone and the last type belongs to semantic code clone. Studies [10], [11] have shown that Type-3 clones account for more in some open source software systems. Nevertheless, there is a lack of clarity on the true distribution of different types of code clones in the open-source community. The distribution of real-world clones is important, allowing researchers to conduct more targeted studies and develop more practical tools. For example, if most of Type-3 clones in the open-source community are simple Type-3, that is, the amount of modified code is small, then it is not necessary for researchers to design those complicated clone detection tools for real-world code clone scanning. In addition, with the vigorous development of the open source ecosystem, the scale of code reuse continues to expand, and the software system becomes more and more complex, making large-scale clone detection particularly necessary [12], [13]. In fact, numerous clone detection technologies based on different code representations have been designed to date, and they both have advantages and disadvantages. For example, text-based and token-based methods have better scalability while AST-based, CFG-based, and PDG-based tools are more accurate in detecting code clones. A feasible idea is to combine the

* Yueming Wu and Yinxing Xue are the corresponding authors.

advantages of different techniques to achieve scalable yet accurate code clone detection. Regrettably, there is a lack of clarity on how to combine methods of different representations to accomplish large-scale clone detection.

In this paper, we conduct a comprehensive empirical study to thoroughly compare the ability of technologies based on different code representations and find the most suitable combination to achieve ideal code clone analysis for the open-source community. Specifically, we mainly study the following three research questions:

- *RQ1: Which code representation is more effective in detection code clones?*
- *RQ2: What is the distribution of different clone types in the open-source community?*
- *RQ3: How to combine different methods to achieve efficient yet effective large-scale code clone detection?*

To answer the first question, we reproduce 12 clone detection algorithms based on different representations. Due to the unclear boundary between Type-3 and Type-4, Jeffrey *et.al.* further divide Type-3 code clone into four subclasses (*i.e.,* Very Strongly Type-3, Strongly Type-3, Moderately Type-3, and Weakly Type-3) by analyzing the degree of code differences at line-level and token-level (*i.e., TokenDiff*). In fact, this classification of Type-3 is too coarse-grained and does not consider any program syntactic or semantic details. To achieve more precise results, we propose *TreeDiff* to describe the code differences at AST-level and divide Type-3 code clones into more fine-grained divisions, so as to obtain a clearer detection capability of different representation technologies. Through the reported results, we find that token-based and AST-based techniques perform better in detecting simple Type-3 clones, while graph-based (*i.e.,* CFG-based and PDG-based) techniques are only more effective for more complex Type-3 clones.

To answer the second question, we utilize 12 clone detection algorithms to mine as many clones as possible on real-world open source datasets. After manually verifying the clone pairs, we perform a fine-grained classification of the obtained true clones by analyzing *TokenDiff* and *TreeDiff*. After collecting all data, we observe that the *TokenDiff* and *TreeDiff* of more than 90% of Type-3 clones are less than 38% and 48%, respectively. Such results indicate that most Type-3 clones belong to simple Type-3. Therefore, we discard complicated code clone detection methods (*i.e.,* graph-based methods) and only choose lightweight techniques (*i.e.,* token-based and AST-based) to commence our combination experiments.

To answer the third question, we design a series of algorithm combination schemes to conduct recall measurement experiments. After analysis, we obtain an optimal algorithm combination, that is, a low-threshold token-based algorithm for filtering, and an AST-based algorithm for further clone verification. In order to verify the feasibility of this combination and support fast and accurate large-scale clone detection, we implement it as a tool, named TACC. Our evaluation experiments demonstrate that TACC performs the best in detecting all common clone types compared to five other state-of-the-art detection tools (*i.e.,* CCAligner [14], SourcererCC [15], NiCad [16], NIL [17], and Siamese [18]). In terms of scalability, TACC can complete 100MLOC clone detection task within 3 hours and 48 minutes.

In summary, our main contributions are as follows:

- We reproduce 12 clone detection techniques with different code representations and conduct an empirical study to figure out which code representation is more effective in detecting code clone.
- We propose *TreeDiff* and perform fine-grained division of real-world clone types. After manual analysis, we observe that most Type-3 clones are simple Type-3.
- We conduct different combination evaluations of different clone detection algorithms and find out a suitable combination that can achieve ideal performance.
- We implement TACC [1], an effective code clone detection tool that can scale to big code. Through our experimental results, we observe that TACC is superior to CCAligner [14], SourcererCC [15], NiCad [16], NIL [17], and Siamese [18].

The remainder of this paper is organized as follows. Section II presents an initial investigation of the clone detection capabilities of different representations, and Section III conducts a more fine-grained study. Section IV describes the distribution of clones in real world. Section V gives the process of finding the best combination of algorithms. Section VI presents clone detection details of our new approach. Section VII evaluates our tool and conducts large-scale experiments in open source projects. Section VIII discusses our work and section IX surveys retated work. Section X concludes the present paper.

II. Preliminary Study

According to the type of representation extracted from source code [19], [20], clone detection techniques can generally be categorized into text-based [16], [21], [22], token-based [6], [14], [15], [23], tree-based [24], [25], graph-based [8], [26], and metrics-based [27], [28].

Lightweight text-based and token-based methods are superior in speed, but they are generally considered not as accurate as tree-based and graph-based methods that take into account syntactic or semantic information. However, there is still a lack of a clear understanding of the ability of clone detection techniques based on different code representations. To fill such a research gap, we reproduce 12 clone detection algorithms based on three different representations and conduct a preliminary study to evaluate their effectiveness on a widely used benchmark dataset namely BigCloneBench [29].

A. Overview of Selected Algorithms

Our selection criteria are the publication year, conference/journal rank, and citations of papers. Table 1 shows the descriptions of our reproduced algorithms. In subsequent experiments, we use abbreviations to refer to algorithms for the convenience of description. Since text-based method NiCad

[1]TACC, https://github.com/TACC-Code/TACC

Table 1. Descriptions of our reproduced algorithms

Representation	Technique	Tool/1st author	Abbr.&Threshold
Token (no-syntactic)	Text-based	Nicad	t1=0.70
	Token-based	SourcererCC [15]	t2=0.70
		Lvmapper [30]	t3=0.70
		NIL [17]	t4=0.70
Tree	AST-based	Lazar [31]	a1=0.80
		Zhao [32]	a2=0.65
		Yang [33]	a3=0.80
		Deckard [25]	a4=0.85
Graph	CFGs-based	StoneDetector [34]	c1=0.80
		GroupDroid [35]	c2=0.95
		ATVHunter [36]	c3=0.85
	PDGs-based	CCgraph [26]	p1=0.90

and token-based methods SourcererCC, Lvmapper, and NIL are all technologies without syntactic information, we classify NiCad into the category of Token for convenience of experiments.

Learning-based clone detection has been a research hotspot in recent years [37], so we chose ASTNN [38] and RtvNN [39] for preliminary testing. We conduct an initial evaluation of both tools using two commonly used datasets, BigCloneBench [29] and Google Code Jam (GCJ) [40]. We found that the precision on GCJ did not meet expectations after the two tools were trained on BigCloneBench, and their training overhead tended to be time-consuming. In other words, learning-based clone detection techniques may not be competent for large-scale real-world clone detection tasks. In most cases, metrics-based clone detection techniques extract metrics of source code from ASTs or CFGs [1], [20]. Metrics-based technology actually overlaps with other algorithms in terms of code representation.

