SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis

Yueming Wu†‡
Huazhong University of Science and Technology, China
wuyueming@hust.edu.cn

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

Shihan Dou†
Huazhong University of Science and Technology, China
shihandou@hust.edu.cn

Sirus Yang†§¶
Huazhong University of Science and Technology, China
yangsiru@hust.edu.cn

Wei Yang
University of Texas at Dallas, United States
wei.yang@utdallas.edu

Feng Cheng†
Huazhong University of Science and Technology, China
fcheng@hust.edu.cn

Hong Liang†
Huazhong University of Science and Technology, China
hongliang@hust.edu.cn

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

Abstract

Code clone detection is to find out code fragments with similar functionalities, which has been more and more important in software engineering. Many approaches have been proposed to detect code clones, in which token-based methods are the most scalable but cannot handle semantic clones because of the lack of consideration of program semantics. To address the issue, researchers conduct program analysis to distill the program semantics into a graph representation and detect clones by matching the graphs. However, such approaches suffer from low scalability since graph matching is typically time-consuming.

In this paper, we propose SCDetector to combine the scalability of token-based methods with the accuracy of graph-based methods for software functional clone detection. Given a function source code, we first extract the control flow graph by static analysis. Instead of using traditional heavyweight graph matching, we treat the graph as a social network and apply social-network-centrality analysis to dig out the centrality of each basic block. Then we assign the centrality to each token in a basic block and sum the centrality of the same token in different basic blocks. By this, a graph is turned into certain tokens with graph details (i.e., centrality), called semantic tokens. Finally, these semantic tokens are fed into a Siamese architecture neural network to train a code clone detector. We evaluate SCDetector on two large datasets of functionally similar code. Experimental results indicate that our system is superior to four state-of-the-art methods (i.e., SourcererCC, Deckard, RtvNN, and ASTNN) and the time cost of SCDetector is 14 times less than a traditional graph-based method (i.e., CCSharp) on detecting semantic clones.

CCS Concepts

• Software and its engineering → Software maintenance tools.

Keywords

Social Network Centrality, Semantic Tokens, Siamese Network

ACM Reference Format:


1 INTRODUCTION

Code clone detection aims to dig out code snippets with similar functionalities, which has attracted wide attention in software engineering. Commonly, code clone types are classified into four categories based on the syntactic or semantic level’s differences. The first three types of clones are syntactically similar clones, while the last type of clones characterizes semantically similar clones. As for syntactic similarity, it usually occurs when programmers conduct code copying and pasting while semantic similarity is introduced
when developers implement certain functionally similar codes from scratch.

Many approaches have been proposed to detect code clones. For example, CCFinder [28] extracts a token sequence from the input code by lexical analysis and applies several rule-based transformations to convert the token sequence into a regular form to detect Type-1 and Type-2 clones. In an effort to detect more types of clones, another state-of-the-art token-based tool, SourcererCC [43], has been designed. It captures the tokens’ overlap similarity among different methods to detect near-miss Type-3 clones. SourcererCC [43] is the most scalable code clone detector which can scale to very big code (e.g., 250M line codes). However, because of the lack of consideration of program semantics, these token-based approaches cannot handle Type-4 clones (i.e., semantic clones). To address the issues, researchers conduct program analysis to distill the semantics of code fragments into graph representations and perform graph matching (e.g., excavating isomorphic sub-graphs) to measure the similarity between given codes. Compared to token-based techniques [20, 28, 43], these graph-based detectors [32, 34, 50] achieve higher effectiveness on detecting functional code clones. However, they cannot scale to big code due to the complexity of graph isomorphism and heavy-weight time consumption of graph matching. Given large-scale clone detection is essential for daily software engineering activities such as code search [30], mining library candidates [22], and license violation detection [19, 33], there is an increasing need for a scalable technique to detect semantic clones on a daily basis.

In this paper, we propose a novel method to combine the scalability of token-based techniques with the accuracy of graph-based approaches to detect semantic code clones. Specifically, we address two major challenges.

