Tritor: Detecting Semantic Code Clones by Building Social Network-Based Triads Model

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

Siyue Feng[*†]
Huazhong University of Science and Technology, China
fengsiyue@hust.edu.cn

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

Wenqi Suo[*†]
Huazhong University of Science and Technology, China
suowenqi@hust.edu.cn

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

ABSTRACT

Code clone detection refers to finding the functional similarities between two code fragments, which is becoming increasingly important with the evolution of software engineering. Numbers of code clone detection methods have been proposed, including tree-based methods that are capable of detecting semantic code clones. However, since tree structure is complex, these methods are difficult to apply to large-scale clone detection. In this paper, we propose a scalable semantic code clone detector based on semantically enhanced abstract syntax tree. Specifically, we add the control flow and data flow details into the original tree and regard the enhanced tree as a social network. Then we build a social network-based triads model to collect the similarity features between the two methods by analyzing different types of triads within the network. After obtaining all features, we use them to train a machine learning-based code clone detector (*i.e., Tritor*). Our comparative experimental results show that *Tritor* is superior to *SourcererCC*, *RtvNN*, *Deckard*, *ASTNN*, *TBCNN*, *CDLH*, and *SCDetector*, are equally good with *DeepSim* and *FCCA*. As for scalability, *Tritor* is about 39 times faster than another current state-of-the-art tree-based code clone detector *ASTNN*.

CCS CONCEPTS

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

KEYWORDS

Semantic Clones, Abstract Syntax Tree, Social Network, Triads

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

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616354

ACM Reference Format:
Deqing Zou, Siyue Feng, Yueming Wu, Wenqi Suo, and Hai Jin. 2023. Tritor: Detecting Semantic Code Clones by Building Social Network-Based Triads Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616354

1 INTRODUCTION

As the field of software engineering is constantly evolving, the demand for software is increasing. As a result, many software developers choose to build or maintain software code by code cloning to save time and effort. In reality, code clones are divided into syntactic and semantic clones. Syntactic clones are usually found when copying and pasting code, and are divided into three types in descending order of similarity, namely Type-1 (textual similarity), Type-2 (lexical similarity), and Type-3 (syntactic similarity). Semantic clones are usually introduced when using different code syntax to implement a same functionality, which is Type-4 (semantically similarity). Although code cloning facilitates software development, it also increases maintenance costs and even causes the propagation of vulnerabilities, which can have a negative impact on software security [17, 31, 40, 48]. Therefore, code clone detection is more and more important in the field of software engineering.

A number of code clone detection techniques have been proposed. For example, the token-based detection technique *CCFinder* [32] performs lexical analysis of the code to extract the token sequence, which is then converted into a rule form to detect Type-1 and Type-2 clones. Token-based approaches can also detect a fraction of Type-3 clones, such as *SourcererCC* [47]. It detects Type-3 clones by calculating the overlapping similarity between the tokens of two methods. However, as only the program syntax is considered, these token-based methods cannot detect semantic clones. To solve this problem, researchers intend to capture semantic information by extracting intermediate representations of programs, thus equipping these methods with the ability to detect semantic clones. Graph-based approaches [35, 36, 52, 61, 62] extract graph structures (*e.g.,* control flow graph) containing semantic details of programs, and then use graph analysis to implement code clone detection. However, graph-based approaches usually have a high time overhead and thus are not scalable to large datasets. This is slightly mitigated by tree-based methods [28, 29, 38, 55, 60], which detect

semantic clones by obtaining a tree representation of the program and using tree matching. However, the problem of the lack of high scalability has not been completely solved, because although the tree analysis algorithm is lighter than the graph comparison algorithm, the tree structure is still complex [57]. For example, all the nodes and the child edges in Figure 4 are the original *abstract syntax tree* (AST) generated for the code in Figure 4. We can see from the figure that a simple function with only 11 lines generates a complex subtree of 44 nodes, not to mention more complex programs. As a result, the tree analysis also incurs a significant overhead and is difficult to be applied to large-scale code clone analysis.

In this paper, we implement a novel system for scalable semantic code clone detection. Specifically, we address two main challenges:

- *Challenge 1: AST has complex tree structure, which results in a high time overhead if only a simple tree matching algorithm is used to measure similarity. Then, how to design a lightweight model that can process the complex tree in a succinct way?*
- *Challenge 2: AST only contains the syntactic features of the code and lacks the semantic information to handle semantic code clones. Then, how to design an effective model that can handle semantic code clones?*

To tackle the first challenge, we build a novel triads model to represent the tree details of a complex AST. Specifically, we first add the control flow and data flow details of codes into the original AST to enrich the code semantics. After obtaining the *semantically enhanced AST* (SE-AST), we treat it as a social network and build a network-based triads model to process the tree structures. A type of triads describes a type of relationship among three nodes within a network, and it can well represent the nature of node's relationships. In our approach, we extract 10 types of triads and leverage them to complete the model building. By this, we can achieve scalable tree analysis while maintaining the program semantics.

To solve the second challenge, we compute the similarity scores of each type of triads and use them to train a code clone detector. Specifically, we first use the built triads model to divide all triads into different groups according to their corresponding node types. After completing the triads grouping, we then extract the similarity of all groups between two methods and leverage them to construct feature vectors. These vectors will be used to train a machine learning classifier for code clone detection. By this, we can achieve effective semantic code clone analysis.

We implement a prototype system namely *Tritor* and conduct comparative experiments with other nine state-of-the-art code clone detection systems on two widely used datasets, *Google Code Jam* (GCJ) [1] and *BigCloneBench* (BCB) [2, 50]. The nine code clone detection systems include two token-based methods (*i.e., SourcererCC* [47] and *RtvNN* [56]), four tree-based methods (*i.e., Deckard* [28], *ASTNN* [60], *TBCNN* [41], and *CDLH* [55]), and three graph-based methods (*i.e., SCDetector* [58], *DeepSim* [61], and *FCCA* [25]). Experimental results show that our system not only has good detection performance but also has ideal scalability. Although it takes more time than the token-based approach (*i.e., SourcererCC*), it is much faster than other tree-based and graph-based techniques. For example, compared to another recent state-of-the-art tree-based code clone detector (*i.e., ASTNN*), *Tritor* only requires 487 seconds to accomplish the analysis on one million code pairs while *ASTNN* consumes about 18,990 seconds.

Overall, our contributions to this paper are as follows:

- We extract the semantically enhanced AST to maintain the program details and build a novel social network-based triads model to represent the tree details.
- We implement a prototype system namely *Tritor* [9] by using the built model to construct feature vectors and train a semantic code clone detector.
- We evaluate *Tritor* with other nine tools on the GCJ and BCB datasets. The experimental results show that *Tritor* has great detection performance and strong scalability on semantic code clone analysis.

2 DEFINITIONS AND PRELIMINARY STUDY

2.1 Definitions

We first give some formal definitions of the terms we used in the paper.

2.1.1 Clone Types. Code cloning can be classified into four types according to the degree of similarity. In our paper, we use the following definitions of code cloning types [12, 45]:

- **Type-1 (textual similarity)**: Identical code fragments, except for different white-space, layouts, and comments.
- **Type-2 (lexical similarity)**: Identical code fragments, except for differences in identifier names and lexical values, in addition to the differences in Type-1 clones.
- **Type-3 (syntactic similarity)**: Syntactically similar code snippets that differ at the statement level. In addition to Type-1 and Type-2 clone differences, the fragments have statements added, modified, and/or removed with respect to each other.
- **Type-4 (semantically similarity)**: Syntactically dissimilar code fragments that implement the same functionality.

2.1.2 Triads. In the field of social networks, triad is a census algorithm for networks and represents the 16 relationships between three nodes in the absence of cyclic relationships, which can well represent the properties of the relationship between nodes [10]. Therefore, in our paper, Triad means triples of three nodes. As illustrated in Figure 1, according to the relationship between the three nodes, Triads are divided into 16 categories.

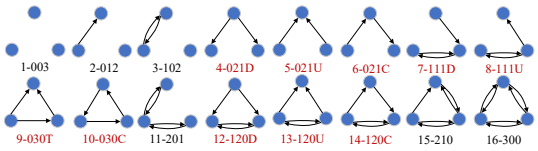


Figure 1: 16 categories of triads in a social network

2.1.3 Jaccard Similarity. The Jaccard similarity coefficient, also known as the Jaccard index, is used to compare the similarity between finite sample sets. Given two sets A and B, the Jaccard index is defined as the ratio of the size of the intersection of A and B to the size of the union set. It is calculated as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (1)$$

2.2 Preliminary Study

Triads are widely used in the analysis of social networks, and have a wide range of applications in fields like network structure analysis [11], population census [14], and social systems analysis [18].

