TreeCen: Building Tree Graph for Scalable Semantic Code Clone Detection

Yutao Hu[*] [†]
Huazhong University of Science and Technology
China
yutaohu@hust.edu.cn

Deqing Zou[*] [†]
Huazhong University of Science and Technology
China
deqingzou@hust.edu.cn

Junru Peng
Xidian University
China
pengjunru0716@outlook.com

Yueming Wu[‡]
Nanyang Technological University
Singapore
wuyueming21@gmail.com

Junjie Shan
KTH Royal Institute of Technology
Sweden
jshan@kth.se

Hai Jin[†] [§]
Huazhong University of Science and Technology
China
hjin@hust.edu.cn

ABSTRACT

Code clone detection is an important research problem that has attracted wide attention in software engineering. Many methods have been proposed for detecting code clone, among which text-based and token-based approaches are scalable but lack consideration of code semantics, thus resulting in the inability to detect semantic code clones. Methods based on intermediate representations of codes can solve the problem of semantic code clone detection. However, graph-based methods are not practicable due to code compilation, and existing tree-based approaches are limited by the scale of trees for scalable code clone detection.

In this paper, we propose *TreeCen*, a scalable tree-based code clone detector, which satisfies scalability while detecting semantic clones effectively. Given the source code of a method, we first extract its *abstract syntax tree* (AST) based on static analysis and transform it into a simple graph representation (*i.e., tree graph*) according to the node type, rather than using traditional heavyweight tree matching. We then treat the *tree graph* as a social network and adopt centrality analysis on each node to maintain the tree details. By this, the original complex tree can be converted into a 72-dimensional vector while containing comprehensive structural information of the AST. Finally, these vectors are fed into a machine learning model to train a detector and use it to find code clones. We conduct comparative evaluations on effectiveness and scalability. The experimental results show that *TreeCen* maintains the best performance of the other six state-of-the-art methods (*i.e., SourcererCC*, *RtvNN*, *DeepSim*, *SCDetector*, *Deckard*, and *ASTNN*) with F1 scores of 0.99 and 0.95 on BigCloneBench and Google Code Jam datasets, respectively. In terms of scalability, *TreeCen* is about 79 times faster than the other state-of-the-art tree-based semantic code clone detector (*ASTNN*), about 13 times faster than the fastest graph-based approach (*SCDetector*), and even about 22 times faster than the one-time trained token-based detector (*RtvNN*).

[*]Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
[‡]Yueming Wu is the corresponding author
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556927

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

ACM Reference Format:
Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. 2022. TreeCen: Building Tree Graph for Scalable Semantic Code Clone Detection. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556927

1 INTRODUCTION

Code clone detection is an important research domain in software engineering. A cloned code is a code fragment that is identical or similar to another snippet. Generally, clones are classified into four types (*i.e.,* Type 1-4 clones) depending on the degree of code similarity. The first three types of clones (*i.e.,* Type 1-3 clones) are code pairs with syntactic similarity, generally introduced by developers copying code. These clone types are easily detected as the code pair with high similarity. The last type of clone (*i.e.,* Type-4 clone) is semantically similar to other code fragments, which may result from the developer referring to code with similar functionality while programming. Semantic clones have an implicit similarity. Therefore, clone detection for them is a complex research problem to solve.

Many methods have been proposed to detect code clones, categorized into two dominant groups according to the detection effect: scalable clone detection and semantic clone detection. For scalable clone detection, the main methods are text-based and token-based [49], [32], [21], [28], [27], [43], which directly transform code fragments into text or token sequences and then perform similarity comparison. Although these methods require a short execution time for clone detection, they cannot cope with the detection of semantic clones due to the lack of consideration of code semantic information. In order to detect semantic clones effectively, some detection methods based on the intermediate representation of the code have been proposed [55], [54], [56], [50]. Related works have proven that graph-based and tree-based detection methods can detect semantic clones, but the existing methods are not applicable for scalable detection. For graph-based approaches, both *program dependency graphs* (PDGs) and *control flow graphs* (CFGs) have complex structures, leading to a significant runtime overhead for graph similarity comparison. Moreover, generating accurate graph representations requires code compilation, resulting in limited detection of some code fragments (*e.g.,* individual functions, code segments). To make clone detection more scalable, some researchers have proposed tree-based clone detection schemes that effectively avoid the difficulties of compiling code. However, a major limitation of the existing tree-based methods is that the structure of tree representation is also highly complex, resulting in a high time overhead when comparing similarities. For example, there are 40 nodes in the AST generated for a mere 4-line code fragment, as shown in Figure 3.

In this paper, we intend to mitigate the high runtime overhead of tree-based approaches and propose a novel scalable tree-based semantic code clone detector. Specifically, we face one main challenge:

- *How to transform a complex tree into a simple representation form while preserving the tree structure information?*

To address this challenge, we first transform the AST of a method by node type into a simpler graph representation, namely *tree graph*. After obtaining the *tree graph*, we regard it as a social network and apply social network-based centrality analysis to dig out the centrality of each node. Centrality analysis was first proposed in social network analysis, whose original purpose was to measure the importance of a person within a network. Empirical studies [13, 14, 18] have validated that centrality analysis can retain network structural properties and has been used in many different areas [16, 25, 34]. After performing centrality analysis on the *tree graph*, we can obtain a fixed-length vector. In this way, the complex AST is transformed into a 72-dimensional vector while maintaining the tree details. By analyzing succinct vectors, we can achieve scalable semantic code clone detection.

We implement a simple yet effective system, *TreeCen*, and evaluate its detection accuracy and scalability on two popular benchmark datasets, termed *Google Code Jam* (GCJ) [4] and *BigCloneBench* (BCB) [1]. We also compare *TreeCen* with six state-of-the-art code clone detection methods including two token-based approaches (*i.e., SourcererCC* [45] and *RtvNN* [53]), two graph-based tools (*i.e., DeepSim* [56] and *SCDetector* [54]), and two tree-based detectors (*i.e., Deckard* [26] and *ASTNN* [55]). The experimental results show that *TreeCen* outperforms the above comparative systems. In detail, the most effective comparison tool on the BCB dataset is *DeepSim*, reaching an F1 score of 0.98, and the best one on the GCJ dataset is *ASTNN*, with an F1 score of 0.91. In fact, *TreeCen* improves the F1 scores to 0.99 and 0.95 on two benchmark datasets, respectively. *TreeCen* also achieves excellent performance in terms of time overhead. Specifically, *TreeCen* takes 239.2s to detect one million code pairs, which is at least 79 times faster than another tree-based semantic code clone detector (*i.e., ASTNN*), about 13 times faster than the fastest graph-based approach (*i.e., SCDetector*), and even about 22 times faster than the one-time trained token-based detector (*i.e., RtvNN*). Such results indicate that *TreeCen* is suitable for large-scale code clone detection.

Contributions. In summary, our paper makes the following contributions:

- We propose a novel approach to simplify an AST into a simple graph representation according to the node types and adopt centrality analysis to covert it into a fixed-length vector. The transformation helps to avoid high-cost tree comparison while preserving comprehensive structural information of the AST.
- We design and implement a simple yet effective system, *TreeCen* [1], which is a tree-based detector with both semantic clone detection capability and scalability.
- We conduct extensive evaluations in terms of effectiveness and scalability. The experimental results show that *TreeCen* is superior to the other six state-of-the-art methods (*i.e., SourcererCC* [45], *RtvNN* [53], *DeepSim* [56], *SCDetector* [54], *Deckard* [26], and *ASTNN* [55]).

Paper Outline. Section 2 presents the preliminary study on several centrality measures of code clones. Section 3 introduces the related definitions. Section 4 describes the system architecture of *TreeCen*. Section 5 reports the experimental results. Section 6 discusses the future work. Section 7 presents the related work. Section 8 concludes the present paper.