Token (non-syntactic). After preprocessing the source code, NiCad [16] obtains similarity by comparing the *longest common sub-sequence* (LCS) between texts. NIL [17] and Lvmapper [30] also use the LCS algorithm to calculate the similarity of token sequences, then use the Hunt-Szymanski algorithm [41] and heuristic algorithm for optimization, respectively. SourcererCC [15] uses the global token position map and computes the overlap similarity for code clone detection.

Tree. The approach proposed by Lazar *et al.* [31] is the same as Baxter *et al.* [42], comparing the same number of nodes in the subtree for similarity. Zhao *et al.* [32] proposed AST-CC which converts the tree structure of storing form into a linear list and groups the syntax tree information according to the number of sub-nodes to reduce the number of comparisons. Yang *et al.* [33] proposed an approach that replaces original nodes in the AST with a more abstract code representation through the defined node type, and then uses the Smith-Waterman algorithm [43] for similarity comparison. Deckard [25] extracts characteristic vectors from ASTs and uses the *locality sensitive hashing* (LSH) algorithm for the clustering of clones.

Graph. Amme *et al.* [34] developed a tool called StoneDetector, which analyzes CFGs to obtain dominance trees and compares dominance paths to detect clones. Marastoni *et al.* [35] proposed GroupDroid, which can measure similarity by extracting feature vectors from CFGs of methods. ATVHunter [36] is a CFG-based third-party library detection tool for Android applications, which can also be used for clone detection. CCgraph [26] is a PDG-based method, which uses an approximate graph matching algorithm to detect clones.

B. Experiments

Dataset. We conduct our evaluation on BigCloneBench [29] which is a Java-based and function-level dataset composed of more than 8,000,000 tagged clone pairs. Due to the fuzzy boundary division between Type-3 and Type-4, BigCloneBench makes a more fine-grained division of Type-3 and Type-4 by measuring the syntactic similarity: Very-Strongly Type-3 clones (VST3) with a similarity between [0.9, 1); Strongly Type-3 clones (ST3) with a similarity between [0.7, 0.9); Moderately Type-3 (MT3) with a similarity between [0.5, 0.7); Weakly Type-3/Type-4 clones (WT3/T4) with a similarity between [0, 0.5). The syntactic similarity is measured by calculating the minimum ratio of common lines or tokens between two code fragments after eliminating Type-1 formatting differences and Type-2 normalization. Common lines or tokens are obtained by a diff algorithm that takes into account the order of the lines or tokens. Specifically, we choose the same dataset used in ASTNN [38] which is a subset of BigCloneBench. This dataset consists of 15555 T1, 3663 T2, 1804 VST3, 9289 ST3, 10473 MT3 and 31835 WT3/T4. The total number of clone pairs is 72,619.

Experimental settings. We reproduce the above 12 clone detection algorithms in Table 1 using Python. The feature extraction for source code is implemented by leveraging javalang [44] and joern [45]. Javalang is a pure Python library, which can provide a lexer and parser for Java source code. Joern is a cross-language code analysis tool that is capable of generating CFGs and PDGs for Java source code.

The similarity thresholds of the above 12 algorithms are as consistent as possible with the configurations in articles [15]–[17], [26], [30], [36]. For algorithms whose optimal thresholds are not given in papers, we measure precision by manually validating clones. We sample 400 clone pairs for each algorithm separately and adjust the threshold in 5% steps to measure precision. We set the threshold to the corresponding value where the precision first approaches around 95%. The thresholds of each algorithm are set as t1 = 0.70, t2 = 0.70, t3 = 0.70, t4 = 0.70, a1 = 0.80, a2 = 0.65, a3 = 0.80, a4 = 0.85, c1 = 0.80, c2 = 0.95, c3 = 0.85 and p1 = 0.90. We set the minimum clone size to 6 lines which is often considered as the minimum granularity for functional clones [15], [46]. We conduct all the experiments on an AMAX computing server. It has two 2.1GHz 24-core CPUs and 384G memory.

Result. We measure recall by computing the union of clone pairs detected by algorithms that belong to the same

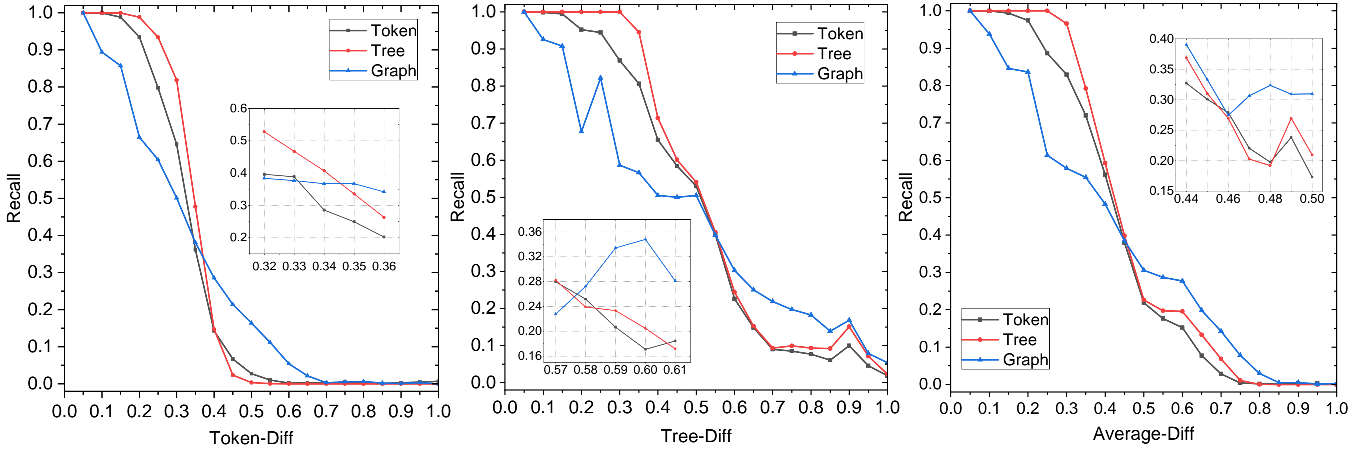

Fig. 1. Recall of different code representations under different degree of code differences

Table 2. Recall of algorithms based on different code representations

TYPE	Token	Tree	Graph
T1	**1**	**1**	**1**
T2	**1**	**1**	**1**
VST3	**1**	**1**	0.95
ST3	0.96	**0.98**	0.77
MT3	0.58	**0.65**	0.51
WT3/T4	0.04	0.06	**0.14**

representation (*i.e.,* Token, Tree, and Graph). The recall results of different representations are shown in Table 2.

All three representations achieve 100% recall on both Type-1 and Type-2 clone detection. On the recall of Very Strongly Type-3, Strongly Type-3, and Moderately Type-3, algorithms based on tree representation perform the best, with recall reaching 1, 0.98, and 0.65, respectively. For Weakly Type-3 clone detection, graph-based methods can achieve the best recall. This may be because in some simple Type-3 clones, the use of '+' for string concatenation often occurs. If the number of string concatenations is different, the generated CFGs will also be affected, resulting in false negatives. Instead, this situation is reflected in tokens and ASTs, with only minor changes. Another reason is that token-based and AST-based technologies can tolerate small modifications to statements. But if most of the modified statements are method calls, it will cause greater changes to the graph structure of CFGs than tokens and ASTs.