For code clone detection, AST is a type of intermediate representation of code, and its structure can reflect the syntactic information of the code well. Since AST extraction does not require compilation, it is more lightweight and often used in code clone detection compared to *program dependency graph* (PDG) and *control flow graph* (CFG). Methods that implement different functionalities have different AST structures, while methods with the same functionalities have similar AST structures. However, no researchers have attempted to apply triads to the code clone detection field to analyse the structure of AST. Therefore, we do not know whether triads are suitable for code clone detection or not. To answer the question, we conduct a preliminary study to figure out whether triads can reflect the structural information of AST, resulting in a significant difference between clone pairs and non-clone pairs.

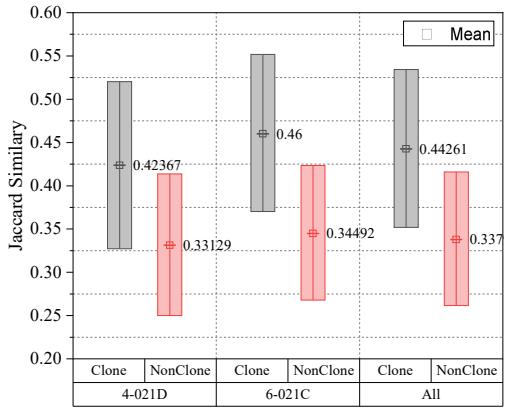


Figure 2: Jaccard similarity of clone pairs and non-clone pairs

Specifically, we perform experiments on 270,000 clone pairs and 270,000 non-clone pairs on the GCJ dataset. At first, we conduct static analysis to obtain the ASTs of all methods. The nodes in the AST are regarded as nodes of the social network, and the relationship between the three connected nodes is extracted to obtain triads. There are only two types of relationships between three nodes in AST since AST is a tree structure. One is that one node points to the other two nodes (*i.e.,* 4-021D in Figure 1), and the other is that one node points to the second node, which in turn points to the third node (*i.e.,* 6-021C in Figure 1). For a code pair, we compute the Jaccard similarity between the triads of two methods. First, we extract all the triads in the two methods separately and compute the concatenation and intersection of the two sets. The set consisting of triads shared by two methods is considered to be the intersection, and the set consisting of all triads occurring in both methods is considered to be the concatenation. Then we calculate the value of the intersection divided by the concatenation as the result of computing Jaccard similarity. After obtaining all similarities, we apply statistical analysis to study whether the similarity is higher for clone pairs and lower for non-clone pairs. In detail, we conduct experiments on similarity differences between clone and non-clone pairs in three cases: 4-021D, 6-021C, and all of these two types.

Through the results in Figure 2, we observe three findings:

- First, the similarity between clone and non-clone pairs has obvious differences, which indicates that triads can accurately reflect the structural information of AST, making higher similarities of clone pairs and lower similarities of non-clone pairs.
- Second, the average similarity of both clone and non-clone pairs is not high, neither exceeding 70%. So, we cannot directly extract triads and simply calculate the similarity based on the number to perform semantic code clone detection.
- Third, the discrepancy of the similarities between clone and non-clone pairs obtained by different types of triads differs.

According to these findings, we know that triads can reflect the differences between clone and non-clone code pairs, but cannot be directly used to detect semantic code clones. However, different types of triads have different similarities. If we can learn the differences between various categories of triads and find the most representative triad in detecting semantic clones, it will be a great candidate to use triads for semantic clone analysis. In this paper, we propose to use machine learning to learn the differences and design a novel triads-based semantic code clone detector.

3 SYSTEM

In this section, we present our proposed system namely *Tritor*.

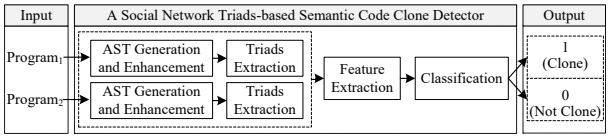


Figure 3: System architecture of *Tritor*

3.1 Overview

Figure 3 shows that *Tritor* consists of *AST Generation and Enhancement*, *Triads Extraction*, *Feature Extraction*, and *Classification*.

- ***AST Generation and Enhancement***: The purpose of this phase is to perform static analysis to extract the AST and add the control flow and data flow details to the AST to enrich the semantic information incorporated in the AST. The input of this phase is a method and the output is a SE-AST.
- ***Triads Extraction***: The purpose of this phase is to partition the SE-AST into different types of triads and group them according to the node types. The input is an SE-AST and the output is the number of various triads in each group.
- ***Feature Extraction***: The purpose of this phase is to extract the similarity scores of triads in the same group one by one. The input of this phase is the triads of two methods and the output is the similarity vector.
- ***Classification***: The purpose of this phase is to determine whether two methods are a clone based on the machine learning model trained in advance. The input of this phase is a similarity vector of two methods and the output is whether they are a clone pair.

3.2 AST Generation and Enhancement

In this paper, our objective is to detect semantic clones between programs. Therefore, we need to extract the semantic information of the program and use it as the basis for judging whether they are clones. AST is the common intermediate representation that contains syntactic information about the code. So, we perform static analysis on the method to obtain the ASTs of the methods. There are different static analysis tools for different programming languages

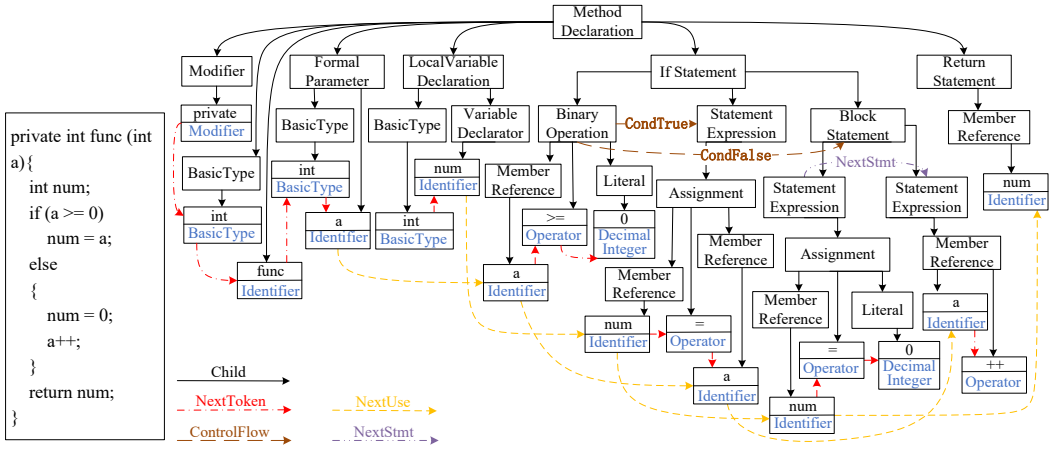


Figure 4: The semantically enhanced AST of the code on the left

that can be utilized. For example, if the programming language is Java, we can use *Javalang* [8] to extract the ASTs of the methods.

AST is rich in syntactic information but lacks semantic information. By adding data flow and control flow to the original AST, we can enrich the semantics contained in AST [54]. To this end, we adopt the method of flow-augmented abstract syntax tree in a recent work [54] to maintain the program semantics. However, we do not add all the edges but make trade-offs according to our needs.

For data flow, we add *NextToken* and *NextUse* edges in addition to the *Child* edges that are already in the AST. The *NextToken* edge concatenates the terminal nodes that belong to the same statement in token order, which reflects the sequence of tokens well. The *NextUse* edge connects the node where the variable is located to the next occurrence of that variable, which reflects the information of the data flow. We do not add the *Parent* and *NextSib* edges because the *Child* edge is sufficient to represent the relationship between parent and child nodes for our purposes. Moreover, the relationship between sibling nodes contained in the *NextSib* edge does not have great significance for the increase of semantic information. Therefore, we do not add these two edges.