- **Challenge 1: How to transform the high-cost graph matching into succinct token analysis while preserving the graph details?**
- **Challenge 2: How to design a scalable yet accurate similarity computation process to handle semantic clones?**

To address the first challenge, we treat the control flow graph (CFG) of a method as a social network and apply social-network centrality analysis to dig out the centrality of all basic blocks of the CFG. Centrality analysis was first introduced in social network analysis while the purpose is to find out the importance of nodes in a network. Centrality can retain the graph details and have the potential to reflect the structural properties of a graph. Therefore, instead of using traditional high-cost graph analysis, we assign the centrality to each token in a basic block and sum the centrality of the same token in different basic blocks. The outputs of this phase are certain tokens with graph details (i.e., centrality), called semantic tokens. By this, we transform the CFG into certain semantic tokens to avoid the high-cost graph matching.

To solve the second challenge, we design a Siamese network [10] to measure the similarity of a code pair. Siamese network has been widely applied in many areas, such as paraphrase scoring, where the inputs are two sentences and the output is a score of how similar they are. Given two methods, the Siamese network first maps them to the same feature space. If they are not a clone pair, the distance between them will be adjusted larger and larger as the training progresses. On the contrary, if the pair is a clone pair, the distance will be adjusted to become smaller with training, making it possible to detect semantic clones although they are syntactically dissimilar.

We implement a prototype system, SCDetector, and evaluate it on two widely used datasets, namely Google Code Jam [1] and BigCloneBench [2, 45]. In our experiments, the number of used code pairs are about 1.4 million and 0.28 million in Google Code Jam dataset and BigCloneBench dataset, respectively. Our evaluation results show that SCDetector is superior to four state-of-the-art comparative systems including one token-based method (i.e., SourcererCC [43]), one tree-based approach (i.e., Deckard [24]), and two deep learning-based detectors (i.e., RtvNN [53] and ASTNN [57]). For example, when detecting clones in BigCloneBench, the F1 scores of SourcererCC [43], Deckard [24], RtvNN [53], and ASTNN [57] are 14%, 12%, 1%, and 92% while SCDetector is able to maintain 98% of F1. Moreover, we also examine the scalability of SCDetector and our comparative systems, the results report that SCDetector consumes more time to detect code clones compared to a token-based method (i.e., SourcererCC [43]) because of the consideration of graph details. However, compared to a traditional graph-based method (i.e., CSSharp [50]), SCDetector is 14 times faster on detecting semantic clones.

In summary, this paper makes the following contributions:

- We propose a novel method to transform a CFG into certain semantic tokens (i.e., tokens with graph details) by centrality analysis. The generation of semantic tokens avoids high-cost graph analysis while preserving program semantics on detecting semantic clones.
- We design a prototype system, SCDetector, to combine the scalability of token-based approaches with the accuracy of graph-based tools for semantic clone detection.
- We conduct comparative evaluations on two widely used datasets, namely Google Code Jam [1] and BigCloneBench [2]. Experimental results show that SCDetector is able to maintain the best performance than other four state-of-the-art clone detectors (i.e., SourcererCC [43], Deckard [24], RtvNN [53], and ASTNN [57]).

**Paper organization.** The remainder of the paper is organized as follows. Section 2 presents our motivation. Section 3 shows the definitions. Section 4 introduces our system. Section 5 reports the experimental results. Section 6 discusses the future work. Section 7 shows the limitations. Section 8 describes the related work. Section 9 concludes the present paper.

## 2 MOTIVATION

To illustrate how we develop the proposed approach, we use a simplified example, which is a clone pair in BigCloneBench [2]. As shown in List 1 and 2, these two methods are both to calculate the greatest common divisor of two integers. In BigCloneBench, the clone pair is classified into a Type-4 clone, called semantic clone since they implement the same functionality with syntactically dissimilar code.