Overall, we observe that token and tree representations can achieve better performance than graph when detecting simple Type-3 (*i.e.,* VST3, ST3, and MT3). However, since the codes of WT3 clones become complicated, we need to consider more program semantics to detect them. At this time, graph representation is more effective than token and tree representations.

III. Fine-grained Study

Currently, clones are usually classified into four categories [1], [46], corresponding to textual similarity, lexical similarity, syntactic similarity, and semantic similarity in clone detection. In Section II, we find that token and tree representations outperform graph representation in detecting simple Type-3 clones. However, when investigating the ability of clone detection algorithms based on different code representations, BigCloneBench dataset suffers from two flaws: first, the granularity of BigCloneBench's classification of Type-3 and Type-4 is coarse, and cannot reflect the capabilities of different code representation algorithms in more detail; second, the syntactic similarity for clone classification in BigCloneBench only considers text lines and tokens, lacking structural and syntactic information.

Procedure. To enrich the information considered when classifying clones, we propose *TreeDiff* to describe the difference between the ASTs of two code fragments. Formally, given two ASTs $Tree_A$, $Tree_B$, *TreeDiff* can be measured as equation (1), where the $|Tree_A \cap Tree_B|$ is obtained by calculating the number of common nodes through tree matching algorithm [47]. Correspondingly, we also define *TokenDiff*, which is measured as equation (2). The calculation of syntactic similarity is the same as that in BigCloneBench. Then *AverageDiff* can be measured as equation (3), which is a compromise between *TokenDiff* and *TreeDiff*. *AverageDiff* allows us to consider both tokens and syntactic information when describing code differences.

$$TreeDiff = 1 - \frac{|Tree_A \cap Tree_B|}{|Tree_A \cup Tree_B|} \tag{1}$$

$$TokenDiff = 1 - SyntacticalSimilarity \tag{2}$$

$$AverageDiff = \frac{TokenDiff + TreeDiff}{2} \tag{3}$$

To more clearly study the capability of different code representations for clone detection, we divide the Type-3 and Type-4 clones of BigCloneBench into fine-grained partitions. Our methodology is as follows: first, we collect the *TokenDiff*, *TreeDiff*, and *AverageDiff* of Type-3 and Type-4 clone pairs

in BigCloneBench, respectively; second, we divide all clones into 20 subclasses with a granularity of 5%. Taking *TokenDiff* as an example, the *TokenDiff* value of the first subclass of clone is between 0-5% while the value of the 20th type of clone is between 95%-100%; third, we compute the recall of different code representations in detecting these 20 subclasses clones.

Result. Figure 1 shows the recall results of different code representations on three diffs (*i.e., TokenDiff*, *TreeDiff*, and *AverageDiff*). Through the results, we can observe that regardless of which diff is used to represent the degree of code difference, and regardless of which code representation is used to detect clones, the recall will decrease as the diff increases. It is reasonable because larger diffs indicate more differences between clone pairs, making them harder to detect. Moreover, we also find that when diff is at a small value, both token and tree representations can detect more clones than graph representation. However, when the diff exceeds a threshold, graph representation can detect more clones. This threshold is also different for different diffs. Specifically, the thresholds for *TokenDiff*, *TreeDiff*, and *AverageDiff* are 0.35, 0.58, and 0.47, respectively. This result is consistent with the results in Table 2, that is, token and tree representations perform better on simple clone detection, while graph representation performs better on complex clone detection. However, our results are more fine-grained, and specific thresholds can be obtained through Figure 1.

IV. Real-world Code Clone Distribution

In section II and III, our experimental datasets are all tagged clones in BigCloneBench, of which WT3/T4 accounts for as high as 43.8%. This may not represent the distribution of clones of a real scene. On the other hand, the verification process of clones is often accompanied by inevitable human subjective problems [48]. For some more complex clones, especially semantic clones, the judges' opinions are usually inconsistent. Therefore, finding out all clones in a subject system is an almost impossible task. An alternative is to use tools based on different representations to mine as many clones as possible and verify manually. In this way, it is as close to the true distribution of clones to the greatest extent possible.

Procedure. To obtain the real-world code clone distribution, we randomly pick 3,000 functions from IJaDataset [11], a large data repository containing 2.3 million java files. Then we execute our 12 code clone detection algorithms to scan clones from these 3,000 functions. After taking the union of all clones detected by the 12 algorithms, we obtain a total of 1,534 clone candidates. The clone detection results are manually verified by two judges. In case of disagreement, the third judge will verify and reach a consensus with the other two judges. Finally, we get 1,285 pairs of true clones. Of the clones that we verified as false positives, we considered some to be nonsense clones. An example of this is shown in List 1. Both Function 1 and Function 2 are used to bundle a few unit test

Listing 1. False Positive Example

```
1
2   //Function 1
3   public static Test suite () {
4           TestSuite suite = new TestSuite("Analyzers
                suite");
5           suite.addTest(TextAnalyzeSuite.suite());
6           suite.addTest( TabletextSuite.suite());
7           suite.addTestSuite(ElementFoundTest.class);
8           return suite;
9   }
10  //Function 2
11  public static Test suite () {
12          TestSuite suite = new TestSuite("Tests for
                pg.jbert.Utilities.mat2urban");
13          suite.addTestSuite(MyZoneTest.class);
14          suite.addTestSuite(MyZoneReaderTest.class);
15          suite.addTestSuite(MyPlansProcessorTest.class);
16          suite.addTestSuite(MyZoneToZoneRouterTest.class
                );
17          suite.addTestSuite(MyCdfMapperTest.class);
18          return suite;
19  }
```

cases and run them together. After code normalization, their token sequences are extremely similar. However, this is a very common way of writing project unit tests, so we don't think it's a meaningful clone. In addition to this, we also discard some of the candidate pairs for exception handling. This candidate pair is also a meaningless clone when the number of lines of code is small.

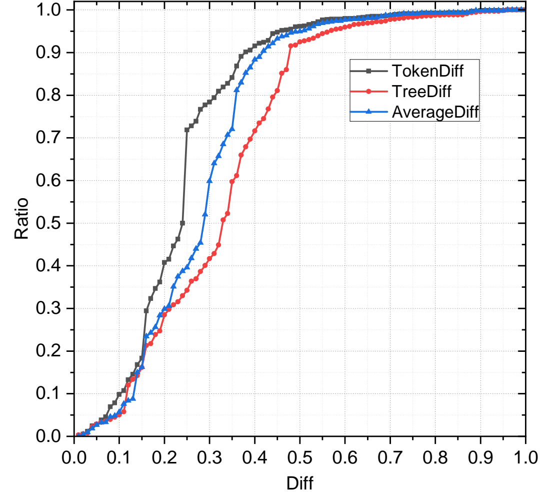

Fig. 2. Cumulative distribution function of *TokenDiff*, *TreeDiff*, and *AverageDiff* on real code clone pairs

Result. After eliminating Type-1 and Type-2 clone pairs (*i.e., TokenDiff* = 0), we calculate the *TokenDiff*, *TreeDiff*, and *AverageDiff* on the remaining 995 clone pairs. Figure 2 presents the *cumulative distribution function* (CDF) of *TokenDiff*, *TreeDiff*, and *AverageDiff*, respectively. In addition, we also record some important points in Table 3. First, we show the thresholds (*i.e.,* 0.35 for *TokenDiff*, 0.58 for *TreeDiff*, and 0.47 for *AverageDiff*) in Figure 1 and their

corresponding clone ratios. Second, we show the diff ranges when the cumulative ratio of clones reaches 90%, 95%, 99%, and 100% for the first time.