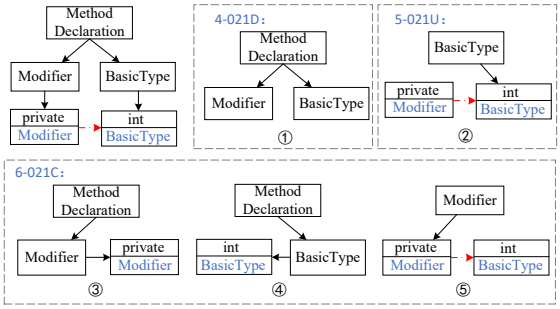


Figure 5: Categorize the triads of one subtree in Figure 4

For control flow, we add *Sequential Execution*, *If statements*, *While*, and *For* loops. *Sequential Execution* adds edges between the children nodes of a *BlockStatement*, pointing from the previous child node to the subsequent child node, to indicate the sequential execution of the statement. The *If* edge adds a *CondTrue* edge between the conditional statement and the statement when the condition is true, and a *CondFalse* edge between the conditional statement and the statement when the condition is false (if there is no else statement, it is not added). Both *While* and *For* edges add two bidirectional edges between the condition and the body. Figure 4 shows a graphical representation of the SE-AST corresponding to the code on the left that is generated by *javalang*.

Algorithm 1 Triads Extraction

Input: *graph*, the SE-AST to be analysed.
Output: *TriadsList*, all triads included in the SE-AST.

```
TriadsList ← []
for each node: v in graph do
    v_nbrs ← Pred_nodes(v) + Succ_nodes(v)
    for each node: u in v_nbrs do
        if id[u] > id[v] then
            u_nbrs ← Pred_nodes(u) + Succ_nodes(u)
            neighbors ← v_nbrs + u_nbrs
            for each node: w in neighbors do
                if id[u] < id[w] or (id[v] < id[w] < id[u] and v not
in Pred_nodes(w) and v not in Succ_nodes(w)) then
                    Add the (u, v, w) into the TriadsList
                end if
            end for
        end if
    end for
end for
```

3.3 Triads Extraction

After semantic enhancement of the AST, relationships other than parent-child relationships are found between nodes. In the preliminary study section, we mention that only two types of triads could be extracted from the AST: 4-021D and 6-021C. However, with the enrichment of node relationships in SE-AST (*i.e.,* the variety of edges increases), it is possible that triads types other than 4-021D and 6-021C could be presented in the SE-AST. In order to determine the types of triads used in the model, we calculate the SE-ASTs generated by all files in our open source projects, where the open source projects come from the top 10,000 java projects in GitHub in terms of criticality score. This score can be used to describe the impact and importance of an open source project [4]. Excluding the first three cases where there is no connection between the nodes, we find that there are a total of 10 categories of triads (*i.e.,* 4-021D, 5-021U, 6-021C, 7-111D, 8-111U, 9-030T, 10-030C, 12-120D, 13-120U, and 14-120C). They are marked in red in Figure 1. Therefore, we use these 10 types in our model. If we treat the resulting SE-AST as a network, the similarity between two networks can be measured by analyzing the relationships between nodes in the network.

Therefore, compared to the traditional graph matching method to compare the similarity of two graphs, we count all the triads appearing in the SE-AST and classify them.

The extraction of triads from SE-AST will be divided into three steps: First, as described in Algorithm 1, we traverse each node v in a SE-AST from top to bottom, finding the first-level neighbor nodes *v_nbrs* using the edges it connects as indices (line 3 in Algorithm 1), and then using these nodes in turn as criteria to find the second-level neighbor nodes *neighbors* (line 4-7 in Algorithm 1). In this way, we get all the triads from SE-AST. Line 5 and line 9 are designed to avoid duplicate searches by a criterion that only nodes with an id greater than the benchmark node can be used as neighboring nodes. Pred_nodes and Succ_nodes are methods to get the predecessor nodes and successor nodes of a node. For the same tree, the triads obtained by different traversal methods are the same, so using different traversal methods will not affect triads extraction.

Second, as similarity can only be calculated between triads of the same category, so we count all the triads that appear in each SE-AST and categorize them into one of the ten categories. As shown in Figure 5, the top left part of the picture is a part of Figure 4. For this subtree, we can get a total of five triads (*i.e.,* the part in the dashed box of Figure 5), and we can see that these five triads belong to three categories, namely 4-021D, 5-021U, and 6-021C.

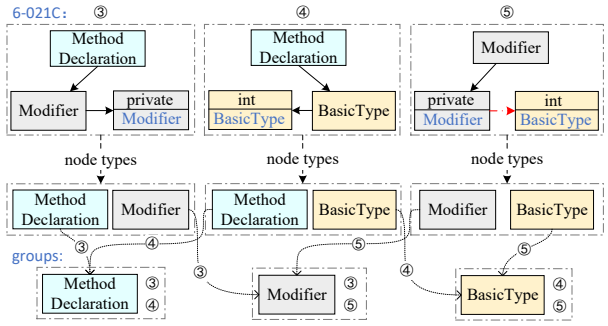
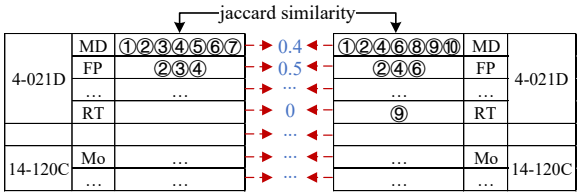


Figure 6: Group the triads of type 6-021C in Figure 5

Finally, different triads in each category may have completely different nodes. If we calculate the similarity between triads categories directly (as we did in our preliminary study), it may not give the ideal similarity score. Therefore, for each triads, they are grouped according to the types of the three nodes in the triads. As we can see in Figure 4, two kinds of nodes appear in the AST, one being non-leaf nodes and the other being leaf nodes. The types of non-leaf nodes can represent the code syntax of a method while the leaf nodes of an AST are tokens obtained by parsing the source code of a method. For the non-leaf nodes, to obtain a determinate result of node types, we extract the ASTs of all methods in our open source projects which consists of more than 300M lines of code to analyze the code syntax types from these trees. Eventually, we obtain a total of 57 code syntax types. For the leaf nodes (*i.e.,* tokens) in ASTs, we cannot extract an exact number. In order for these tokens to be assigned to a fixed group, we intend to represent them by their token types. For example, the token "int" is replaced by its type "*BasicType*". The blue texts in Figure 4 are leaf nodes represented by token types. We still choose the open source projects for the collection of token types as we did for the analysis of non-leaf nodes. After our statistical analysis, we find that 14 types appear in most of ASTs. In fact, the proportion of these 14 types accounts for more than 99.5% of all nodes. Therefore, we choose these 14 types as the final token types and add a *Null* type to represent other types. In final, we collect a total of 72 types which consists of 57 code syntax types and 15 token types.

For each triads, we group them according to the types of the three nodes in the triads. For example, the three triads of type 6-021C in Figure 5 can belong to three groups according to their node types: *MethodDeclaration*, *Modifier*, and *BasicType*. For the first triads of 6-021C in Figure 6, since the node types consist of *MethodDeclaration* and *Modifier*, it belongs to these two groups, respectively. Since there are a total of 72 node types, one type of triads will contain at most 72 groups. After these three steps, all triads contained in each SE-AST will be assigned to specific groups.



MD: MethodDeclaration; FP: FormalParameter; RT: ReferenceType; Mo: Modifier

Figure 7: Obtain the similarity vector of two methods

3.4 Feature Extraction

After obtaining the specific grouping information, we obtain their similarity vectors by sequentially calculating the Jaccard index of the groups corresponding to the two methods. As shown in Figure 7, after the previous steps, all triads in the SE-AST are divided into different categories according to triads types, and the triads in each category are divided into different groups according to the types of nodes. We then calculate the Jaccard index between the corresponding groups, and each group will output a similarity score. After collecting all similarity results, we concatenate them to construct a feature vector. After our statistical analysis on our open source projects, the number of groups contained in each type of triad can be obtained, with 4-021D containing 67 groups, 5-021U containing 71 groups, 6-021C containing 67 groups, 7-111D containing 18 groups, 8-111U containing 41 groups, 9-030T containing 49 groups, 10-030C containing two groups, 12-120D containing 19 groups, 13-120U containing seven groups, and 14-120C containing seven groups. As a result, the feature vector has a total of 338 dimensions.