2 PRELIMINARY STUDY

Centrality measures have been widely used in social network analysis, which is adopted to evaluate the importance of nodes in a network. Many research works in other fields have also been proposed and demonstrated that centrality measures could effectively reveal the structure of a graph or network [25], [34], [16].

For code analysis, the AST is a common and essential code intermediate representation that reflects syntactic and semantic information. Since ASTs can be obtained without compilation compared to PDGs or CFGs, AST-based approaches are more scalable for code clone detection tasks. For code cloning, similar codes should have a more similar AST structure, while codes with different functionalities should have a more distinctive AST. However, there has been no work to demonstrate whether centrality analysis can effectively reveal the structural features of trees, as the centrality measures can only be analyzed for graphs and cannot be directly adapted to tree analysis. To this end, we perform a preliminary study to investigate the below questions:

- *Can AST be converted into a simpler graph representation?*
- *Can the centrality measures effectively reflect the structural information of the ASTs, thus showing high similarity for clone code pairs and low similarity for code pairs with different functionality?*

To answer the proposed questions, we conduct a preliminary study on clone and non-clone code pairs in the BCB dataset. Since

[1]https://github.com/CGCL-codes/TreeCen

the number of non-clone pairs in it exceeds 1,000,000 while the number of clone pairs is only 270,000, we investigate all clone pairs and 270,000 randomly selected non-clone pairs in the BCB dataset to ensure the fairness of the test.

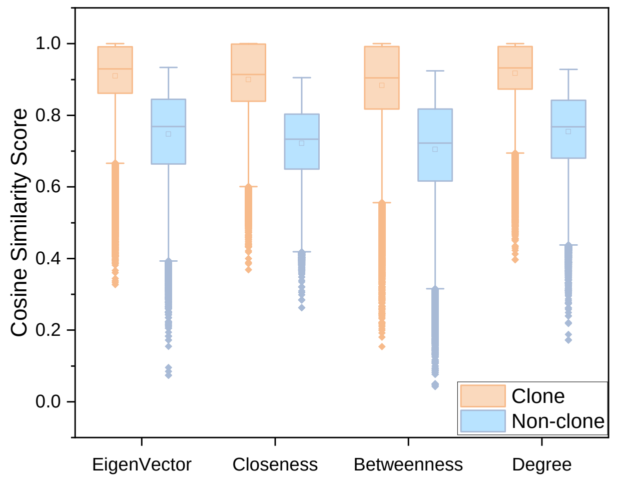


Figure 1: The cosine similarity distributions of several centrality measures of clone and non-clone code pairs

Given a code fragment, we first perform static analysis to extract the AST. To simplify the AST into a graph representation, we set each node type of the AST as a node in the graph, and then assign edges and edge weights between the nodes in the graph based on the edge relationships in the AST. After obtaining the graph, we select four popular centrality measures (*i.e., EigenVector Centrality* [13], *Closeness Centrality* [14], *Betweenness Centrality* [18], and *Degree Centrality* [19]) to generate centrality vectors. Finally, we calculate the cosine similarities of these centrality vectors and check whether centrality can show the inherent differences between clone and non-clone code pairs. In particular, the more similar the code pair is, the more the cosine similarity converges to 1. Conversely, the more different the code pair is, the more the cosine similarity converges to 0. As we can see from the box line plots in Figure 1, there are significant differences between clones and non-clone codes. The box plots for clone codes (orange ones) clearly are closer to 1 than for non-clone code pairs (blue ones). Therefore, the investigation results can answer the two above questions. In other words, an AST can be simplified into a graph representation, and the centrality analysis can reflect the structural information of the AST, which shows a significant discrepancy in the similarity between clone and non-clone code pairs.

Inspired by these insights, we propose a novel approach to transform complex ASTs into graphs (*i.e., tree graphs*) and perform centrality analysis on the *tree graphs* to obtain briefer code representation vectors for scalable code clone detection.

3 DEFINITION

In this paper, we adopt the definitions of code clones and clone types, following the previous work [42], [14].

Type-1 Clone (Textual Similarity): These code clones are identical code fragments, excluding spaces, blank lines, and comments.

Type-2 Clone (Lexical Similarity): The code in this category is identical code fragments except for some renamed unique identifiers (*i.e.,* function name, class name, variables).

Type-3 Clone (Syntactic Similarity): Code fragments that are syntactically similar differ only at statement- level. These fragments have statements added, modified, or deleted from each other.

Type-4 Clone (Semantic Similarity): This clone type indicates syntactically dissimilar fragments that implement the same functionality. Type-1, Type-2, and Type-3 clones demonstrate textual similarity, whereas Type-4 clones represent functional similarity.

Code Fragment: According to the granularity, a continuous segment of source code can be divided into *Token level*, *Statement level*, *Function level*, *File level*, and *Program level*. In this paper, we aim to conduct code clone detection at function level.

4 SYSTEM

In this section, we introduce our proposed clone detector, namely *TreeCen* (**Tree**-based code clone detector by using **Cen**trality).