Table 3. Some important diff ranges and their corresponding clone ratios of *TokenDiff*, *TreeDiff*, and *AverageDiff*

TokenDiff		TreeDiff		AverageDiff	
diff range	clones ratio	diff range	clones ratio	diff range	clones ratio
(0, 0.35]	84.2%	(0, 0.58]	95.5%	(0, 0.47]	94.0%
(0, 0.38]	90.2%	(0, 0.48]	91.5%	(0, 0.42]	90.4%
(0, 0.46]	95.2%	(0, 0.57]	95.2%	(0, 0.50]	95.0%
(0, 0.72]	99.0%	(0, 0.87]	99.1%	(0, 0.73]	99.0%
(0, 0.96]	100%	(0, 0.96]	100%	(0, 0.96]	100%

Since both the lexical information and syntactic information of source codes are taken into account, the experimental results of *AverageDiff* are more representative. We find that the proportion of clone pairs is as high as 94.0% when the *AverageDiff* is between (0, 0.47]. When the *AverageDiff* of clone pairs is higher than 0.47, techniques based on graph representation can detect more clones. However, only 6% of clones have an *AverageDiff* higher than 0.47. This means that the clones that exist in the open-source community may be mostly in the range that token and tree representations can detect. In addition, when the *AverageDiff* is between (0, 0.96], the proportion of clone pairs reaches 100%, but only 1% of clone pairs with *AverageDiff* at (0.73, 0.96]. In other words, almost all clones are distributed in the interval of *AverageDiff* below 0.73. Therefore, to a certain extent, clone detection and evaluation works can ignore candidate clone pairs with an *AverageDiff* higher than 0.73 to speed up the efficiency.

V. AST&TOKEN WITH DIFFERENT THRESHOLD

We have found that when the *AverageDiff* is lower than 0.47, the clone detection technology based on token and tree representations is better than the one based on graph, and most of the clones that can be detected are distributed in this interval. In reality, the clone detection algorithms based on graph representation are time-consuming. In our experiments, the graph-based algorithms take about 4 hours to extract CFGs or PDGs for 60,000 files in the file preprocessing stage. Therefore, it is difficult to achieve ideal efficiency when using graph-based clone detection algorithms in large projects or systems. In this way, it seems that using only clone detection algorithms based on token and tree representations is sufficient. In this section, we focus on finding a suitable combination of token-based and tree-based algorithms to achieve scalable and effective code clone detection.

A. Combination of Token-based Algorithms

Procedure. In order to find the most reasonable combination of algorithms, we set 14 combinations for the four algorithms based on token. We evaluate the recall of each combination on the BigCloneBench dataset, and the similarity threshold of all algorithms adopts the optimal configuration in section II. The difference is that we know the possible distribution of code clones in real world, so we use *AverageDiff* as the basis for BigClonebench clone division, because *AverageDiff* can describe code differences at both token-level and syntax-level. The recall values of Type-1 and Type-2 clones on BigCloneBench for each algorithm are almost 100%, so we discard them in our experimental results. We only focus on clone pairs with *AverageDiff* between (0, 0.73], because 99% of clone pairs that can be detected in real world are distributed in this interval. In detail, we divide (0, 0.73] into five partitions with a granularity of 15%, and evaluate the recall of different combinations of algorithms on different *AverageDiff* partitions.

Result. Table 4 presents the recall results for combinations of token-based algorithms. Obviously, the recall of the combination of the three algorithms is stronger. Although this combination barely loses recall ability compared to all algorithms overall, the overhead problem cannot be ignored. In contrast, the combination of the two algorithms is more reasonable because the overhead is more acceptable and the recall only loses a little. Therefore, discarding the combination of the three algorithms, we mark the best recall in Table 4, and the same goes for Table 5. In the two combinations of algorithms, the combination of t2 and t3 works better, with the best recall at [0,15, 0.30), [0.30, 0.45), and [0.45, 0.60). In other words, the clones detected by the combination of t2 and t3 approximately contain clones recalled by all token-based algorithms. Therefore, from the perspective of recall and runtime overhead, we take a balanced approach, that is, choose a combination of t2 and t3 instead of using all token algorithms.

B. Combination of Tree-based Algorithms

Procedure. The combinations and settings of the AST-based algorithms are the same as token-based methods.

Result. Table 5 presents the recall results for combinations of AST-based algorithms. The experimental results are similar to the token-based algorithm combinations. In detail, the recall of the combination of the three algorithms is better. However, as aforementioned, the time cost factor cannot be ignored, so it is more reasonable to choose a combination of two algorithms. The recall of the combination of a2 and a4 is the highest on [0, 0.15), [0.30, 0.45), [0.45, 0.60), and [0.60, 0.73), and is almost the same as the overall recall. Therefore, we select a2 and a4 to replace all AST algorithms.

C. Combination of Token-based and Tree-based Algorithms

Based on the previous experiments, we have found a suitable combination of token-based and tree-based algorithms, respectively. However, compared to tokens, since extracting ASTs is more time-consuming and the number of nodes in the AST of the same code is more than the number of tokens, the overhead problem will be further magnified. To mitigate the issue, we propose to use a low-threshold token-based clone detection method for filtering, and then verify the clone pairs

Table 4. The recall of different combinations of token-based algorithms

AverageDiff	t1	t2	t3	t4	t12	t13	t14	t23	t24	t34	t123	t124	t134	t234	all
[0, 0.15)	0.995	0.986	0.737	0.813	**0.997**	0.996	0.995	0.991	0.987	0.899	0.998	0.997	0.996	0.991	0.998
[0.15, 0.30)	0.775	0.773	0.579	0.644	0.82	0.872	0.805	**0.882**	0.821	0.764	0.903	0.849	0.878	0.89	0.907
[0.30, 0.45)	0.291	0.41	0.191	0.187	0.441	0.395	0.362	**0.512**	0.489	0.282	0.529	0.507	0.424	0.543	0.555
[0.45, 0.60)	0.073	0.138	0.051	0.076	0.142	0.097	0.096	**0.16**	0.16	0.095	0.163	0.162	0.113	0.176	0.177
[0.60, 0.73)	0.007	0.028	0.008	0.016	0.029	0.014	0.018	0.035	**0.038**	0.02	0.035	0.039	0.022	0.042	0.042

Table 5. The recall of different combinations of tree-based algorithms

AverageDiff	a1	a2	a3	a4	a12	a13	a14	a23	a24	a34	a123	a124	a134	a234	all
[0, 0.15)	1.000	0.997	1.000	0.990	1.000	1.000	1.000	1.000	**1.000**	1.000	1.000	1.000	1.000	1.000	1.000
[0.15, 0.30)	0.875	0.912	0.632	0.624	0.982	0.929	**0.949**	0.937	0.936	0.761	0.986	0.989	0.963	0.947	0.990
[0.30, 0.45)	0.032	0.540	0.094	0.234	0.552	0.124	0.263	0.547	**0.580**	0.267	0.558	0.591	0.294	0.585	0.596
[0.45, 0.60)	0.000	0.185	0.054	0.087	0.185	0.054	0.087	0.189	**0.201**	0.109	0.189	0.201	0.109	0.204	0.204
[0.60, 0.73)	0.000	0.072	0.004	0.019	0.072	0.004	0.019	0.074	**0.081**	0.022	0.074	0.081	0.022	0.082	0.082