3.5 Classification

Machine learning divides the knowledge structure by simulating the human learning method of the existing content, which effectively improves the learning efficiency. Since it can accurately and efficiently predict once trained, it has both accuracy and scalability [49]. In our paper, we choose certain widely used algorithms (*i.e., k-nearest neighbor* (KNN) [16], *random forest* (RF) [13], *decision tree* (DT) [43], *adaptiva boosting* (Adaboost) [20], and *gradient boosting decision tree* (GDBT) [21]) to commence our evaluations.

We perform feature extraction on all clone pairs and non-clone pairs to obtain similarity vectors. All the similarity vectors with labels are put into a machine learning model for training, and the obtained model is saved. Two methods to be detected are processed in the above three phases and their similarity vector will be fed into the model to obtain an output of zero (non-clone) or one (clone).

4 EXPERIMENTS

In this section, we discuss the following five questions:

- *RQ1: What is the detection effectiveness of Tritor when using different machine learning algorithms?*
- *RQ2: Can Tritor outperform other code clone detectors?*
- *RQ3: Does semantic enhancement on AST improve clone detection?*
- *RQ4: What is the runtime overhead of Tritor when detecting clones?*
- *RQ5: Why is Tritor effective in detecting semantic code clones?*

4.1 Experimental Settings

4.1.1 Datasets. Similar to previous work, we conduct experiments on two datasets: GCJ and BCB. The programs in the GCJ dataset [61] are derived from an online programming competition held by Google and contain 1,669 projects from 12 different competition problems which are written by different programmers. So projects of the same competing problem are almost syntactically different but semantically similar, and we treat them as clone pairs. Projects that solve different problems are not similar, and we regard them as non-clone pairs. As a result, we obtain 275,570 semantic clone pairs and 1,116,376 non-clone pairs. We randomly select 270,000 pairs from all non-clone pairs to balance our dataset.

The second dataset is the popular large code clone benchmark BCB dataset [2], which contains over eight million labeled clone pairs from 25,000 systems. The reason why we choose the BCB dataset is that the code granularity of their clone pairs is function-level, which is in line with the detection granularity of *Tritor*. Moreover, the clone pairs in BCB are assigned different clone types to facilitate our observation of the effectiveness in detecting different types of clones. However, due to the unclear boundary between Type-3 and Type-4, it is further divided into three subclasses by similarity scores measured by line-level and token-level as follows: i) *Strongly Type-3* (ST3) with the similarity between 70-100%, ii) *Moderately Type-3* (MT3) with the similarity between 50-70%, and iii) *Weakly Type-3/Type-4* (WT3/T4) with the similarity between 0-50%. We randomly select 270,000 clone pairs from the eight million clone pairs since the number of non-clone code pairs is 270,000. The clone pairs we select include 48,116 clone pairs of *Type-1* (T1), 4,234 clone pairs of *Type-2* (T2), 21,395 clone pairs of ST3, 86,341 clone pairs of MT3, and 109,914 clone pairs of WT3/T4.

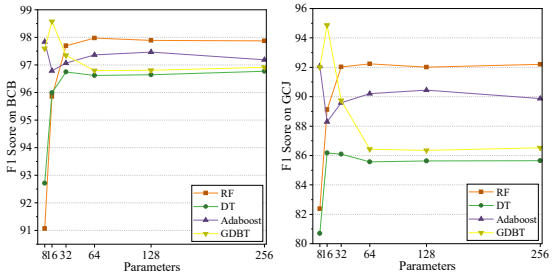


Figure 8: The F1 scores by using different depth parameters of different machine learning algorithms

4.1.2 Implementations. We use *Javalang* [8] to obtain AST in the AST generation and enhancement phase as the programming language of our dataset is Java. In the classification phase, we use *Sklearn* [7] to implement KNN, RF, DT, Adaboost, and GDBT classification algorithms. We run all experiments on a server with 8 cores of CPU and a GTX 1080 GPU. For recording the experimental effects, we adopt ten-fold cross-validations for training and validation.

4.1.3 Comparative Systems. In order to make our evaluation more comprehensive, we select some representative work from a large number of code clone detection tools to conduct comparative experiments. Specifically, we select two token-based code clone detectors (*i.e., SourcererCC* [47] and *RtvNN* [56]), four tree-based methods (*i.e., Deckard* [28], *ASTNN* [60], *TBCNN* [41], and *CDLH* [55]), and three graph-based methods (*i.e., SCDetector* [58], *DeepSim* [61], and *FCCA* [25]). *SourcererCC* [47] is a popular token-based code clone detector which can scale to big code. *RtvNN* [56] is a popular RNN-based code clone detector which encodes source code tokens and ASTs. *Deckard* [28] is a popular AST-based code clone detector which clusters the vectors of AST subtree. *ASTNN* [60] is a popular AST-based code clone detector which splits a large tree into certain statement trees and trains an RNN model to detect code clones. *TBCNN* [41] is a popular AST-based clone detection detector by using a convolutional neural network. *CDLH* [55] is a popular AST-based clone detection tool with a long short-term memory network. *SCDetector* [58] is a popular graph-based code clone detector which extracts the control flow graph of a method and applies centrality analysis to detect code clones. *DeepSim* [61] is an advanced graph-based clone detection tool by using a deep neural network. *FCCA* [25] is an advanced graph-based clone detection tool by using hybrid code representations with high accuracy. For the parameter settings [9] of these tools, we select the parameters reported in their published papers since they can perform best with these parameters.

4.1.4 Metrics. To measure the detection effectiveness of all detectors, we adopt widely used metrics such as *Precision* (P), *Recall* (R), and *F-measure* (F1). These metrics are described on our website [9].

4.2 RQ1: Comparison of Different Methods

To achieve higher precision and recall in detecting semantic clones, we measure the classification performance of different machine learning algorithms and select the best one for subsequent experiments. We choose five commonly used machine learning algorithms (*i.e.,* KNN, RF, DT, Adaboost, and GDBT) for training and testing. They are popular algorithms that are often used in classification problems and can achieve good results in previous work [24, 57]. For KNN, there has been a lot of work demonstrating that neighbor parameter selects one and three are the most widely used and can achieve good results [24, 57]. Therefore, we first select the parameters that allow other machine learning models to obtain the best results. The F1 scores achieved by RF, DT, Adaboost, and GBDT machine learning algorithms on the BCB and GCJ datasets with different depth parameters are presented in Figure 8. We can see that RF, Adaboost, and GDBT achieve the highest F1 scores with depth parameters of 64, 8, and 16, respectively. However, DT does not have the same parameter value for achieving the highest F1 score on both datasets. Since DT is more stable when the parameter is 32, we choose 32 as the parameter for DT. Therefore, the neighbor parameters of KNN are selected as one and three because they are the most commonly used. The depth parameter of RF, DT, Adaboost, and GDBT is selected as 64, 32, 8, and 16 respectively, as they can achieve the highest F1 scores.

Observing the F1 score, precision, and recall shown in Figure 9, we see that the GDBT algorithm has the best results on the BCB

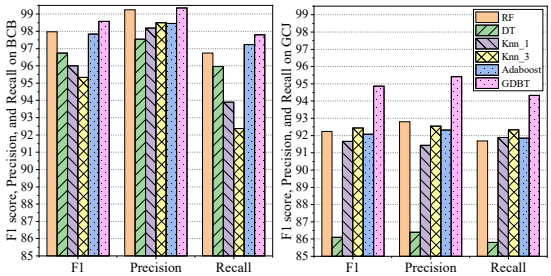


Figure 9: F1 score, Precision, and Recall of *Tritor* using different machine learning classification methods

dataset, with RF algorithm being the next best. For example, the GDBT algorithm and RF algorithm have an F1 score of 98.57% and 97.97%, while the other four scores are 96.74%, 96.00%, 95.33%, and 97.84% on the BCB dataset. However the detection effectiveness of all models decreases on GCJ dataset, some algorithms such as DT decrease even more. This is due to the fact that DT does not handle more complex semantic clone classification very well. As an improvement to DT, RF algorithm is relatively outstanding. Because RF consists of multiple decision trees actually. A test sample can obtain the most probable classification among the classification results of each decision tree in the random forest. GDBT has the best results, but the time overhead of GDBT is much higher than that of RF, for instance, 19.8 times higher on BCB. So we choose the RF algorithm for the subsequent experiments.