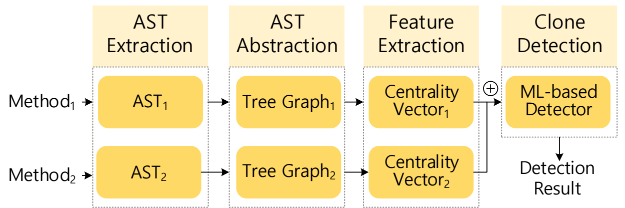


Figure 2: System overview of *TreeCen*

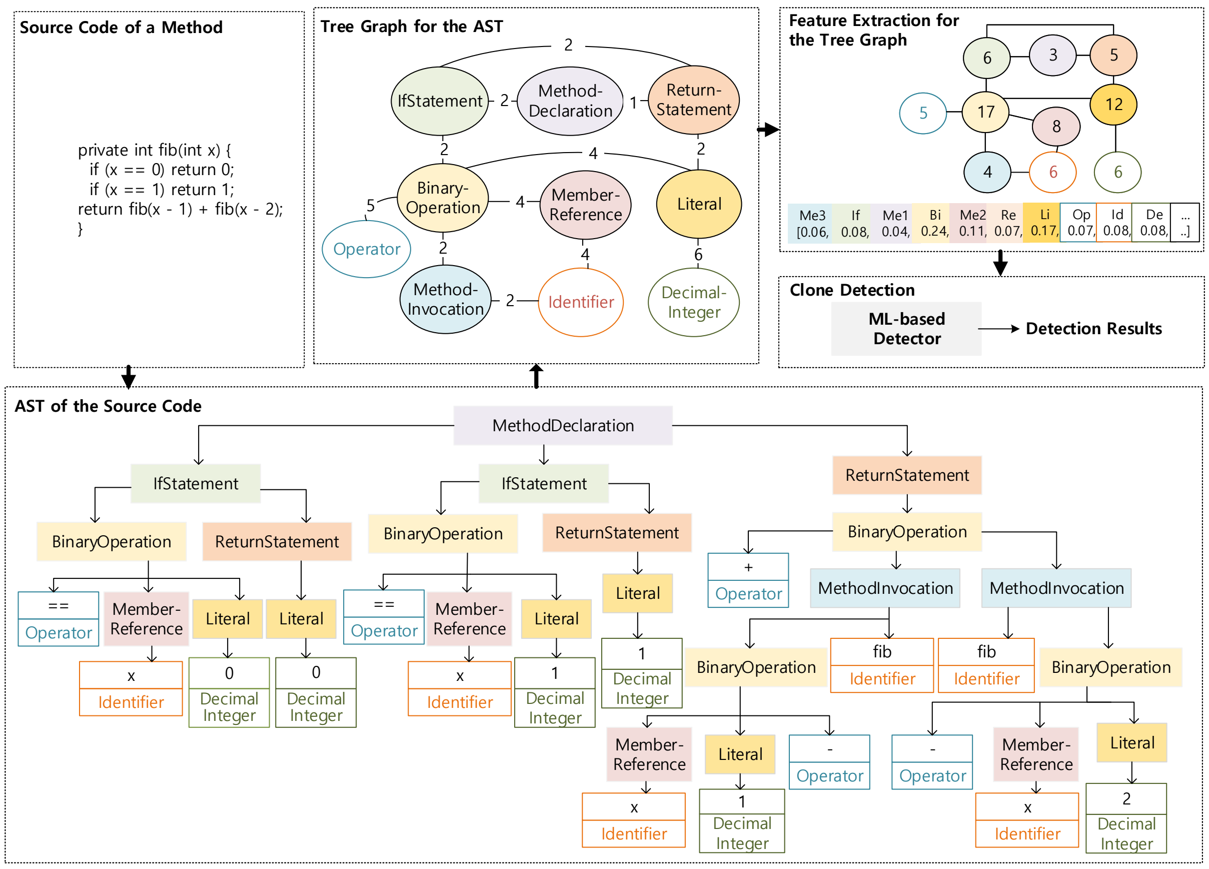
4.1 System Overview

The framework of *TreeCen* is shown in Figure 2. *TreeCen* consists of four main phases: *AST Extraction*, *AST Abstraction*, *Feature Extraction*, and *Clone Detection*.

- **AST Extraction:** This phase aims to extract the AST for a method based on static analysis, whose input is the source code, and the output is an AST.
- **AST Abstraction:** This phase is designed to simplify the AST while preserving its structural information. The input is an AST, and the output is a *tree graph* for the AST, where each node represents a node type in the AST, and each edge conveys an edge relationship in the AST.
- **Feature Extraction:** In this phase, we assign centrality to each node within the *tree graph* generated in the AST abstraction phase. The output is centrality vectors containing concrete features about the AST.
- **Clone Detection:** Given the centrality vector of a pair of codes, we concatenate them into one vector and then feed it into a machine learning model to train the detector after annotating the corresponding labels (clone or non-clone).

4.2 AST Extraction

This paper devotes to building a detector for scalable semantic code clones. To make a trade-off between scalability and semantic



```
private int fib(int x) {
  if (x == 0) return 0;
  if (x == 1) return 1;
return fib(x - 1) + fib(x - 2);
}
```

Figure 3: A detailed example for *TreeCen*

preservation, we adopt the AST as an intermediate representation of code. In terms of scalability, generating an accurate AST does not require compilation compared to other intermediate representations of code (*e.g.,* PDG and CFG). Therefore, it is adaptive to code fragments of all granularity written in all languages. As for semantics preservation, some works have proven that AST can reflect code semantics effectively for semantic clone detection [55].

We first perform static analysis to parse the source code and extract its corresponding tree representation. Then, we conduct the experiments on two common and standard clone detection datasets, namely BCB [1] and GCJ [4], written in Java. Therefore, we utilize *javalang* to perform static analysis on the source code to obtain ASTs. In terms of other languages, such as C/C++, we suggest that other code parsing tools such as *Joern* [6] can be applied. To describe the workflow of *TreeCen* more clearly, we have shown a detailed example in Figure 3.

4.3 AST Abstraction

In this phase, we abstract the AST extracted from the previous subsection to simplify the AST for scalable clone detection. The AST is a tree structure consisting of *token nodes*, *type nodes*, and *edges*. Specifically, *token nodes* are the leaf nodes of the AST (the double-layered rectangles in Figure 3) containing node types and corresponding code. The other nodes are *type nodes* (the single-layer rectangles in Figure 3), indicating the type of this node (e.g., `ReturnStatement`, `Literal`). The semantic and syntactic information of an AST is mainly captured in these node types and edges.

To abstract the AST, we compress the node types and edge relationships of ASTs into a tighter data structure, which helps to preserve comprehensive code information for scalable clone detection. To this end, we analyze all code fragments in the BCB dataset, more than 250 *million lines of code* (MLOC), and then count the node types of all their ASTs, finally finding 57 types of *type nodes*. For *token nodes*, the statistical results show that 14 types (occupying 99.5% of all *token nodes*) appear in most ASTs and the other types rarely appear (only 0.5%). Therefore, we safely assume these 14 types cover a majority of the subject systems and determine them. Meanwhile, we set a null type to represent the other types. In brief, in this paper, we consider that AST has 57 types of *type nodes* and 15 types of *token nodes*, for a total of 72 node types.

Based on the above observation, we design an undirected graph to represent the abstraction of AST, termed *tree graph*. In particular, each node type in the AST is regarded as a node in the *tree graph*, and the parent-to-child node relationship in the AST is considered an edge between the parent type and child node type in the *tree* *graph*. Furthermore, the weight of an edge in the *tree graph* depends on the number of edge occurrences between the corresponding node types in the AST.

For example, as shown in Figure 3, there are 10 nodes in the *tree graph* , which correspond to the 10 node types appearing in the AST, including three types of *token nodes* and seven types of *type nodes*. Moreover, the edge between node type `MemberReference` (light red single-layer rectangle) and node type `Identifier` (orange double-layer rectangle) appears four times in the AST. Therefore, we construct an edge between the `Identifier` node to `MemberReference` node in the *tree graph* and assign four as the weight for this edge, as presented in Figure 3.

In short, we simplify the AST to a graph representation (*i.e., tree graph*) consisting of at most 72 nodes in this subsection. Therefore, *TreeCen* can perform scalable code clone detection with the help of the *tree graph*.

4.4 Feature Extraction

In the feature extraction phase, we aim to further extract the centrality features of the nodes in the *tree graph* generated from the previous phase. In other words, the *tree graph* is transformed into a feature vector representation that reflects the semantics and syntax of the code. Several centrality methods have been proposed in the field of social networks [19], [29], [18], [13], [37], and we present the definitions of six of them briefly.

Degree Centrality [19] assigns an importance score based simply on the number of links held by each node. It is normalized by dividing by the maximum possible degree in a graph $N-1$, where N denotes the number of nodes within the graph, $deg(v)$ is the degree of node v.

$$C_d(v) = \frac{deg(v)}{N-1} \tag{1}$$

Katz Centrality [29] computes the relative influence of a node within a graph by measuring the number of the immediate neighbors and also all other nodes in the graph that connect to the node under consideration through these immediate neighbors. If $C_{katz}(i)$ denotes *Katz Centrality* of a node i, where the element at location (i, j) of the adjacency matrix A raised to the power k (*i.e.,* A^k) reflects the total number of k degree connections between nodes i and j. The α denotes an attenuation factor, then mathematically:

$$C_k(i) = \textstyle\sum_{k=1}^{\infty}\sum_{j=1}^{\infty}\alpha^k(A^k)_{ji} \tag{2}$$

Betweenness Centrality [18] characterizes the importance of a node in terms of the number of shortest paths through it. The *Betweenness Centrality* of a node v is given by below expression, where δ_{st} is the number of shortest paths from node s to node t and $\delta_{st}(v)$ is the number of those paths that pass through v.

$$C_b(v) = \textstyle\sum_{s\neq v\neq t}\frac{\delta_{st}(v)}{\delta_{st}} \tag{3}$$

EigenVector Centrality [13] considers that the importance of a node depends on the number of its neighboring nodes and the importance of its neighboring nodes. Assuming that x_i represents the importance of node i in a graph with n nodes. c is a proportionality constant, and $a_{ij} = 1$ if and only if i is connected to j, otherwise it is 0.

$$C_e(N_i) = x_i = c\textstyle\sum_{j=1}^{n}a_{ij}x_j \tag{4}$$

Closeness Centrality [14] measures how easy it is for a node to reach other nodes, which is the reciprocal of the average of the distances to all other nodes. The formula is as follow, where $dis(i, j)$ denotes the distance from node i to node j and n is the number of nodes in the graph.