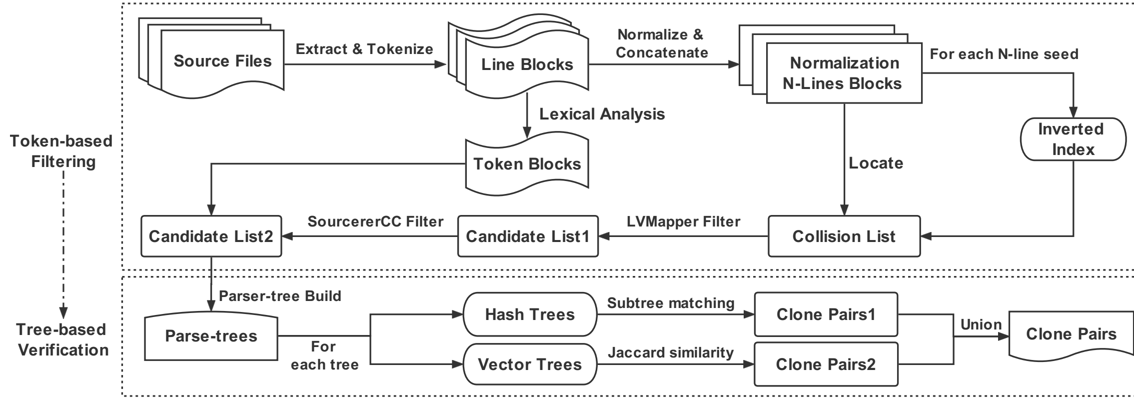

Fig. 3. Overview of TACC which consists of two phases: Token-based Filtering and Tree-based Verification

Table 6. The recall of combination between AST and Token with different thresholds

AverageDiff	T=0.4	T=0.45	T=0.5	T=0.55	AST	a2&a4
[0,0.15)	1.000	1.000	1.000	1.000	1.000	1.000
[0.15, 0.30)	0.936	0.936	0.936	0.935	0.990	0.936
[0.30, 0.45)	0.580	0.580	0.580	0.576	0.596	0.580
[0.45, 0.60)	0.201	0.201	0.201	0.198	0.204	0.201
[0.60, 0.73)	0.081	0.081	0.080	0.077	0.082	0.081

by using AST-based detection techniques after reducing most of clone candidates.

Procedure. We use low-threshold t2 and t3 as filtering, which aims to include as many clones as possible that can be detected by token-based algorithms, without erroneously filtering clones that can be detected by AST-based algorithms. After filtering by a token-based combination, we utilize a2 and a4 for clone verification and measure the recall results.

Result. Table 6 shows the recall results of a2 and a4 when t2 and t3 with different thresholds are used as the filter. It turns out that the recall of a2 and a4 does not decrease when the thresholds of filtering algorithms t2 and t3 are not higher than 0.45. This indicates that filtering by the low threshold t2 and t3 algorithms will not result in more false negative clones. Furthermore, when the thresholds of t2 and t3 are set to 0.50, the final recalls of a2 and a4 are only slightly lower than the original in the interval [0.60, 0.73). However, this difference is negligible. On the contrary, if the filter threshold is set to 0.50, more clone candidate pairs can be filtered out, thereby reducing the verification time of AST-based algorithms, which is worthwhile. Therefore, we finally choose a threshold of 0.5 for token-based algorithms.

VI. Approach

Based on the above experimental results, we obtain an ideal combination of algorithms and then develop it into a complete clone detection tool, called TACC (A **T**oken and **A**ST-based **C**ode **C**lone Detector). The overall process of our method is shown in Figure 3, which consists of two parts: Token-based Filtering and Tree-based Verification.

During the Token-based Filtering phase, we first extract all functions from the source files and concatenate all the adjacent N lines of code for each function. In this way, the inverted index can be created for locating. Then, we adopt the filter similarity calculation formula used in Lvmapper mainly based on the number of shared N-Lines as the first filter. After that, we compute the SourcererCC similarity of the remaining pairs based on their token sequences as the second filter.

During Tree-based Verification phase, two techniques based on AST are adopted in our work. The first technique is to use

a subtree matching algorithm based on hash trees to detect clones while the other is to analyze the tree vectors for clone detection. We use the above two methods to verify the token-filtered pairs separately, and finally combine their results to report our detected clone pairs. We elaborate the details of our approach in the following parts.

A. Token-based Filtering

1) Preprocess

In the preprocessing phase, we adopt TXL [49] to extract all functions from source files and then transform them into line sequences. After lexical analysis of these line sequences, we get token sequences for each function and sort them for subsequent SourcererCC filter utilization. Next, we normalize the line sequences including identifier replacement, comment removal, and blank line deletion. During the above normalization process, we also concatenate adjacent n lines of code to get N-lines. After code normalization, there is a high probability that a pair of non-clone functions have the same line of code, such as the variable declaration, 'return', 'if', 'try', etc. However, it is not common to have the same and consecutive n lines of code. So inverted index can be created from the generated N-Lines. We use a dictionary to store the inverted index whose key is the MurmurHash value [50] of N-Lines, which has a low collision probability, and value is the function index containing the corresponding N-Lines.

2) Location

In the location phase, we use the inverted index generated in the preprocessing phase to perform the locating operation which can significantly reduce the number of clone pairs that need to be detected at a small cost. For a given function, we traverse its N-Lines hash value and use it as the key to collect its corresponding function index in the inverted index.

3) Filtering

In the filter phase, we adopt two methods to remove these pairs with low similarity. The first method is the same as the filtering stage in LVmapper. It calculates the similarity through the shared N-lines of pairs, which is measured as equation (4):

$$SR(A,B)=\frac{S}{T}=\frac{S}{L-N+1}=\frac{S}{max(|A|,|B|)-N+1} \quad (4)$$

Except for S, the definitions of other parameters are consistent with LVmapper. S represents the approximate number of common N-lines of functions A and B, which can be calculated by the following steps: given functions A and B, traverse the N-lines hash sequences of A and get their corresponding value in the inverted index, and finally determine whether the index of function B is in it. By computing S in this way, the time complexity can be reduced from $O(|\textit{N-lines(A)}|\times|\textit{N-lines(B)}|)$ to $O(|\textit{N-lines(A)}|)$. Although computing common N-lines in this way ignores repeated N-lines in function B, this happens rarely and is tolerable during the preliminary filtering stage. We remove pairs whose SR score is less than the first filtration threshold θ_1.

After fast N-lines filtering, we use another fine-grained token-based algorithm for further filtering. We use the similarity determination formula in the verification phase of SourcererCC as our further filtering basis, which is measured as equation (5):

$$|B_x \cap B_y| \geq \lceil \theta_2 \cdot max(|B_x|,|B_y|) \rceil \quad (5)$$

Note that θ_2 is the threshold of SourcererCC filter and $|B_x|, |B_y|$ are the length of two token sequences, respectively. Since x and y are already sorted in the preprocessing phase, SourcererCC filter's time complexity can be reduced from O(m×n) to O(m+n). Finally, those pairs whose overlap tokens are less than $\lceil \theta_2 \cdot max(|B_x|,|B_y|) \rceil$ are filtered out.

At this stage, the use of token-based algorithms is not limited to filtering candidate pairs, but can also be used for clone verification. If the SR is higher than the threshold θ_3 or the overlap similarity is higher than the threshold θ_4, the candidate pair can be directly regarded as a clone pair without further verification of the AST algorithms. In this way, the detection efficiency is accelerated by reducing unnecessary verification.