Summary: *The difference in F1 scores between GDBT and RF is small, but the time overhead of GDBT is much higher than that of RF, so we choose the RF algorithm for the subsequent experiments.*

4.3 RQ2: Overall Effectiveness

4.3.1 Results on Google Code Jam. First, we conduct experiments on the GCJ dataset to evaluate the effectiveness of *Tritor*. As we mentioned earlier, clone pairs on the GCJ dataset are naturally semantically similar, and almost unlikely to be syntactically similar. Therefore, we treat all similar pairs as semantic clones (*i.e.,* Type-4 clone) and run experiments on these pairs to evaluate the effectiveness of *Tritor* in detecting semantic clones. Table 1 presents the detection results of nine comparative systems and *Tritor*.

Table 1: Results of clone detection on GCJ and BCB datasets

Group	Method	GCJ			BCB		
		R	P	F1	R	P	F1
Token-based	SourcererCC	0.11	0.43	0.17	0.07	0.98	0.14
	RtvNN	0.90	0.20	0.33	0.01	0.95	0.01
Graph-based	SCDetector	0.87	0.81	0.82	0.92	0.97	0.94
	DeepSim	0.82	0.71	0.76	**0.98**	0.97	0.98
	FCCA	0.90	**0.95**	0.92	0.92	0.98	0.95
Tree-based	Deckard	0.44	0.45	0.44	0.06	0.93	0.12
	ASTNN	0.87	0.95	0.91	0.94	0.92	0.93
	TBCNN	0.89	0.91	0.90	0.81	0.90	0.85
	CDLH	0.70	0.46	0.55	0.74	0.92	0.82
Our tool	Tritor	**0.92**	0.93	**0.92**	0.97	**0.99**	**0.98**

For token-based approaches: *SourcererCC* has both low recall and precision. This is because *SourcererCC* is a token-based code clone detector, which only considers the overlapping similarity of tokens between two methods (*i.e.,* the ratio of the number of identical tokens shared by two methods to the maximum number of tokens of the two methods) and lacks the consideration of any semantic information, so it does not perform well on a dataset with all semantic clones like GCJ. The recall of *RtvNN* is high, but the precision is low. This is because *RtvNN* computes a simple Euclidean distance metric for only the tokens and ASTs of a code by using recurrent neural networks to measure the similarity of code pairs. However, the distances between most pairs computed by this method do not differ significantly (*i.e.,* only between 2.0 to 2.8). Changing the threshold can increase the precision but decrease the recall. Therefore, *RtvNN* cannot have a high F1 score.

For tree-based approaches: Compared to other tree-based methods, the detection results of *Deckard* and *CDLH* are not satisfactory. This is because *Deckard* uses vectors to carry syntactic information in the parse tree, it finds the vectors' nearest neighbors by clustering, which requires the feature vectors at the root of the parse tree to be very close. However, most code clone pairs have different parser tree structures, leading a poor precision and recall on the GCJ dataset. *CDLH* learns hash functions, structural information, and code fragments by using an AST-based long short-term memory network. But the representations are all lexical and syntactic, leading to low detection performance. The other two tree-based methods *ASTNN* and *TBCNN* have a comparatively good capability of detecting semantic clones. *ASTNN* splits each large AST into lots of small sentence trees and encodes these sentence trees as vectors. Then *ASTNN* selectively stores more important node information by using BiGRU and RvNN encoders, so ASTNN has a high precision. But the segmentation of the AST may lead to some semantics loss, which in turn results in a relatively low recall. *TBCNN* has good detection results because it captures the structural features of the AST very well by sliding the convolutional kernels. However, one weakness of the convolutional layer is that it cannot capture long-range contextual information. In this way, if the AST is deep or has many nodes, the operation of converting the AST into a binary tree exacerbates the problem of a long-term dependence on the original semantics of the source code.

For graph-based approaches: *SCDetector* converts the CFG of the method into semantic tokens with graph details and then feeds these semantic tokens into a *Siamese* network to train a model to detect code clone pairs, so *SCDetector* is good at detecting semantic clones. *DeepSim* uses a deep learning model to learn binary matrix abstracted from the variables, basic blocks of CFG, and the relationships between them. As these representations contain code semantics, *DeepSim* is proficient at detecting semantic clones. *FCCA* feeds the comprehensive hybrid code representation into a deep learning model with an attention mechanism. The combination of structured representations (*i.e.,* tokens) and unstructured representations (*i.e.,* AST and CFG) enables *FCCA* to detect most semantic code clones with a high degree of precision.

4.3.2 Results on BigCloneBench. In this subsection, we analyze the effectiveness of detecting all types of clone pairs on the BCB dataset and make a comparison with our comparative tools. Table 1 presents the detection results.

As can be seen from the performance of the results shown in Table 1, *Tritor* outperforms most of the other detectors in precision and F1 score, indicating that *Tritor* is also good at detecting code clones on BCB dataset. Different from the results on GCJ, *SourcererCC*, *Deckard*, and *RtvNN* all have high precision and low recall.

This is because these tools can only detect code clones that are textually or syntactically similar. Thus, they are only able to detect syntactically similar clone pairs (*i.e.,* T1 and T2) in BCB, but not semantic similarity. Moreover, we also observe that the detection results on BCB are almost always better than those on GCJ. This is because the clone pairs in the BCB dataset are constructed by experts deliberately. Apart from minor differences, the code structures are very similar and can be easily detected. On the other hand, the clone pairs in the GCJ dataset are answers given by different programmers to the same competition question and have a completely different code structure, which makes them difficult to detect.

Table 2: F1 for each clone type on BCB

Group	Method	T1	T2	ST3	MT3	T4
Token-based	SourcererCC	1.00	1.00	0.65	0.20	0.02
	RtvNN	1.00	0.97	0.6	0.03	0.00
Graph-based	SCDetector	1.00	1.00	0.97	0.97	0.94
	DeepSim	0.99	0.99	0.99	0.98	0.95
	FCCA	1.00	1.00	0.99	0.97	0.95
Tree-based	Deckard	0.73	0.71	0.54	0.21	0.02
	ASTNN	1.00	1.00	0.99	0.98	0.92
	TBCNN	1.00	1.00	0.93	0.80	0.86
	CDLH	1.00	1.00	0.94	0.88	0.82
Our tool	Tritor	**1.00**	**1.00**	**1.00**	**0.99**	**0.95**

Next, we analyze how the clone detector performs in detecting each of the five types of clones and compare it to the advanced code clone detection technique in terms of F1 scores. We select the number of clone pairs and non-clone pairs for each type as described in the experimental dataset in subsection 4.1. Table 2 shows the F1 scores for *Tritor* and nine comparative systems for detecting five types of code clones. We can see that *Tritor* is superior to other clone detectors in detecting each type of code clones. Particularly when detecting WT3/T4, the F1 scores of *SourcererCC*, *RtvNN*, *Deckard*, *TBCNN*, and *CDLH* are 2%, 0%, 2%, 86%, and 82% respectively, while *Tritor* can reach an F1 score of 95%. This demonstrates that *Tritor* can detect semantic clones comprehensively and precisely. *SCDetector*, *DeepSim*, *FCCA*, and *ASTNN* also perform well in detecting WT3/T4 clones. However, they all require GPUs to complete the complex deep neural network training. For *Tritor*, we only need CPU to train simple machine learning models for classification, which means that *Tritor* requires less computational resources than *SCDetector*, *DeepSim*, *FCCA*, and *ASTNN*.

The reason for *Tritor*'s good ability to detect semantic clones lies in three aspects. First, the semantically enhanced AST contains more semantic information, thus enhancing the *Tritor*'s ability to detect semantic clones. Second, we regard the enhanced AST as a social network, and measure the similarity between two methods by analyzing the relationship among three nodes in the network, avoiding the high overheads associated with tree matching while preserving the details of the tree structure. Thirdly, instead of detecting code clones directly by threshold after obtaining similarity, our method puts the similarity vector into a machine learning classifier for clone detection. The use of machine learning algorithms allows our approach to be highly accurate and scalable.

Summary: *Compared to most code clone detectors, Tritor performs well in detecting code clones on both GCJ dataset and BCB dataset, especially in detecting semantic clones. Moreover, Tritor only needs CPU for classification, which means that Tritor has a much faster speed than those graph-based approaches.*

4.4 RQ3: The Significance of SE-AST

To check whether the augmented data flow and control flow on AST can contribute to *Tritor* or not, we perform an ablation experiment. In the experiment, we perform the same feature extraction operation on both the original AST and the SE-AST with the added data flow and control flow, then they are both fed into the RF machine learning algorithm for training and testing. We record their respective detection results in Table 3.