$$C_c = \frac{1}{\sum_{j=1}^{n}dis(i,j)} \tag{5}$$

Harmonic Centrality [37] is designed to compute *Closeness Centrality* for disconnected graphs. Therefore, the portion of the *Harmonic Centrality* formula is consistent with the formula of the *Closeness Centrality*.

$$C_h = \textstyle\sum_{j=1}^{n}\frac{1}{dis(i,j)} \tag{6}$$

Additionally, to measure the importance of a vertex in a graph by combining multiple centrality measures, we construct two other kinds of integrated centrality measures (*i.e., Mean Centrality* and *Concatenate Centrality*). *Mean Centrality* vector is a 72-dimensional vector obtained by summing the above six single centrality vectors and averaging them by place.

$$C_{mean} = \frac{1}{6}(C_d + C_k + C_b + C_e + C_c + C_h) \tag{7}$$

Concatenate Centrality vector is a 432-dimensional vector (72∗6) generated by concatenating each single centrality vector. Mathematically, n denotes the dimension of the vector derived by individual centrality measure.

$$C_{concat} = [C_d, C_k, C_b, C_e, C_c, C_h] \in \mathbb{R}^{n\times 6} \tag{8}$$

Since *Degree Centrality* is the simplest computation of the centrality measures, it is chosen as the sample in Figure 3. Given the *tree graph*, we compute the *Degree Centrality* for each of the 72 types of nodes in turn. For example, there are 72 degrees in the *tree graph*, of which six are derived from the `Identifier` node (the circle with the orange border), eight connecting to the `MemberReference` (the light red circle), and four for `MethodInvocation` (the light blue circle), respectively. Therefore, the degree of `Identifier` is six. According to the *Degree Centrality* formula, the `Identifier` node's *Degree Centrality* can be calculated as 0.08. In addition, if there is no node of that type, the corresponding position of the vector is set to zero. After the feature extraction, we obtain a 72-dimensional centrality vector.

4.5 Clone Detection

Our last phase is clone detection, *i.e.,* identifying a code pair as clone or non-clone. To this end, we choose five candidate machine learning algorithms. They are *1-Nearest Neighbor* (1-NN), *3-Nearest Neighbor* (3-NN), *Decision Tree* (DT), *Random Forest* (RF), and *Logistic Regression* (LR). First, there are two centrality vectors, V_1 and V_2, generated by two code fragments in a code pair, and we concatenate them to obtain a vector representation V of that code pair, as shown in the following equation.

$$V = V_1 \oplus V_2 \tag{9}$$

Then we annotate the vector V of the code pair according to whether it is a clone or a non-clone pair. After dividing the labeled dataset into a training and test set, we perform 10-fold cross-validation based on the five machine learning models above. The input of this part is the centrality vector of a concatenated pair of codes, and the output is the result of clone detection (clone or non-clone).

5 EXPERIMENTS

In this section, our experiments are centered on answering the following *Research Questions* (RQs):

- *RQ1: What is the overall performance of TreeCen on code clone detection?*
- *RQ2: What is the detection performance of TreeCen compared to other state-of-the-art approaches?*
- *RQ3: What is the time performance of TreeCen compared to other state-of-the-art clone detectors?*

5.1 Dataset Description

We evaluate *TreeCen* on two widely used datasets, *i.e., Google Code Jam* (GCJ) [4] and *BigCloneBench* (BCB) [1]. The GCJ dataset comes from a programming competition held by Google Inc. We used the GCJ dataset collected by *DeepSim* [56], containing 1,669 projects from 12 competition problems. Since programs from the same competition problem aim to solve the same problem, all programs under a problem can be viewed almost as semantic clones (*i.e.,* Type-4 clones). In contrast, programs from different problems are considered non-clones.

Table 1: The summary of BCB and GCJ

Type		BCB	GCJ
Clone Pairs	T1	48,116	-
	T2	4,234	-
	ST3	21,395	-
	MT3	86,341	-
	T4	109,914	275,570
Non-clone Pairs		270,000	275,570

The BCB dataset is provided by Svajle *et al.* [46], which is manually assigned a clone type for each pair of functions. The BCB dataset composes of over 8,000,000 tagged clone pairs covering all clone types. In addition, it further divides Type-3 clones into *Strongly Type-3* (ST3), *Moderately Type-3* (MT3), and *Weakly Type-3* (WT3). However, the total number of code pairs obtained from the download database is slightly different from that reported in [47]. We filter out the cloned code pairs with less than five code lines. Finally, 270,000 cloned code pairs have remained, and 270,000 false tagged clone pairs are randomly selected to complete the training and testing. The detailed number of samples of the above datasets is shown in Table 1.

5.2 Experimental Settings

5.2.1 Implementation. We run all experiments on a standard server with 128GB RAM and 16 cores of CPU. For the above datasets, we adopt 10-fold cross-validation to train and evaluate *TreeCen*. Specifically, we divide the dataset into 10 subsets, and each time we pick one subset as the test set and the remaining nine subsets as the training set. We repeat this 10 times, each time choosing a different test subset. The results reported below are the average of these 10 times.

In the *AST Extraction* phase, we utilize a pure Python library, *javalang* [5], to parse Java source code and generate the AST for each method without compilation. In the *AST Abstraction* and *Feature Extraction* phases, we leverage a Python package, *networkx* [7], to construct the *tree graph* for the AST and then extract the multiple centrality vectors for them. In the clone detection phase, we use a Python package, *sklearn* [10], to implement machine learning models (*i.e.,* 1-NN, 3-NN, DT, RF, and LR).

5.2.2 Comparisions. We compare *TreeCen* with existing state-of-the-art code clone detection approaches. To ensure the comprehensiveness of the evaluation, we select representative work from each of the code intermediate representations for comparative experiments.

Token-Based Approach: These detection approaches are scalable, but hard to detect Type-4 clones.

- **SourcererCC** [45] detects clones by calculating the overlap similarity of tokens between two methods.
- **RtvNN** [53] is a RNN-based clone detection method for tokens.

Tree-Based Approach: These methods are able to detect semantic clones, especially MT3 clones.

- **Deckard** [26] is a popular detector by clustering the similar subtrees of ASTs.
- **ASTNN** [55] is based on deep learning that applies GRU to ASTs.

Graph-Based Approach: These tools are able to detect semantics clones yet suffer from a large execution time.

- **SCDetector** [54] trains a Siamese architecture neural network based on CFG.
- **DeepSim** [56] extracts data flow and control flow of code to train a DNN-based detection model.

5.2.3 Metrics. Since the code clone detection is a binary classification task, we adopt widely used metrics to measure its performance. The descriptions of our used metrics are as follows:

- *True Positive* (TP): the number of samples correctly classified as clone pairs.
- *True Negative* (TN): the number of samples correctly classified as non-clone pairs.
- *False Positive* (FP): the number of samples incorrectly classified as clone pairs.
- *False Negative* (FN): the number of samples incorrectly classified as non-clone pairs.
- *Precision=TP/(TP+FP).* The correct rate of detection.
- *Recall=TP/(TP+FN).* The percentage of clone pairs that are successfully detected.
- *F1=2*Precision*Recall/(Precision+Recall).* A comprehensive metric of detection.