B. Tree-based Verification

1) Tree Information Generation

At this stage, we obtain the hash tree and characteristic vectors of the function. The idea of building the hash tree comes from a2 [32]. Comparing syntax tree nodes directly is complex and inefficient. So we can build a special hash syntax tree for fast execution of subtree matching algorithms. The specific method is as follows:

- First, we conduct static analysis to extract the AST of a given function;
- Second, we traverse the entire tree in post-order traversal. For leaf nodes, we directly call the hash function to calculate and obtain the hash value. For non-leaf nodes, its hash value is the sum of all its child nodes and its own hash value;
- Third, we count the hash values of tree nodes, and store the tree hash information into a dictionary whose key is the tree node hash and value is the number of key occurrences in the hash tree.

After the above processing, since there are less than 100 types of tree nodes, the probability of hash collision is very low. Therefore, we can directly judge whether the subtree rooted at the node is the same by comparing whether the hash values of the two nodes are the same. In this way, the time and space overhead of the algorithm can be greatly reduced on the premise of losing grammatical information.

The extraction of characteristic vectors is similar to that done by Deckard. Characteristic vectors can capture the structural information of the tree and store it in the form of a vector. Vector generation is similar to hash tree, and it also traverses each node of the tree in post-order, whose characteristic vectors are generated by summing up its children and own vectors. The dimension of the vector $n = |\zeta|^{2q-1}$, where q represents the atomic pattern level, and ζ represents the label set size. The node type is automatically generated by a static analysis tool (*i.e.,* JavaParser) during tree parsing.

2) Candidates Similarity Calculation

After the two stages of token-based filtering and tree information generation, we only need to call a2 and a4 algorithms to verify the remaining pairs.

For a2 algorithm, given a pair (A, B), the algorithm first selects the function with the smaller *hash node count dictionary* (HNCD) size in the pair, and then repeatedly judges whether A's HNCD key is in B's HNCD. If it exists, it will add the smaller value of the corresponding value of the key in the two dictionaries to the common nodes of A and B. Finally, we calculate the similarity between the two by the following equation (6):

$$AT(A,B) = \frac{common_nodes}{max(|HNCD(A)|, |HNCD(B)|)} \quad (6)$$

For a4 algorithm, given a pair (A, B), we do not verify the clone by computing the norm of the vector as in [25] because it is unfair to take a fixed value for trees of different sizes. The average distance between pairs with fewer tree nodes will be smaller than the average distance between pairs with more tree nodes. After trying indicators such as cosine similarity, Pearson correlation, and Jaccard similarity coefficient, we finally choose Jaccard similarity coefficient to measure the similarity between two vectors since it can perform the best in our experiments. The specific calculation formula is as equation (7):

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cup B|} \quad (7)$$

Finally, if $AT(A,B) \geq \delta_1$ or $J(A,B) \geq \delta_2$, we treat the candidate pair as a clone and add it to the clone pairs.

VII. Evaluation

In this section, we thoroughly evaluate the detection performance of TACC. Specifically, we first describe the parameter settings of TACC. We then evaluate TACC on the BigCloneBench dataset. Finally, we check the scalability of TACC and conduct a large-scale study to demonstrate the feasibility of TACC on big code analisys. In addition, we also compare the above experimental results with five other state-of-the-art clone detectors (*i.e.,* CCAligner [14], SourcererCC [15], Siamese [18], NIL [17], and NiCad [16]). The settings of these detectors are all derived from their published papers [14]–[18].

A. Parameter Settings

TACC requires seven parameters including N for N-lines, filter thresholds θ_1 and θ_2, token-based verification thresholds θ_3 and θ_4, and tree-based verification thresholds δ_1 and δ_2. First of all, the selection of the N-lines parameter N will greatly affect the final detection result and execution time. We measure recall and execution time for different N (*i.e.,* 1-5) on a subset of BigCloneBench, as in Section II. After analyzing the experimental results, we find that as the value of N goes from 1 to 3, the execution time decreases significantly, but the corresponding recall also decreases gradually. Therefore, considering the balance between recall and execution efficiency, we take 3 for N as a suitable solution.

Overall, we set θ_1 to 0.1 as LVmapper does. The other parameters are consistent with the experimental settings in Sections II and V, that is, $\theta_2 = 0.5$, $\theta_3 = 0.7$, $\theta_4 = 0.7$, $\delta_1 = 0.65$, and $\delta_2 = 0.85$. In other words, in the Token-based Filtering stage, we first select LVmapper with a threshold of 0.1 as the first filter, and then select SourcererCC with a threshold of 0.5 as the second filter. Meanwhile, pairs with a similarity greater than 0.7 are directly considered as clone pairs. In the Tree-based Verification stage, we use both a2 and a4 to analyze the filtered candidates. When the similarity of a2 is above 0.65 or the similarity of a4 is above 0.85, we will recognize that it is a clone pair.

B. Recall and Precision

We evaluate the recall and precision of TACC on the subset of BigCloneBench, whose components are shown in Section II. In addition, we also divide the dataset by $AverageDiff$ and perform recall measurement as in the above experiments.

Table 7. Detection performance of TACC, CCAligner, SourcererCC, Siamese, NIL, and NiCad on general BigCloneBench and $AverageDiff$-based BigCloneBench. Note that CCA, Sou, Sia, and Nic denote CCAligner, SourcererCC, Siamese, and NiCad, respectively.

	Tool	TACC	CCA	Sou	Sia	NIL	NiC
Recall	Type-1	**100**	100	94	100	99	98
	Type-2	**100**	100	78	96	97	84
	VST3	**100**	99	54	85	88	97
	ST3	**94**	65	12	59	66	52
	MT3	**55**	14	1	14	19	2
	WT3/T4	**2**	1	0	1	1	0
	[0, 0.15)	**96**	77	28	64	72	51
	[0.15, 0.30)	**84**	49	4	47	57	42
	[0.30, 0.45)	**52**	10	1	12	15	8
	[0.45, 0.60)	**12**	5	1	5	5	2
	[0.60, 0.73)	**4**	2	0	1	1	1
Precision		95	61	**100**	98	86	99

1) Recall

As shown in Table 7, TACC performs better than other tools on all clone categories. On the recall of simpler clones like T1, T2, and VST3, TACC performs flawlessly. For recall in detecting ST3 clones, TACC (93%) leads the second NIL (66%) by nearly 28%. For MT3 clones, TACC shows an unparalleled advantage over the recall of other comparative tools, leading the other tools by more than 36%. In Section V, we divide T3/T4 clones with $AverageDiff$ between [0, 0.73) by 15% steps and measure the recall of each tool in each interval. The recall results are shown in Table 7. After removing T1 and T2 clones, the difference in recall for each tool is more pronounced. TACC achieves the best recall in all intervals compared to other tools. According to the distribution of real clones, the proportion of clones in the interval of $AverageDiff$ between [0, 45] exceeds 90%, and the recall performance of TACC is also far superior to other tools.

2) Precision

As for the measurement of precision, similar to other previous works [14], [51], we randomly sample 400 clone pairs from clone reports in each tool and conduct manual analysis to validate them. Each clone pair is checked independently by two experts. If there is a conflict, a final decision will be made after a discussion with another expert. The principle rule for judging is based on the overall similarity between the two clone fragments and on whether they perform similar functionality. After manual inspection, we find that TACC has a precision of 95%, which is slightly lower than SourcererCC but acceptable performance.