Table 3: Recall, Precision, and F1 of original AST and semantically enhanced AST in detecting clones

Dataset	AST	R	P	F1
GCJ	SE-AST	0.92	0.93	0.92
	Original AST	0.89	0.90	0.90
BCB_ALL	SE-AST	0.97	0.99	0.98
	Original AST	0.96	0.98	0.97
BCB_WT3/T4	SE-AST	0.92	0.99	0.95
	Original AST	0.89	0.98	0.93

Table 3 illustrates the recall, precision, and F1 scores on the two datasets, respectively. For BCB dataset, we not only record the comparative results when analyzing the whole dataset (*i.e.,* BCB_ALL), but also collect the results when detecting semantic code clones (*i.e.,* BCB_WT3/T4). It can be seen that SE-AST has better results in detecting clones on both BCB and GCJ datasets. For example, on the GCJ dataset, when we use the original AST for clone detection, the F1 score is only 90%. After enhancing the AST, the F1 score can increase to 92%. The main reason for this improvement is that ASTs contain mainly syntactic information about the program, and the added data flow and control flow enrich the semantic information of the AST to a large extent. Since semantic clones are almost syntactically different, their ASTs are likely to be different. The semantically enhanced AST can contain the same semantic information in the Type-4 clones, which is more similar. The second reason is that there are only two kinds of triads 4-021D and 6-021C in Figure 1 in the original AST, while the semantically enhanced AST contains 10 kinds of triads. The increase in the types of triads also allows the similarity of the two graphs to be measured from more perspectives. The response in the dimension of the feature is that the original AST obtains an 112-dimensional feature vector, while the semantically enhanced AST obtains a 338-dimensional feature vector. The increase of this dimension can measure similarity from more angles, which improves the effect of clone detection to a certain extent.

Summary: *SE-AST has better results in detecting clones on both BCB and GCJ datasets. The two reasons for this improvement are that the added edges enrich the semantic information of the AST to a large extent and the increase in the types of triads also allows the similarity of the two graphs to be measured from more perspectives.*

4.5 RQ4: Scalability

In this subsection, we run all experiments to compare the running overhead of *Tritor* on a server with an 8 cores CPU and a GTX 1080 GPU to test its scalability. Similar to previous work [24, 47, 57, 58, 62], we randomly select one million code pairs from the GCJ dataset for the experiment. A total of 10 random selections are made, recording the time overhead each time and using the average

as the final time overhead. The time overhead for each method is presented in Table 4, including training time and prediction time. For *Tritor*, the training time consists of the processing time for the preliminary steps and the time to train the model.

Table 4: Runtime on analyzing one million code pairs

Group	Method	Training	Prediction
Token-based	SourcererCC	-	16s
	RtvNN	5,206s	35s
Graph-based	SCDetector	2,937s	139s
	DeepSim	13,545s	34s
	FCCA	56,769s	91s
Tree-based	Deckard	-	72s
	ASTNN	16,096s	2,894s
	TBCNN	41,168s	86s
	CDLH	45,317s	90s
Our tool	Tritor	467s	20s

For clone detectors that are not based on deep learning algorithms (*i.e., SourcererCC* and *Deckard*), their prediction time is zero, and the prediction time is the time for all processes to detect clones. For other seven comparative systems, their training phases require GPUs as they are deep learning-based methods. However, even though they have GPUs to accelerate the training phase and the testing phase, their time overheads are still higher than that of *Tritor* using only CPUs. Such results demonstrate that *Tritor* is more scalable than *RtvNN*, *SCDetector*, *DeepSim*, *FCCA*, *ASTNN*, *TBCNN*, and *CDLH*. For *ASTNN*, which is also tree-based and has good capability for code clone detection, it takes a total of 18,990 seconds (16,096 seconds for training and 2,894 seconds for testing) to complete the entire training and testing phase. While *Tritor* consumes only 487 seconds (467 seconds for training and 20 seconds for testing) to complete the whole procedure. Overall, *Tritor* is about 39 (*i.e.,* $(16,096 + 2,894)/(467 + 20) = 38.99$) times faster than *ASTNN*.

Summary: *Tritor spends more time than SourcererCC because of the consideration of AST construction. However, because of social network-based triads model and the use of a machine learning algorithm, Tritor is more scalable than RtvNN, SCDetector, DeepSim, FCCA, ASTNN, TBCNN, and CDLH.*

4.6 RQ5: Interpretability

To explore why *Tritor* is effective in detecting semantic clones, we extract the importance of each feature of the similarity vector. Due to the interpretability of the RF algorithm, we extract the weight of each feature in the vector and sort them according to the weight, so that we can clarify which features are more important in detecting semantic code clones. Due to space limitations, we only show the top 20 features in the 338-dimensional vector obtained on the GCJ dataset in Table 5.

By observing the ranking of feature importance, we find two obvious phenomenons: The first is that different types of triads have different levels of importance. According to our observations, only Type 6-021C, Type 5-021U, Type 4-021D, Type 8-111U, and Type 9-030T appear in the top 20 features in terms of importance, and the number of occurrences is uneven. For example, Type 6-021C appears most frequently (ten times), followed by Type 5-021U, which appears four times. This phenomenon suggests that the structure of these five types of triads is more important for preserving program semantics in SE-AST. This is because the triads of Type 6-021C and Type 4-021D are typical structures that already exist in the AST, these two structures support the entire framework of the AST and therefore carry a large number of program details contained in the AST. The triads of Type 5-021U usually contain additional data flow edges. DecimalInteger_5-021U and Operator_5-021U, which occur in Table 5, are usually the type of leaf node and contain additional *NextToken* edges, indicating that the addition of data flow is significant in detecting semantic clones. The triads of Type 8-111U contain the two bidirectional edges we add between the conditions and the body of *While* and *For* loops, and the triads of Type 9-030T usually contain the edges we add between the subnodes of a *BlockStatement* node to indicate the sequential execution of the statement. The high importance of these two types of triads indicates that the addition of edges to these two types of control flow plays an important role in detecting semantic clones.

Table 5: Top 20 features of *Tritor* in detecting semantic clones

R	Feature Name	W%	R	Feature Name	W%
1	BinaryOperation_6	3.31	11	Operator_5	1.81
2	DecimalInteger_6	2.97	12	MemberReference_4	1.79
3	Literal_6	2.97	13	MemberReference_5	1.77
4	Operator_6	2.63	14	BlockStatement_9	1.70
5	DecimalInteger_5	2.44	15	BinaryOperation_8	1.69
6	MemberReference_6	2.43	16	ForControl_4	1.68
7	StatementExpression_6	2.27	17	ForStatement_6	1.66
8	BlockStatement_6	2.11	18	ForStatement_8	1.65
9	BasicType_6	2.07	19	ForControl_6	1.59
10	BinaryOperation_4	1.94	20	BasicType_5	1.58

The second phenomenon is that three node types are more important among the top 20 weighted features, with *BinaryOperation* and *MemberReference* appearing three times each, and *DecimalInteger* appearing twice in the top five features. This suggests that these three node types are particularly important for detecting semantic code clones because they can well reflect the semantic information of the code. *BinaryOperation* represents a binary operation (*e.g.,* "`i>=0`"), which is usually found in conditional judgments such as *If* statements, *While* and *For* conditional judgment statements. The functionalities implemented by these three statements are important for the embodiment of semantics in the methods, and therefore *BinaryOperation* has a high importance. *MemberReference* is usually linked to the *Identifier* node type. It is associated with the use of variables and carries information about the data flow to a large extent. Therefore, it plays an important role in detecting code clones. *DecimalInteger* is a kind of constant. It is possible to be in the position of the leaf node, so it will be related to the data flow. It may also appear in statements such as assignment statements or conditional judgment statements which are related to control flow. Therefore, having both data flow and control flow information is extremely important for detecting semantic clones, allowing *DecimalInteger* to appear twice in the top five ranking of importance.