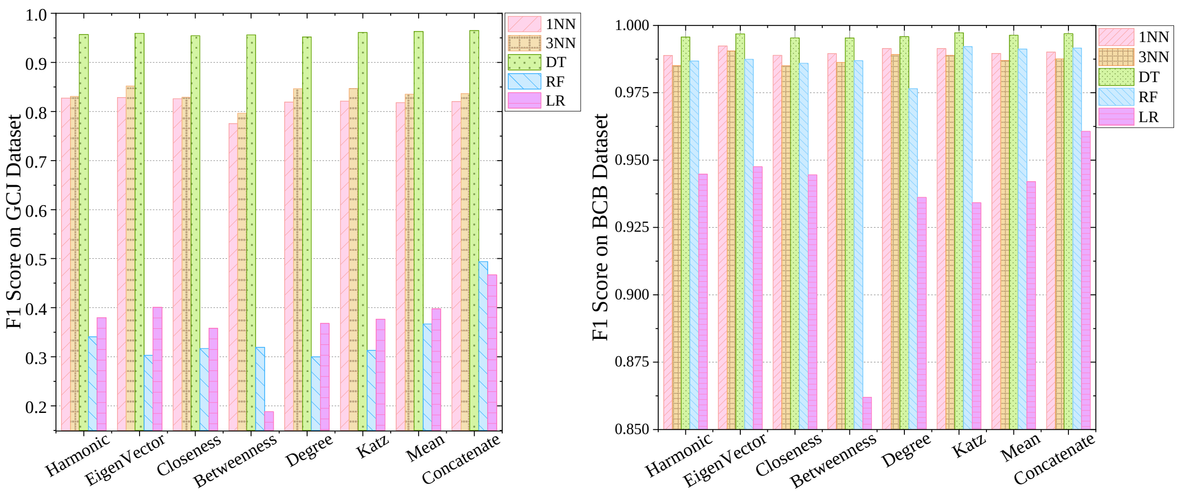


Figure 4: F1 scores of *TreeCen* on GCJ and BCB datasets

5.3 RQ1 : Overall Effectiveness

In this subsection, we evaluate how well *TreeCen* is on code clone detection. To this end, we conduct experiments on the GCJ dataset and BCB dataset, as shown in Table 1. In order to evaluate which machine learning algorithm is more applicable to *TreeCen* for clone and non-clone classification tasks, we first employ several machine learning models (*i.e.,* 1-NN, 3-NN, DT, RF, and LR) for targeted centrality measures. Figure 4 represents the clone detection results achieved by *TreeCen* using above mentioned machine learning algorithms. Specifically, we adopt the F1 score for this experiment as it is a comprehensive metric for clone detection tasks that can effectively illustrate detection effectiveness.

As shown in Figure 4, we can see that the DT-based detector offers the most satisfactory detection performance than that based on other machine learning models on the GCJ and BCB datasets. For the GCJ dataset, the DT-based detector achieves an F1 score of up to 95% or higher for each of the eight centrality measures. As for the BCB dataset, the DT-based detector also outperforms other machine learning approaches, with an F1 score above 99.5% for all the centrality measures. Therefore, we finally choose DT as the final detector for *TreeCen* based on the above experimental results.

For the performance of various centrality measures, we can see that the DT-based classifier does not distinguish much from the detection results of all centrality measures, with an F1 score between 95% and 97% in Figure 4. Among them, the *Concatenate Centrality* is the most effective, which is able to achieve an F1 score of 96.5% on the GCJ dataset. We consider the reasons for the good performance of the *Concatenate Centrality* consisting of all centrality measures, which means it covers the most comprehensive code semantics.

The detection with various centrality measures has also achieved good results on the BCB dataset. The F1 score can achieve between 99.5% and 99.9% with multiple centrality measures, as shown in Figure 4, whose gaps are also relatively subtle. Additionally, the F1 score of *Concatenate Centrality* and *Katz Centrality* both can maintain more than 99.8% . In detail, the *Katz Centrality* measure performs well since its algorithm considers more node relationships and thus obtains more detailed information about the graph.

Based on the above analysis, there is a slight discrepancy in the detection effectiveness of classifiers with different centrality features. To conduct a more scalable clone detector, we determine to use the time overhead to specify the final centrality measure. Obviously, due to the simplicity of the *Degree Centrality* algorithm, the time overhead for generating *Degree Centrality* features is the shortest among the listed centrality measures. In subsection 5.5, we provide a statistical and detailed description of the time overhead of each component, and the results show that the *Degree Centrality* measure costs the least amount of time in all processions (*i.e.,* centrality generation, training, and testing). So we adopt the *Degree Centrality* measure for *TreeCen.*

In summary, the detector based on DT achieves the best detection performance, and the *Degree Centrality* obtains the lowest time overhead. Thus, *TreeCen* adopts DT as the final detector with the feature vectors generated by the *Degree Centrality.*

5.4 RQ2 : Effectiveness Comparison with Others

5.4.1 Results on Google Code Jam. We first conduct comparative experiments with the state-of-the-art clone detectors on the *Google Code Jam* (GCJ) dataset. The sample volumes in the GCJ dataset are shown in Table 1, containing 275,570 clone method pairs and 1,116,376 non-clone method pairs. As described in Section 5.1, we assume that all the clone pairs in the GCJ dataset are Type-4 clones. Therefore, conducting experiments on this dataset can effectively prove the effectiveness of *TreeCen* for semantic clone detection.

The GCJ dataset is first collected and provided by *DeepSim* [56], which performs the same experimental comparisons with *Deckard* [26] and *RtvNN* [53] on this dataset. *SCDetector* [54] conducts the experiments on the GCJ dataset as well. To make a fair comparison, we directly adopt the experimental results provided in [56] and [54]. Table 2 illustrates the experimental results on the GCJ dataset.

Table 2: Results on Google Code Jam

Groups	Methods	Recall	Precision	F1
Token-based	SourcererCC	0.11	0.43	0.17
	RtvNN	0.90	0.20	0.33
Graph-based	DeepSim	0.82	0.71	0.76
	SCDetector	0.87	0.90	0.89
Tree-based	Deckard	0.44	0.45	0.44
	ASTNN	0.87	0.95	0.91
	TreeCen	**0.95**	**0.95**	**0.95**

Token-based methods suffer from poor performances on the GCJ dataset. The reason is that the token-based approaches only consider the syntactic features of the code and ignore the semantic features, making it hard to detect semantically similar (Type-4 clones) code pairs. An interesting observation is that *RtvNN* obtains high recall but low precision, which means that it detects most of the code pairs as clone pairs directly. This reflects the drawback of token-based detectors, where two completely different semantic codes are considered a clone pair even when they use part of similar syntactic components (*e.g.,* the common library API, arithmetic expression).

Graph-based methods obtain more comprehensive code semantic information with the help of PDG or CFG, and therefore yield good detection results. In this category, *DeepSim* is designed based on both data flow and control flow to train a *deep neural networks* (DNN) model, while *SCDetector* achieves better detection results using control flow only. The reason behind this may be that the Siamese architecture neural network effectively helps *SCDetector* improve the detection performance, as mentioned in [54]. However, the recall of *SCDetector* is only 0.87, which is 8% lower than that of *TreeCen*. Likewise, as a detection method using the centrality analysis, *TreeCen* employs centrality analysis to highlight the structural knowledge of the AST further, thus avoiding the false negatives introduced by *SCDetector* conducting centrality analysis on tokens.

Tree-based methods achieve good detection performances on the GCJ dataset. Both *ASTNN* and *Deckard* employ the similarity of subtrees of AST for clone detection. *ASTNN* works better than *Deckard*. *Deckard* considers the similarity of feature vectors of parser tree roots rather than semantic information. Nevertheless, *ASTNN* uses bottom-up aggregation of leaf nodes to root nodes, which considers more comprehensive semantic information than *Deckard*. However, both methods underperform compared to *TreeCen*, as shown in Table 2. We consider the reason is that *TreeCen* does not split AST into subtrees, which avoids some of the code semantic loss (*e.g.,* `if/else` branches may be lost in splitting subtrees). The experimental results further show that *TreeCen* achieves the best detection performance on the semantic clone dataset, demonstrating that the AST abstraction (*i.e., tree graph*) and feature extraction (*i.e.,* centrality analysis) help *TreeCen* retain the comprehensive code semantics.