Table 8. Runtime overhead of TACC, CCAligner, SourcererCC, Siamese, NIL, and NiCad when processing different sizes of code

Tools	TACC	CCAligner	SourcererCC	NiCad	Siamese	NIL
1K	1s	1s	3s	1s	4s	1s
10K	1s	2s	5s	3s	14s	1s
100K	3s	6s	7s	36s	45s	3s
1M	12s	11m52s	37s	6m13s	40m1s	11s
10M	3m4s	29m48s	12m21s	2h10m	14h11m	1m3s
100M	3h48m	-	12h27m	-	6d16h46m	29m11s

C. Scalability

In this part, we focus on evaluating the scalability of TACC and our comparative tools. Specifically, we construct datasets of different orders of magnitude as input to tools and record their runtime. The size of the dataset is measured by the number of *lines of code* (LOC), from 1KLOC to 100MLOC. We limit the memory of the machine to 32G for scalability experiments. The execution times of tools under different magnitudes of input are shown in Table VII-B2. For inputs from 1KLOC to 1MLOC, TACC exhibits comparable execution efficiency to NIL. On larger inputs like 10MLOC and 100MLOC, the execution time of TACC only lags behind NIL. This result can be predicted that compared with pure Token-based technology, TACC's AST-based verification stage is more time-consuming. But it only takes about 3 hours and 48 minutes for TACC to analyze 100MLOC, indicating that TACC is suitable for scanning large-scale code clones.

D. Large Scale Clone Detection

In this part, we conduct an in-depth empirical study of function clones across 4,500 open-source projects by using TACC. In the selection of open-source datasets, we follow the indicators of open-source project criticality scores [52]. Criticality score can be used to describe the influence and importance of an open source project. According to the score ranking table of nearly 100K open source projects provided in the Github repository of criticality score, we select the top-ranked java projects. Specifically, we download 4,500 projects and apply TACC to detect code clones from these projects. This dataset contains about 1.4M files and 150MLOC. We detect about 7.55 billion clone pairs in these 4,500 projects and analyze these clones.

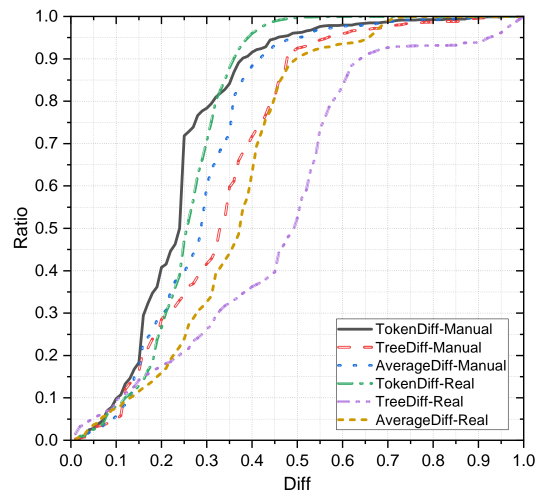

Fig. 4. Cumulative distribution function of $TokenDiff$, $TreeDiff$, and $AverageDiff$ on our manually verified clone pairs and detected open-source clone pairs

Table 9. Some important diff ranges and their corresponding clone ratios of $TokenDiff$, $TreeDiff$, and $AverageDiff$

TokenDiff		TreeDiff		AverageDiff	
diff range	clones ratio	diff range	clones ratio	diff range	clones ratio
(0, 0.35]	87.8%	(0, 0.58]	80.3%	(0, 0.47]	86.3%
(0, 0.36]	90.1%	(0, 0.65]	90.1%	(0, 0.50]	90.2%

We randomly sample 10,000,000 clone pairs from the detected clones. First, we classify clones in general types by measuring $TokenDiff$, and the results show that the proportions of T1, T2, and T3 are 9.60%, 11.28%, and 79.12%, respectively. As done in Section IV, we collect three diffs of these clones and eliminate T1 and T2 clones. Figure 4 presents the CDF of $TokenDiff$, $TreeDiff$, and $AverageDiff$, respectively. We also show some important points in Table 9 as before. As shown in Figure 4, the distribution of clones based on $TokenDiff$ is roughly consistent with that in Section IV. while the distribution of clones based on $TreeDiff$ is somewhat different. However, the proportion of clones with $AverageDiff$ between [0, 0.47] only decreases by nearly 8%. In fact, the distribution of clones in such a large-scale dataset is closer to the true distribution. According to the $AverageDiff$-based distribution, only nearly 14% of clones fall into the range where graph representation is more efficient. Considering the proportion of T3 clones, this proportion is only about 11% (79.12% × 14%). In other words, although there are differences from the results in Section IV, tree and token representations can still handle most clones, which also confirms the rationality of our TACC from the side.

VIII. Discussion

Distribution of clones. Our results on the distribution of real-world clones and large-scale projects are not entirely accurate. Mining all clones from the codebase is a nearly

impossible task. Even if the verification of candidate clones is completely manual, it is not necessarily reliable. Since judging the most complex clones is accompanied by subjective factors, disagreements inevitably arise. Therefore, we chose to mine as many clones as possible by using more tools. However, there are still quite a few semantic clones that cannot be detected by existing tools. Therefore, the distribution of clones obtained by manual analysis or mining of large-scale data sets cannot be regarded as an accurate distribution of clones. Nonetheless, our work can still reflect the distribution of real-world clones to a certain extent and serve as a reference for researchers. In the future, we consider investigating more real codebases to increase the effectiveness of the distribution of code clones.

IX. Related Work

A. Clone Detector Evaluation

As the field of code cloning continues to advance, many new techniques and tools have emerged. It is necessary to evaluate these tools to provide reference for developers or researchers to select appropriate tools. The main metrics for evaluating clone detection tools can be divided into three: precision, recall, scalability.

Dataset. The comparison of clone detectors requires a benchmark, which is difficult to be established at scale, especially since measuring the recall of a tool requires prior knowledge of all clones in the test sample. Manual verification of large numbers of candidate clones is expensive, and the process is always accompanied by intractable human subjectivity issues [48]. The first real attempt to establish a benchmark was made by Bellon et al. [9], but this job took 77 hours to validate just 2% of 325,935 candidate clone pairs. Nevertheless, the validity of the Bellon's benchmark is debatable, with the research [53] showing that Bellon's judgments about the types of clones in the benchmark are inconsistent with other judges. Roy et al. [54], [55] developed the Mutation and Injection Framework to generate synthetic clones. This framework can generate clones by mimicking the code-paste-copy-and-modify behavior of developers without the need for cumbersome manual verification of clone pairs. It is automated and can solve the difficulty of creating benchmarks to a certain extent, but its limitation is that it does not fully represent real clones, nor can it synthesize clones of Type4. GCJ and OJClone [56] are collected from submissions on programming contest websites. Different answers to the same question in datasets can be regarded as semantic clones for testing. But GCJ and OJClone are not collected from a real development environment, which threatens validity. Another popular clone benchmark at the moment is BigCloneBench [29], which is mined from real-world Java repositories. BigCloneBench provides a big-data, diverse and comprehensive benchmark for modern large-scale clone detection tools.