---

2. The function ID of gcd1.java and gcd2.java in BigCloneBench are 28,840 and 428,867, respectively.
We first illustrate how SourcererCC [43] (i.e., a state-of-the-art token-based clone detector) computes the similarity of two methods. SourcererCC [43] uses Overlap to measure the similarity because it intuitively captures the notion of overlap among different methods. For instance, given two methods $M_1$ and $M_2$, the overlap similarity $S(M_1, M_2)$ is calculated as the ratio between the number of same tokens shared by $M_1$ and $M_2$ and the maximum number of tokens in $M_1$ and $M_2$.

$$S(M_1, M_2) = \frac{|M_1 \cap M_2|}{\max(|M_1|, |M_2|)}$$

We conduct lexical analysis to parse the methods into several tokens. The analysis results show that the number of tokens in gcd1.java and gcd2.java is 38 and 32, respectively. Then the same tokens shared by gcd1.java and gcd2.java are obtained for computing the overlap similarity. We observe that there are 19 same tokens shared by these two methods, in other words, the overlap similarity of gcd1.java and gcd2.java is $19/38=0.5$. The default setting of similarity threshold in SourcererCC is 70%, which means that SourcererCC reports two methods as a clone pair only when the similarity of them is larger than 70%. In this case SourcererCC will cause a false negative by not reporting methods in gcd1.java and gcd2.java as a clone pair.

To achieve a more accurate clone detection, we need to incorporate information on CFGs to reflect program semantics. To achieve a scalable clone detection, we plan to distill the topology of CFGs and program semantics into certain tokens with their corresponding degree. First, we use Soot [3] to conduct static analysis to obtain the CFGs of gcd1.java and gcd2.java where each node is a basic block. Figure 1 presents the two CFGs, it is obvious that although these two methods are syntactically different, their CFGs are structurally similar because they are all designed to achieve the same functionality. Second, we dig out the degree of each basic block and assign the degree value to each token in a basic block. Finally, we sum the degree of the same token in all basic blocks and treat it as the weight of the token. For example, the degree of basic blocks 'a = b' and 'b = t' in gcd1.java are both 2. Then the weight of tokens $a = b$, $b = t$ are computed as $2+2=4$, and 2, respectively. After obtaining the corresponding weight of all tokens, we compute the overlap similarity by weight instead of by the number of same tokens shared between two methods. In this way, the total weight of all tokens in CFGs of gcd1.java and gcd2.java are 112 and 79, respectively. After overlap analysis, we find that the weight of the same tokens shared by these two graphs is 70. Therefore, the overlap similarity can be calculated as $70/112=0.625$ which is greater than $19/38=0.5$.

However, clone detection may not be accurate if we only consider the degree of a basic block as the weight in a graph. For example, suppose that there are two CFGs, namely cfg1 and cfg2, the number of basic blocks of these two graphs are 201 and 21, respectively. The degree of two basic blocks (bb1 in cfg1 and bb2 in cfg2) are both 10. It is obvious that the weights of bb1 and bb2 are different because bb1 in cfg1 is associated with half (i.e., $10/20=0.5$) of the other basic blocks while bb2 in cfg2 is only associated with 5% (i.e., $10/200=0.05$) of other basic blocks.

<table>
<thead>
<tr>
<th>Token Type</th>
<th>OriToken_Num</th>
<th>DegreeToken_Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd1.java</td>
<td>38</td>
<td>112</td>
</tr>
<tr>
<td>gcd2.java</td>
<td>32</td>
<td>79</td>
</tr>
<tr>
<td>Same Num</td>
<td>19</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 1: The similarity of different types of tokens between gcd1.java and gcd2.java

Therefore, we compute another weight of tokens by normalizing the sum of degree. In other words, the weight of a token is normalized by dividing the maximum possible degree in a graph N-1 where N is the number of nodes in the graph. As shown in Figure 1, the number of basic blocks in gcd1.java and gcd2.java are 15 and 10, respectively. Therefore, the normalized weight of all tokens in these two graphs can be computed as $112/(15-1)=8$ and $79/(10-1)=8.7778$, respectively. Similarly, we discover that the normalized weight of the same tokens shared by these two graphs is 6.3889. In other words, the overlap similarity will be $6.3889/8.7778=0.7986$ which is the highest among the three calculated similarities presented in Table 1.