Compared to *SourcererCC*, *RtvNN*, *DeepSim*, *SCDetecotr*, *Deckard*, and *ASTNN*, *TreeCen* achieves the best detection performance on the GCJ dataset. Specifically, *TreeCen* obtains a 4% improvement over the current state-of-the-art with an F1 score of 95% for semantic clone detection.

5.4.2 Results on BigCloneBench. Tables 3 and 4 demonstrate the clone detection results on the BCB dataset. As shown in Table 3, *TreeCen* performs significantly better than other clone detectors on the entire dataset, with both its precision and recall being above 0.99. The F1 score of 0.99 better demonstrates the excellent performance of *TreeCen* for all clone types. In fact, the F1 score of *TreeCen* is 0.996 reported in our experiments. However, we show 0.99 in Table 3 to avoid ambiguity.

For each type of code clone detection, *TreeCen* also exceeds most detectors to detect different types of code clones, as can be observed from Table 4. In particular, it is the best performing detector when detecting Type-1/Type-3/Type-4 clones. For example, when detecting semantic clones (*i.e.,* Type-4 clones), the F1 scores of *SourcererCC*, *RtvNN*, *DeepSim*, *SCDetecotr*, *Deckard*, and *ASTNN* are 0.02, 0.00, 0.97, 0.95, 0.02, 0.92, respectively, while the F1 score of *TreeCen* is 0.99.

Table 3: Results on BigCodeBench

Groups	Methods	Recall	Precision	F1
Token-based	SourcererCC	0.07	0.98	0.14
	RtvNN	0.01	0.95	0.01
Graph-based	DeepSim	0.98	0.97	0.98
	SCDetector	0.92	0.97	0.95
Tree-based	Deckard	0.06	0.93	0.12
	ASTNN	0.94	0.92	0.93
	TreeCen	**0.99**	**0.99**	**0.99**

Table 4: F1 score for each clone type

Methods	T1	T2	T3	MT3	WT3/T4
SourcererCC	1.00	**1.00**	0.65	0.20	0.02
RtvNN	1.00	0.97	0.60	0.03	0.00
DeepSim	0.99	0.99	0.99	0.99	0.97
SCDetecotr	1.00	**1.00**	0.97	0.97	0.95
Deckard	0.73	0.71	0.54	0.21	0.02
ASTNN	1.00	**1.00**	0.99	0.98	0.92
TreeCen	**1.00**	0.99	**1.00**	**1.00**	**0.99**

However, the detection performance of *TreeCen* in the Type-2 clone is not the best. To find the reason for this situation, we conduct statistics on the number of clone pairs in the BCB dataset, as listed in Table 1. We notice that the number of Type-2 clones in the BCB dataset is much smaller than that of other types, only 1.6% of the whole dataset. Therefore, we infer that the insufficient sample size makes the machine learning model not learn sufficient features of Type-2 clones, resulting in the final detection results appearing lower than several other clone types. It is believed that the performance of *TreeCen* for detecting Type-2 clones will also be improved if the number of Type-2 clone pairs increases in the subsequent work.

To conclude, *TreeCen* can detect different clone types, especially for the semantic clones that are difficult to detect. It achieves an

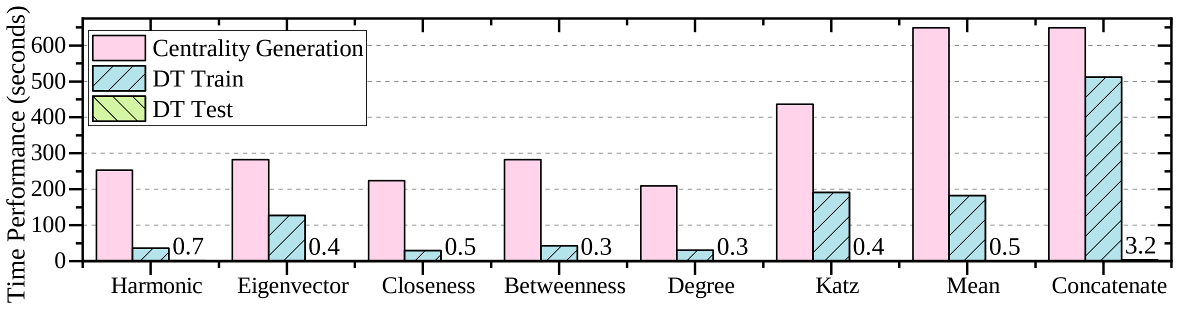


Figure 5: Time performance for *TreeCen*

ideal performance on the GCJ and BCB datasets compared to the clone detectors.

5.5 RQ3 : Scalability Evaluation

5.5.1 Time performance of TreeCen. In this phase, we focus on the effect of using different centrality measures on the time overhead of *TreeCen*. As a preparation for the experiment, we randomly select one million code pairs from the GCJ dataset. The experimental results are obtained by averaging the time after running each detector three times, as presented in Table 5 and Figure 5. The time overhead of *TreeCen* consists of three main components: centrality generation, decision tree model training, and testing, as illustrated in Figure 5. To be specific, the centrality generation consists of the progress of the AST generation, AST abstraction, and centrality analysis.

For mixed centrality measures, we find that the time cost is significantly higher than that for other individual centrality measures. It is because the mixed measures generation requires calculating the multiple individual centrality vectors and then concatenating or averaging them. Obviously, it requires more time in centrality generation. In addition, the vector derived from the mixed centrality measures is more complex, especially the *Concatenate Centrality*. Its feature vector is six times more dimensions than others, leading to a longer training time.

For individual ones, there is a slight difference in the time overhead of centrality generation. The *Katz Centrality* and the *EigenVector Centrality* cost the longest time, while the *Degree Centrality* needs the least. As introduced in section 4.4 for each centrality measure, it can be seen that the algorithm of *Degree Centrality* is the simplest one that only considers the outgoing and incoming degrees of nodes. In contrast, *Katz* and *EigenVector Centrality* consider more graph node relationships, which we believe is the origin of the efficiency discrepancy in centrality generation.

In terms of model training time, the *Degree Centrality* is significantly faster than that of *EigenVector* and *Katz Centrality* measures. To explain this phenomenon, we compare the feature vectors derived from *Degree Centrality* with those generated from *EigenVector Centrality* and *Katz Centrality*. It can be observed that the feature vectors generated by *Degree Centrality* are clearly simpler than those calculated by other centrality measures. Evidently, the discrimination between the simplicity vectors is more obvious, making it easier for the decision tree to find decision boundaries and thus shortening the model training time.

In summary, due to the superior performance of *Degree Centrality* in terms of time overhead (*i.e.,* 208.7s of data preprocessing time, 30.2s of model training time, and 0.3s of prediction time), we choose to adopt *Degree Centrality* as the centrality measure of *TreeCen*.

5.5.2 Time performance comparisons. In this part, we pay attention to the runtime performance of *TreeCen* and six comparative detectors. As shown in Table 5, since most tools are designed based on deep learning or machine learning, we test for time overhead by training and testing overhead. Specifically, we count the time for data preprocessing in the training time. Note that the results presented in Table 5 are derived from testing one million randomly selected samples from the GCJ dataset.

Table 5: Time performance on GCJ

Groups	Methods	Training	Prediction
Token-based	SourcererCC	-	30s
	RtvNN	5,206s	35s
Graph-based	DeepSim	13,545s	34s
	SCDetector	2,937s	139s
Tree-based	Deckard	-	72s
	ASTNN	16,096s	2,894s
	TreeCen	**238.9s**	**0.3s**

Token-based methods are scalable, as evidenced by the time overhead of *SourcererCC*. As a purely token-based detector, *SourcererCC* detects code clones by comparing vector similarity composed of word frequencies. Therefore, it does not require training time. Although the overall time overhead of *SourcererCC* is low, it is almost impossible to detect semantic clones (with an F1 score of 0.14 on the GCJ dataset), as shown in Table 2. So it is not applicable for large-scale semantic clones. *RtvNN* has similar disadvantages mentioned above and a relatively high time overhead.