B. Code Clone Detection

According to Rattan et al. [20], existing clone detection tools can be classified into text-based, token-based, tree-based, metric-based, and graph-based. Text-based clone detectors [16], [22], [57], [58] treat code as text or strings with little transformation, and then perform similarity comparisons. Detection tools [6], [15], [59] that utilize lexical methods use a parser or lexer to transform the source code into a sequence of tokens before cloning matching. Tree-based clone detectors [42], [60] parse programs into parse trees or abstract syntax trees, and then uses tree-matching algorithms to search for similar subtrees. This code representation is typically able to detect more complex clones due to preserving the syntax information of the code. There are also tools that are graph-based [8], [61], [62], which extract CFGs or PDGs from programs as code representation and can detect Type-4 clones effectively. Tools [27], [51] analyze codes and extract metrics as code representations and then compare the metrics values for clone verification. In addition, there are clone detection approaches that use hybrid techniques [63] or learning-based methods [64]–[68], which tend to achieve better detection results than common methods [69].

X. Conclusion

In this paper, we reproduce 12 text-based, token-based, AST-based, and graph-based algorithms, and conduct a detailed investigation of the ability of different code representations. Then we attempt to analyze the distribution of clones in the real world and find that most clones are simple in nature. Based on the distribution of clones, we abandon heavy graph-based clone detection techniques and find an optimal combination of algorithms to implement as a tool, TACC. Through our experimental results, we observe that TACC is superior to CCAligner [14], SourcererCC [15], NiCad [16], NIL [17], and Siamese [18] in terms of effectiveness. As for scalability, it only takes about 3 hours and 48 minutes to complete the whole analysis on 100MLOC. Such a result indicates that TACC can detect large-scale code clones. In future work, we are planning on optimizing our tool to support more programming languages.

Acknowledgements

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported in part by the National Nature Science Foundation, China (Grant No. 61972373). The research of Dr. Xue is supported by CAS Pioneer Hundred Talents Program. This research of Dr. Liu is partially supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, and the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[2] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[4] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, 2015, pp. 201–210.

[5] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering-Volume 1*, 1993, pp. 171–183.

[6] 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.

[7] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proceedings of the 13rd Working Conference on Reverse Engineering (WCRE'06*. IEEE, 2006, pp. 253–262.

[8] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, 2017, pp. 100–109.

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[10] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.

[11] 2022. IJaDataset. https://sites.google.com/site/asegsecold/projects/seclone/.

[12] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, 2011, pp. 283–292.

[13] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'2012)*, 2012, pp. 1289–1292.

[14] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1066–1077.

[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'2016)*, 2016, pp. 1157–1168.

[16] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th iEEE International Conference on Program Comprehension (ICPC'2008)*. IEEE, 2008, pp. 172–181.

[17] T. Nakagawa, Y. Higo, and S. Kusumoto, "Nil: large-scale detection of large-variance clones," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ASE'21)*, 2021, pp. 830–841.

[18] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2236–2284, 2019.

[19] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[20] 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.

[21] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *Proceedings of the 7th Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 2005, pp. 8–pp.

[22] 2022. Tool Simian. http://www.harukizaemon.com/simian/index.html.

[23] J. Svajlenko and C. K. Roy, "Cloneworks: A fast and flexible large-scale near-miss clone detection tool." in *Proceedings of the 39th International Conference on Software Engineering (ICSE'2017)*, 2017, pp. 177–179.

[24] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," *ACM Sigcse Bulletin*, vol. 31, no. 1, pp. 266–270, 1999.

[25] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[26] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: a pdg-based code clone detector with approximate graph matching," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 2020, pp. 931–942.

[27] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *International Conference on Software Maintenance*, vol. 96, 1996, p. 244.

[28] Z. O. Li and J. Sun, "A metric space based software clone detection approach," in *Proceedings of the 2nd IEEE International Conference on Information Management and Engineering (ICIME'10)*. IEEE, 2010, pp. 393–397.

[29] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 2015, pp. 131–140.

[30] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy, "Lvmapper: A large-variance clone detector using sequencing alignment approach," *IEEE Access*, vol. 8, pp. 27 986–27 997, 2020.

[31] F.-M. Lazar and O. Banias, "Clone detection algorithm based on the abstract syntax tree approach," in *Proceedings of the 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI'14)*. IEEE, 2014, pp. 73–78.

[32] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An ast-based code plagiarism detection algorithm," in *Proceedings of the 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA'15)*. IEEE, 2015, pp. 178–182.

[33] Y. Yang, Z. Ren, X. Chen, and H. Jiang, "Structural function based code clone detection using a new hybrid technique," in *Proceedings of the 42nd Annual Computer Software and Applications Conference (COMPSAC'18)*, vol. 1. IEEE, 2018, pp. 286–291.

[34] W. Amme, T. S. Heinze, and A. Schäfer, "You look so different: Finding structural clones and subclones in java source code," in *Proceedings of the 28th IEEE International Conference on Software Maintenance and Evolution (ICSME'21)*. IEEE, 2021, pp. 70–80.

[35] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, "Groupdroid: Automatically grouping mobile malware by extracting code similarities," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop (SSPREW'17)*, 2017, pp. 1–12.

[36] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE21)*, 2021, pp. 1695–1707.

[37] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, 2022.

[38] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 2019, pp. 783–794.

[39] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 87–98.

[40] 2022. Google Code Jam. https://code.google.com/codejam/past-contests.

[41] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[42] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *International Conference on Software Maintenance*. IEEE, 1998, pp. 368–377.

[43] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[44] 2022. Javalang. https://github.com/c2nes/javalang.

[45] 2022. joern. https://joern.io.

[46] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[47] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[48] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. IEEE, 2018, pp. 26–37.

[49] 2022. TXL. http://www.txl.ca.

[50] 2022. Murmurhash. https://github.com/aappleby/smhasher.

[51] V. Saini, F. Farmahinifarahani, H. Sajnani, and C. Lopes, "Oreo: Scaling clone detection beyond near-miss clones," in *Code Clone Analysis*. Springer, 2021, pp. 63–74.

[52] 2022. Criticality score. https://github.com/ossf/criticality_score.

[53] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (ASE'15)*, 2015, pp. 1–10.

[54] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 476–480.

[55] J. Svajlenko and C. K. Roy, "The mutation and injection framework: evaluating clone detection tools with mutation analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1060–1087, 2019.

[56] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, 2016.

[57] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. IEEE, 2009, pp. 219–228.

[58] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 2005, pp. 8–pp.

[59] 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.

[60] Y. Wu, S. Feng, D. Zou, and H. Jin, "Detecting semantic code clones by building ast-based markov chains model," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*, 2022, pp. 1–13.

[61] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. IEEE, 2001, pp. 301–309.

[62] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Static Analysis Symposium (SAS'01)*. Springer, 2001, pp. 40–56.

[63] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, "Licca: A tool for cross-language clone detection," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. IEEE, 2018, pp. 512–516.

[64] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 2017, pp. 249–260.

[65] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*, 2020, pp. 516–527.

[66] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19)*. IEEE, 2019, pp. 70–80.

[67] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 821–833.

[68] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan, and H. Jin, "Treecen: Building tree graph for scalable semantic code clone detection," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*, 2022, pp. 1–12.

[69] Roopam and G. Singh, "To enhance the code clone detection algorithm by using hybrid approach for detection of code clones," in *Proceedings of the 1st International Conference on Intelligent Computing and Control Systems (ICICCS'17)*, 2017, pp. 192–198.