To give a more visual representation of the effectiveness of these features, we reorder the similarity vectors according to the importance of the features from highest to lowest. Then we sequentially take the top n (n from 1 to 338) features, and record the F1 scores for vectors of different lengths in Figure 10. It can be seen that when only one feature of highest importance is used for classification, the F1 score can reach 0.68. As the number of features continues to increase, the detection effect is getting better and better, and when the number of features reaches around 100 (less than 1/3), the

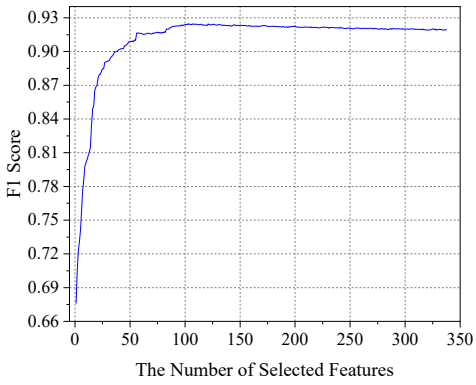


Figure 10: F1 scores of *Tritor* when selecting different numbers of features

F1 score can already be maintained at a relatively high level. This phenomenon indicates that the first 100 features can well meet the needs of clone detection. In the future, we plan to apply different feature selection algorithms to select the most important features to improve scalability without compromising accuracy.

Similar to the distribution of the top 20 weighted features, we find that the distribution of the top 100 weighted features is also consistent with the two phenomena described above. First, the importance of triads types is also in accordance with that in the top 20 in terms of weight. Type 6-021C and Type 4-021D appear most frequently in the top 100 weighted features with 30 occurrences. Type 5-021U, Type 8-111U, and Type 9-030T appear 13, 11, and 12 times, respectively, which is relatively balanced. Moreover, we also find Type 12-120D, which does not appear in the top 20 weighted features, appearing four times among the top 100 weighted features. Similar to Type 8-111U, the structure of Type 12-120D comes from the addition of two bidirectional edges between the condition and the body of *While* and *For* loops. The reason that Type 12-120D is not as important as Type 8-111U is that the structure of Type 12-120D only includes the parent node that points to both the condition node and the body node, but not the subsequent nodes of them. As a result, the semantic information in the subsequent nodes is not included in the Type 12-120D, so the Type 12-120D contains relatively little semantic information and is less important.

Second, the importance of node type is also consistent with the top 20 weighted features. *BinaryOperation* and *MemberReference* continue to be of high importance with five occurrences, respectively. Besides them, *ForStatement* and *BlockStatement* also appear five times, respectively. These two node types also appear quite frequently in the top 20 weighted features. This is because *ForStatement* embodies control flow information, and the nodes and edges it connects to reflect semantic information explicitly. *BlockStatement* adds sequential execution edges between its child nodes, also reflecting control flow information and enhancing semantic entailment. In contrast to the fact that *DecimalInteger* appears twice in the top five features, *DecimalInteger* appears only three times in the top 100 weighted features, not continuing its previous importance. We suspect that this is because *DecimalInteger* is usually a leaf node, and leaf nodes are the edges of the tree, which are inherently limited in frequency of occurrence. As a result, *DecimalInteger* does not appear frequently in the top 100 weighted features.

Summary: *According to the features' importance, the top 100 important features can well meet the needs of clone detection. Some specific types of triads and nodes have a higher degree of importance. This phenomenon applies commonly to features in the top 20 in importance as well as to features in the top 100 in importance.*

5 DISCUSSIONS

5.1 Threats to Validity

The first threat comes from the dataset. The code pairs in the BCB dataset are constructed by experts deliberately. Apart from minor differences, the code structures are very similar and can be easily detected. It would be biased if we only use the results on the BCB dataset to represent the effectiveness of the detection of the whole open source project. To alleviate the impact, we add experiments on the GCJ dataset, which contains 1,669 projects from 12 different competition problems that are written by different programmers.

The second threat comes from the token type. A total of 15 token types were chosen to allow tokens to be grouped into fixed groups. If the number of token types parsed by *Javalang* is not clear, the selection of these types may be variable and lead to inaccurate groupings. To alleviate the impact, we analyze all the token types in the BCB dataset and the GCJ dataset to select the token types that occur frequently and add a *Null* type to represent the remaining types that occur rarely.

The third threat comes from the time overheads. When calculating the time overhead of *Tritor*, we cannot obtain absolutely accurate data due to the different machine statuses, such as CPU usage. To alleviate the impact of this threat, we evaluate our tool ten times and report the average runtime overhead in the paper.

The fourth threat comes from the interpretability. All analyses in RQ5 are run on the GCJ dataset and to alleviate the impact, we add experiments on the BCB dataset. The results show that the distribution of both triads types and node types in the top 100 important features are generally consistent with the results of the experiments on GCJ. Only some minor deviations are due to differences in code between datasets. The detailed data comparison has been placed on our website [9] for page limitation reasons.

The fifth threat comes from the ratios of test sets. The project codes to be detected in real world do not have a balanced test set (*i.e.,* equal number of cloned pairs and non-cloned pairs). Biases may exist by using different proportions of the test set. To alleviate the impact of this threat, we perform experiments on seven test sets with different ratios of clone and non-clone pairs to investigate the sensitivity of *Tritor*. The results show that as the proportion of non-clone pairs increases, the precision and F1 scores become progressively lower, recall remains a relatively stable score. But even so, when the ratio of clone to non-clone pairs is 1:4, *Tritor* is still able to achieve a high F1 score of 96.92% with only 1% fluctuation, demonstrating that *Tritor* still has stable performance against variations in the ratio of the test set. Due to space constraints, the results of the experiment are not presented in the paper and can be found on our website [9].

5.2 Limitation and Future Work

In our paper, we mainly focus on code clone detection in the Java language. However, with a little modification, our method can be extended to other programming languages. For example, in the AST extraction phase, we can use *pycparser* [3] or *joern* [6] to

extract AST for C source code. We can count the types of nodes and triads, complete the extraction of triads and then extract similarity features to detect code clones in C source code. In the future work, our method will be extended to C code to detect code clones.

In this paper, we use Jaccard similarity to obtain the feature vectors of the two methods and use the random forest algorithm to train the classifier. In our future work, we will try other similarity calculation methods and other machine learning algorithms to achieve better detection results. For the comparison system, since many of the advanced tools are not open source, we only select nine open source tools for comparison. In the future work, we will select more tools for more intensive comparison.

Furthermore, we can observe from Figure 10 that using only the top 100 features in terms of importance is sufficient to achieve a high level of effectiveness in detecting semantic code clones. The detection is slightly reduced again due to the interference caused by the inclusion of less important features. Therefore, in our future work, we may use different feature selection techniques to find the most suitable combination of these features to make *Tritor* more scalable and effective for semantic code clone detection.

In addition, to ensure the comprehensiveness of our experiments, we choose five common machine learning algorithms. These machine learning models have many parameters, leading to a wide range of parameter combinations that are challenging to comprehensively cover in the experiments. Among them, the depth parameter has attracted considerable attention. Therefore, we focus on assessing the influence of depth parameters on detection performance. In the future work, we plan to conduct testing of other parameters to explore the optimal parameter combinations that yield better results. In addition, it is important to note that the experiments for parameter selection are conducted on the BCB and GCJ datasets. However, we are uncertain about the performance of these parameters on other datasets. In the future work, we intend to conduct testing on diverse datasets to select parameter configurations that are more suitable and applicable.

Finally, *Tritor* may not be superior to some baselines, such as *Tailor* [39] and *FA-AST* [54]. Due to the fact that our model lacks some automatic extraction of semantic information compared to *graph neural network* (GNN), our results do not surpass them. However, the results of our tool are sufficient to outperform the majority of semantic clone detection methods. Furthermore, as our tool is based on machine learning, it does not require GPU and supports interpretability. In contrast, neither *Tailor* nor *FA-AST* support interpretation and also require a GPU to train and test. Therefore, the advantage of *Tritor* is higher scalability and interpretability. In the future, we intend to use ensemble learning approach [5] to improve the effectiveness of our tools, striving to achieve higher results.

6 RELATED WORK

In this section, we introduce the related works of code clone detection techniques. Among them, text-based and token-based methods are mostly scalable, while tree-based and graph-based tools are mostly capable of detecting semantic clones.

Text-based and token-based approaches require little runtime overhead and can be extended to large-scale clone detection as they do not involve much analysis of the source code. For text-based clone detection methods [19, 27, 30, 33, 34, 44, 46, 59], the core idea of them is to treat the code as normal text to compare the similarity. Ducasse et al. use a string matching method to calculate the similarity of code lines [19]. Roy et al. use a similarity calculation method of the longest common subsequence to detect clones [46]. The token-based approaches [22, 23, 26, 32, 37, 47, 53] perform lexical analysis of source code to obtain tokens, and detect clones by finding common tokens. *SourcererCC* [47] detects clones by calculating the proportion of overlapping tokens. The text-based and token-based methods rarely consider the program semantics and the logic of the code fragment, as a result, these methods do not have the ability to detect semantic clones.