Graph-based methods have a higher time overhead due to excessive consideration of code semantics. First, both *DeepSim* and *SCDetector* are deep learning-based detection methods, which require a long time to train deep learning models. Second, to ensure that accurate PDGs and CFGs are available, they need to compile the code. However, *TreeCen* does not require compilation for AST generation and is designed based on machine learning models. So it is at least ten times faster than graph-based deep learning detectors in terms of training time. That is, the training time of *TreeCen* is only 238.9s, while those of *DeepSim* and *SCDetector* are 13,545s and 2,937s, respectively. Additionally, predicting by *TreeCen* takes 100 times less time than the graph-based methods. In short, *TreeCen* has much less time overhead than the graph-based approaches while enabling the detection of semantic clones.

Tree-based methods are efficient for the clone detection task. *Deckard* detects clones by clustering similar subtrees, so it has zero training time. However, as shown in Table 2 and Table 4, it suffers from poor performance when detecting Type-4 clones. The detection effect of *ASTNN* is acceptable, but it takes the longest time among all the comparative tools. Specifically, *ASTNN* has a training time of 16,096s and a testing time of 2,894s, while *TreeCen* takes only 238.9s for training and 0.3s for testing. The difference of their time overhead can be attributed to the implementation of classifier. The classifier used by *TreeCen* is decision tree, which is faster than that utilized by *ASTNN*. Specifically, *ASTNN* leverages a bi-directional RNN model with 100 hidden dimensions and bi-directional GRUs. In addition, *ASTNN* generates a 128-dimensional vector for each AST, while the AST feature vector extracted by *TreeCen* is only 72-dimensional. Obviously, the lower vector dimensionality makes model training and detection much faster.

In summary, *TreeCen* has the lowest time overhead (79 times faster than *ASTNN*) among the AST-based semantic detectors. The results of comprehensive experiments and analysis show that our approach can efficiently detect semantic clone as well as accomplish scalable clones detection.

6 DISCUSSION

6.1 Differences from *SCDetector*

The most similar related method to *TreeCen* is *SCDetector* [54]. *SCDetector* applies centrality analysis for CFG to detect clones, while *TreeCen* adopts centrality measures for *tree graphs* generated from ASTs. First, the difference in the intermediate representations of the code chosen by *TreeCen* and *SCDetector* leads to their distinctions. Since *SCDetector* is a CFG-based detector, the code needs to be compiled before extracting the CFG. Conversely, *TreeCen* is based on AST and does not require compilation. Hence *TreeCen* is extensible to work with code fragments of arbitrary granularity (*e.g.,* line of code level). In addition, *TreeCen* can also be applied to other languages (*e.g.,* C/C++), as it is practicable to extract ASTs from source code written in any language.

Second, the centrality measure is applied differently on detectors. *SCDetector* is based on the degree centrality of tokens and relies on the common tokens between two programs, leading to the false negatives when the same functionality is implemented using different APIs and different graph structures. In contrast, *TreeCen* obtains a *tree graph* by abstracting the AST and then considers each node in the *tree graph* (*i.e.,* the node type in the AST) as a node in the social network and digs out the centrality for it. By this, *TreeCen* pays more attention to the structural information of the code, thus avoiding false negatives caused by different tokens in code pairs with the same semantics. Therefore, *TreeCen* achieves better detection performance than *SCDetector*.

Finally, *SCDetector* adopts a Siamese architecture neural network as the classifier, meaning that training phase requires the use of GPUs. However, *TreeCen* leverages a decision tree model for training the detector, which is a simple yet effective machine learning model, making *TreeCen* require a little time and computational resources to train and predict. Therefore, *TreeCen* is more scalable than *SCDetector*.

6.2 Why *TreeCen* Performs Better

In terms of effectiveness, *TreeCen* outperforms the other methods on both experimental benchmark datasets. The reason is that the *tree graph* generated by *TreeCen* can effectively retain the structural information of the AST, thus reflecting the semantic and syntactic information of the code. As demonstrated in the preliminary study, centrality measures are widely used in social network analysis to determine the important nodes in the graph or network, thus preserving the network's structure. Similarly, the application of centrality measures in AST further highlights the significant nodes in the AST. Therefore, centrality analysis helps capture the AST's tree structure information to distinguish between cloned and non-cloned code pairs.

In terms of efficiency, *TreeCen* is 79 times faster than the most efficient method (*i.e., ASTNN*) in the same category (tree-based). Compared with the existing tree-based methods, instead of comparing ASTs directly for similarity, we first simplify the complex ASTs into *tree graphs* and then perform centrality analysis. As a result, the vector we feed to the detector has only 144 dimensions ($72*2$), which is easy for the machine learning model to learn and thus helps to reduce the training time. *TreeCen* is more scalable than the graph-based methods, which need time to compile the code before generating accurate graph representations. Another contributing factor is that both graphs and trees have high complexity structures, so comparing similarities is a heavy overhead problem.

6.3 Future Work

TreeCen is a clone detection method that uses AST as a code representation. Specifically, after the AST is refined and transformed into a novel graph, a centrality feature matrix is generated for the graph for model training and detection. In total, *TreeCen* encodes 72 AST node types (57 non-leaf AST nodes and 15 tokenized leaf nodes) during the transformation into the *tree graph*. In the follow-up research, we find that not all the above types of nodes provide valuable information for the clone detection task. We will continue to investigate and confirm which node types can be ignored in the future. To this end, the structure of a *tree graph* can be further simplified into a simpler data structure, which might be more efficient to scale. In addition, Java is an evolving language (with a new version every six months). New language features are typically implemented as new AST node types in parsers. To enable *TreeCen* to be more extensible to support new language features represented as new AST node types, we consider leveraging dynamic machine learning models (*e.g.,* , incremental learning and continue learning) instead of the current detection model in the future.

7 RELATED WORK

7.1 Semantic Clone Detection

Detecting semantic code clones (Type-4) is a complex research problem. Recently, some detectors have been proposed that can be applied for semantic clone detection. Most of these approaches are implemented based on graphs (*i.e.,* PDGs, CFGs) or trees (*i.e.,* ASTs) since these code representations can preserve more code semantics.

For the graph-based methods, PDG and CFG are inherently coding semantic representations, so they can be used to detect semantic clones effectively. *SCDetector* [54] is designed to detect semantic clones by combining the advantage of the token-based and graph-based methods. *FA-AST* [50] constructs a novel AST by adding control flow and data flow edges to make the method consider the code semantic information. *FCCA* [23] incorporates multiple code representations, such as text, AST, and CFG. *DeepSim* [56] encodes CFG and *Data Dependency Graph* (DDG) into a more compact semantic feature matrix and then classifies them using DNN. *FCDetector* [17] leverages a joint representation for deep learning model training.

For tree-based methods, a well-designed approach is able to detect semantic clones effectively. *CDLH* [52] leverages a *Long short-term memory* (LSTM) model to learn the features of the AST and uses hash to optimize the similarity of the method. *TBCNN* [39] proposes a tree-based convolutional kernel consisting of a set of subtree feature detectors that slide the entire AST to extract structural information of the code, with a *Convolutional Neural Networks* (CNN) model for code classification. *CodeRNN* [33] obtains the final code representation vector by a tree from the leaf nodes to the root node for *recurrent neural network* (RNN) training. *ASTNN* [55] divides a complete AST into a sequence of small statement trees to feed the deep learning model to overcome the long-term dependency problem caused by large ASTs. *InferCode* [15] decomposes the AST into subtrees and then encodes the nodes of the subtrees with a modified tree-based convolutional neural network. *HELoC* [51] designs a comparative learning network that allows the network to learn more explicit information about the AST hierarchy nodes.

However, above approaches require long execution time, leading to a limit in scalability. On the one hand, although there exist some means to obtain graphs without compilation, such graphs are not accurate. Therefore, these detectors still generate graphs by compiling to ensure the accuracy of detection. But compilation is inconvenient and infeasible for some code samples and also increases the time overhead. On the other hand, both the graph-based and tree-based methods suffer from the complex structure of representations, resulting in a high overhead for matching. They do not have a good balance of semantic detection and time overhead, and those that can detect Type-4 clones require a long model training time.

7.2 Scalable Clone Detection

Some empirical studies have mentioned the need for large-scale clone detection [12], [24], [20], [38], [48]. There are several different techniques to make the clone detector more scalable.

The token-based or text-based methods are always scalable, whose main idea is to compute the similarity of a pair of code snippets in terms of text or code token sequences. Specifically, these clone detectors [49], [32], [21], [28], [27], [43] directly converts the code into text or string, while the token-based approaches employ lexical analysis on the code to generate token sequences. These clone detection approaches require a little time to execute, and some can even achieve cross-language code clone detection. However, they do not consider the semantic information of the code, which makes it difficult to cope with the Type-4 clone detection.

Several scalable clone detection methods or tools have been proposed with the GPU computation. Solutions for GPGPU programming include Nvidia's CUDA [8] and AMD's CTM [22]. *SAGA* [31] proposes an efficient suffix-array based detector with sophisticated GPU acceleration. [30] make use of GPU for dynamic programming matching to detect code clones. However, it is a good idea to use GPUs for clone detection acceleration, and we believe that leveraging GPU acceleration to *TreeCen* in subsequent work can further reduce the time overhead.

Some commercial tools and methods for clone detection of code repositories have also been proposed. The commercial tools such as *BlackDuck* [2], *Scantist* [9], and *FOSSID* [3] are able to handle clone code scanning for more than 10 million lines of code, but semantic clone detection is not yet supported. *DCCD* [11] and *D-ccfinder* [35] conduct the clone detection with a network of computers. Besides, several works [36], [41], [44], [40] construct publicly available clone detectors at file granularity for code repositories.

However, those scalable approaches cannot detect semantic clones limited by their algorithms. Compared with the above methods, *TreeCen* is a code clone detector with both semantic clone detection capability and scalability.

8 CONCLUSION

In this paper, we propose *TreeCen*, a scalable tree-based semantic code clone detector. We first extract ASTs based on static analysis for the source code and transform them into simple graph representations (*i.e., tree graph*) according to the node type, rather than traditional AST matching. Then we treat the *tree graph* as a network and adopt the centrality analysis to convert it into a fix-length vector. By this, the final vector is only 72-dimensional but contains complete structural information of the AST. Finally, these vectors are fed into the machine learning model to detect clones. The experimental results show that *TreeCen* outperforms other six state-of-the-art methods (*i.e., SourcererCC*, *RtvNN*, *DeepSim*, *SCDetector*, *Deckard*, and *ASTNN*). In terms of scalability, *TreeCen* is 79 times faster than another state-of-the-art tree-based code clone detector (*i.e., ASTNN*), 13 times faster than the fastest graph-based approach (*i.e., SCDetector*), and even about 22 times faster than the one-time trained token-based detector (*i.e., RtvNN*).

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211 and Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032.

REFERENCES

[1] 2022. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[2] 2022. Blackducks. https://www.blackducksoftware.com.
[3] 2022. Fossid. https://fossid.com.
[4] 2022. Google Code Jam. https://code.google.com/codejam/contests.html.
[5] 2022. javalang. https://github.com/c2nes/javalang.
[6] 2022. joern. https://joern.io.
[7] 2022. networkx. https://github.com/networkx/networkx.
[8] 2022. Nvidia. https://www.nvidia.com.
[9] 2022. Scantist. https://scantist.com.
[10] 2022. sklearn. https://scikit-learn.org/stable.
[11] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. 2018. DCCD: An Efficient and Scalable Distributed Code Clone Detection Technique for Big Code.. In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering (SEKE'18)*. 354–353.
[12] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. 2005. Cloning by accident: An empirical study of source code cloning across software systems. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering (ISESE'05)*. 10–30.
[13] A. J. Alvarez-Socorro, G. C. Herrera-Almarza, and L. A. González-Díaz. 2015. Eigencentrality based on dissimilarity measures reveals central nodes in complex networks. *Scientific Reports* 5, 1 (2015), 1–10.
[14] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.
[15] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. 1186–1197.
[16] Nigel Coles. 2001. It's not what you know—it's who you know that counts. Analysing serious crime groups as social networks. *British Journal of Criminology* 41, 4 (2001), 580–594.
[17] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. [n.d.]. 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)*. 516–527.
[18] Linton C. Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.
[19] Linton C. Freeman. 1978. Centrality in social networks conceptual clarification. *Social Networks* 1, 3 (1978), 215–239.
[20] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*. 291–301.
[21] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. 219–228.
[22] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU Gems* 3, 39 (2007), 851–876.
[23] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.
[24] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2012. Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*. 387–391.
[25] Hawoong Jeong, Sean P. Mason, A.-L. Barabási, and Zoltan N. Oltvai. 2001. Lethality and centrality in protein networks. *Nature* 411, 6833 (2001), 41–42.
[26] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 96–105.
[27] Toshihiro Kamiya. 2021. CCFinderX: An interactive code clone analysis environment. In *Code Clone Analysis*. 31–44.
[28] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[29] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.
[30] Thierry Lavoie, Michael Eilers-Smith, and Ettore Merlo. 2010. Challenging cloning related problems with GPU-based algorithms. In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*. 25–32.
[31] Guanhua Li, Yijian Wu, Chanchal K. Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: Efficient and large-scale detection of near-miss clones with GPU acceleration. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. 272–283.
[32] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A deep learning-based clone detection approach. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 249–260.
[33] Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, Vol. 32.
[34] Xiaoming Liu, Johan Bollen, Michael L. Nelson, and Herbert Van de Sompel. 2005. Co-authorship networks in the digital library research community. *Information Processing & Management* 41, 6 (2005), 1462–1480.
[35] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. 2007. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 106–115.
[36] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A map of code duplicates on GitHub. *Proceedings of the 2017 ACM on Programming Languages (OOPSLA'17)*, 1–28.
[37] Massimo Marchiori and Vito Latora. 2000. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications* 285, 3-4 (2000), 539–546.
[38] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2017. Does cloned code increase maintenance effort?. In *Proceedings of the 11th International Workshop on Software Clones (IWSC'17)*. 1–7.
[39] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718* (2014).
[40] Manziba Akanda Nishi and Kostadin Damevski. 2018. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software* 137 (2018), 130–142.
[41] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File cloning in open source java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 283–292.
[42] Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.
[43] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*. 172–181.
[44] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. 2015. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process* 27, 6 (2015), 402–429.
[45] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 1157–1168.
[46] Svajlenko et al. 2021. Bigclonebench. In *Code Clone Analysis*. 93–105.
[47] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. 131–140.
[48] Jeffrey Svajlenko and Chanchal K. Roy. 2017. CloneWorks: A fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 177–179.
[49] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 1066–1077.
[50] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. 261–271.
[51] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. HELoC: Hierarchical Contrastive Learning of Source Code Representation. *arXiv preprint arXiv:2203.14285* (2022).
[52] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *Proceedings of the 2017 International Joint Conferences on Artificial Intelligence (IJCAI'17)*. 3034–3040.
[53] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE'16)*. 87–98.
[54] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. 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)*. 821–833.
[55] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 783–794.
[56] Gang Zhao and Jeff Huang. 2018. Deepsim: Deep learning code functional similarity. In *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. 141–151.