Code clone detectors which can detect semantic clones are essentially tree-based and graph-based. These methods detect clones by analysing intermediate representations (*e.g.,* PDGs, CFGs, and ASTs) with semantic information. The tree-based clone detection techniques [15, 28, 29, 38, 42, 55, 60] perform static analysis of source code to extract parse trees or abstract syntax trees, then use tree matching to detect similar tree structures and thereby detect clones. *Deckard* [28] clusters the similar vectors obtained from AST using locality sensitive hashing to detect clones for any language with grammatical regulations. *CDLH* [55] normalizes the AST into a binary tree before encoding the tree representations to vectors using Tree-LSTM [51]. *ASTNN* [60] encodes each subtree that is divided according to predefined rules into vectors and integrates these vectors into a final vector using the bidirectional RNN model. The graph-based clone detection techniques [35, 36, 52, 58, 61, 62] represent the programs to graph representations, such as CFG and PDG. Most techniques use subgraph matching to detect clones (*e.g.,* [35] and [36]), but the subgraph matching usually spends a lot of time to detect, so *CCSharp* [52] reduces the overhead by modifying the graph structure and filtering the feature vectors. *CCGraph* [62] uses some numerical features to categorize PDG and compute the similarity of the strings to reduce the runtime overhead.

Compared with these methods, our method regards the semantically enhanced AST as a social network and extracts similarity features by analyzing the different relationships among three nodes in the network. The use of triads and the machine learning model makes our code clone detector has both accuracy and scalability.

7 CONCLUSION

In this paper, we propose a scalable semantic code clone detector based on semantically enhanced AST. To avoid the high overhead of tree matching, we regard the semantically enhanced AST as a social network and build a novel triads model to represent the tree details. We design a code clone detector (*i.e., Tritor*) with accuracy and scalability by training a machine learning classifier. We evaluate *Tritor* and other nine state-of-the-art code clone detection systems on GCJ [1] and BCB [2, 50] datasets. The experimental results show that *Tritor* has ideal detection performance and strong scalability. In analyzing code clones, *Tritor* is about 39 times faster than another current state-of-the-art AST-based code clone detector *ASTNN*.

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 the National Science Foundation of China under Grant No. U1936211 and Hubei Key Project under Grant No. 2023BAA024.

REFERENCES

[1] 2017. Google Code Jam. https://code.google.com/codejam/past-contests.
[2] 2023. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[3] 2023. A complete parser of the C language. (Pycparser). https://pypi.python.org/pypi/pycparser/.
[4] 2023. Criticality score. https://github.com/ossf/criticality_score.
[5] 2023. Ensemble Learning. https://en.wikipedia.org/wiki/Ensemble_learning.
[6] 2023. Joern is a platform for robust analysis of source code, bytecode, and binary code. (Joern). https://joern.io/.
[7] 2023. An open source machine learning library that supports supervised and unsupervised learning. (Scikit-learn). https://scikit-learn.org/stable/.
[8] 2023. A pure Python library for working with Java source code, provies a lexer and parser targeting Java 8. (Javalang). https://pypi.org/project/javalang/.
[9] 2023. Tritor. https://github.com/CGCL-codes/Tritor.
[10] Vladimir Batagelj and Andrej Mrvar. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks* 23, 3 (2001), 237–243.
[11] Vladimir Batagelj and Andrej Mrvar. 2004. *Pajek — Analysis and Visualization of Large Networks.* Springer Berlin Heidelberg, 77–103.
[12] 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.
[13] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140.
[14] George Chin, Andres Marquez, Sutanay Choudhury, and Kristyn Maschhoff. 2009. Implementing and evaluating multithreaded triad census algorithms on the Cray XMT. In *Proceedings of the 2009 International Symposium on Parallel Distributed Processing (IPDPS'09)*. 1–9.
[15] Sergej Chodarev, Emilia Pietrikova, and Jan Kollar. 2015. Haskell Clone Detection using Pattern Comparing Algorithm. In *Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES'15)*.
[16] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 1 (1967), 21–27.
[17] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. 2017. Transferring Code-Clone Detection and Analysis to Practice. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE'17)*. 53–62.
[18] Derek Doran. 2015. On the discovery of social roles in large scale social systems. *Social Network Analysis and Mining* 5, 1 (2015), 1–18.
[19] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*. 109–118.
[20] Yoav Freund and Robert E. Schapire. 1997. A decision-theoretic generalization of online learning and an application to boosting. *Computer and System Sciences* 55, 1 (1997), 119–139.
[21] Jerome H. Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* (2001), 1189–1232.
[22] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (ECSMR'09)*. 219–228.
[23] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold token-based code clone detection. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*.
[24] 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 *Proceedings of the 37th International Conference on Automated Software Engineering (ASE'22)*. 1–12.
[25] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2021. FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks. *IEEE Transactions on Reliability* 70, 1 (2021), 304–318.
[26] Yu-Liang Hung and Shingo Takada. 2020. CPPCD: A Token-Based Approach to Detecting Potential Clones. In *Proceedings of the IEEE 14th International Workshop on Software Clones (IWSC'20)*.
[27] Shruti Jadon. 2016. Code Clones Detection Using Machine Learning Technique: Support Vector Machine. In *Proceedings of the 2016 International Conference on Computing, Communication, and Automation (ICCCA'16)*. 299–303.
[28] 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.
[29] Young-Bin Jo, Jihyun Lee, and Cheol-Jung Yoo. 2021. Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network. *Applied Sciences-Basel* 11, 14 (2021).
[30] J. Howard Johnson. 1994. Substring matching for clone detection and change tracking. In *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*. 120–126.
[31] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. 485–495.
[32] 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.
[33] Seulbae Kim and Heejo Lee. 2018. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Computers Security* 77 (2018), 720–736.
[34] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th Symposium on Security and Privacy (SP'17)*. 595–614.
[35] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 2001 International Static Analysis Symposium (ISAS'01)*. 40–56.
[36] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. 301–309.
[37] 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 International Conference on Software Maintenance and Evolution (ICSME'17)*. 249–260.
[38] Hongliang Liang and Lu Ai. 2021. AST-path Based Compare-Aggregate Network for Code Clone Detection. In *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN'21)*.
[39] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.
[40] Manishankar Mondal, Md Saidur Rahman, Chanchal K. Roy, and Kevin A. Schneider. 2018. Is cloned code really stable? *Empirical Software Engineering* 23 (2018), 693–770.
[41] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. 1287–1293.
[42] Jayadeep Pati, Babloo Kumar, Devesh Manjhi, and K K Shukla. 2017. A Comparison Among ARIMA, BP-NN, and MOGA-NN for Software Clone Evolution Prediction. *IEEE ACCESS* 5 (2017), 11841–11851.
[43] J. Ross Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106.
[44] Chaiyong Ragkhitwetsagul and Jens Krinke. 2017. Using Compilation/Decompilation to Enhance Clone Detection. In *Proceedings of the 11th International Workshop on Software Clones (IWSC'17)*. 8–14.
[45] 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.
[46] 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 2008 International Conference on Program Comprehension (ICPC'08)*. 172–181.
[47] 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.
[48] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.
[49] Bo Shuai, Haifeng Li, Mengjun Li, Quan Zhang, and Chaojing Tang. 2013. Automatic classification for vulnerability based on machine learning. In *Proceedings of the 2013 International Conference on Information and Automation (ICIA'13)*. 312–318.
[50] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME'14)*. 476–480.
[51] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).
[52] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. 100–109.
[53] 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.
[54] 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.
[55] 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.
[56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*.

87–98.

[57] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting Semantic Code Clones by Building AST-based Markov Chains Model. In *Proceedings of the 37th International Conference on Automated Software Engineering (ASE'22)*. 1–13.

[58] 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)*.

[59] Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. 2017. Detecting Java Code Clones with Multi-Granularities Based on Bytecode. In *Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC'17)*. 317–326.

[60] 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.

[61] Gang Zhao and Jeff Huang. 2018. Deepsim: Deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. 141–151.

[62] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*. 931–942.

Received 2023-03-02; accepted 2023-07-27