![Figure 1: Control flow graphs of gcd1.java and gcd2.java](image-url)
In conclusion, `gcd1.java` and `gcd2.java` will not be detected as a clone pair if we only consider the frequency or total degree as the weight of each token. However, when we use the normalized degree as the graph details to conduct similarity computation, the code pair can be detected as a clone pair. In other words, transforming a graph representation into tokens with normalized degrees may be a great candidate for handling semantic clones.

Therefore, based on the observation, we propose a novel method by transforming the graph details into tokens with normalized degrees for semantic clone detection.

3 DEFINITIONS

Before introducing our proposed system, we first describe the formal definitions that we use throughout the paper.

```
// original
private long gcd(long a, long b) {
    while (b != 0) {
        long t = a % b;
        a = b;
        b = t;
    }
    return a;
}

// Type-1
private long gcd(long a, long b) {
    while (b != 0) {
        long t = a % b;
        a = b;
        b = t;
    }
    return a;
}

// Type-2
private long gcd(long m, long n) {
    while (n != 0) {
        long t = m % n;
        m = n;
        n = t;
    }
    return m;
}

// Type-3
public static int calculateGCD(int a, int b) {
    while (b != 0) {
        int t = a;
        a = b;
        b = t % b;
    }
    return a;
}

// Type-4
public static int GCD(int a, int b) {
    return gcd(b, a % b);
}
```

List 3: Examples of different clone types

3.1 Clone Type

In our paper, we use the following well-accepted definitions [11, 41] of code clone types.

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

To elaborate on the different types of clones, List 3 presents an example from Type-1 to Type-4 clones. The original method is to compute the greatest common divisor of two numbers. The Type-1 clone (starting in line 12) is identical to the original method. The Type-2 clone (starting in line 22) differs only in identifiers name (i.e., `m` and `n` instead of `a` and `b`). Obviously, the two types mentioned above are easy to detect. The Type-3 clone (starting in line 32) is syntactically similar but differs at the statement level. The first line in Type-3 (line 32) is totally different from the origin (line 2). The method name and types of parameters are all changed. In addition, it calculates the greatest common divisor in a similar but not identical way. Detecting Type-3 clones is harder than the previous two types. Finally, the Type-4 clone (starting in line 42) iterates to compute the greatest common divisor which is a completely different way. Its lexical and syntactic are dissimilar to the original method. Therefore, it requires an in-depth understanding of code fragments to detect Type-4 clones.

3.2 Code Granularity

We also specify a granularity unit which refers to the scale of a code fragment.

- **Token:** This is the minimum unit the compiler can understand. For example, in the statement `int i = 0;` five tokens exist: `int`, `i`, `=`, `0`, and `.`
- **Line:** This represents a sequence of tokens delimited by a new-line character.
- **Function:** This is a collection of consecutive lines that perform a specific task.
- **File:** This contains a set of functions. A file may in fact contain no functions. However, most source files usually contain multiple functions.
- **Program:** This is a collection of files.

In summary, a program is a collection of files containing functions, and a function is a collection of lines that are composed of tokens. Code cloning can occur at any of the listed granularity units. File-level and program-level code clone detection is too coarse in addition to each other, in addition to Type-1 and Type-2 clone differences.

3.3 Centrality

Prior work has validated the effectiveness of centrality analysis on different areas, such as biological network [23], co-authorship network [36], transportation network [21], criminal network [13], and affiliation network [16]. The wide usage of centrality indicates
in a basic block and sum the centrality of the same token in different basic blocks. The outputs are tokens with graph details (i.e., centrality), called semantic tokens.

- **Clone Detection**: Given a pair of code methods, the corresponding semantic tokens are fed into a Siamese network. The output is the probability that these two methods are a clone pair. If the probability is larger than 0.5, we identify that they are a pair of clones. The Siamese network is trained first by using labeled code pairs.

### 4.2 Static Analysis

In this paper, we aim to combine the scalability of token-based methods with the accuracy of graph-based methods for semantic clone detection. Therefore, we first conduct static analysis to extract the graph representation of a program. Because the programming language of the experimental dataset is Java, we implement our static analysis based on a java optimization framework, namely Soot [3], which has been used by many papers [48, 55]. In fact, the purpose of static analysis is to convert a method into a graph representation. It is not limited to which programming language (e.g., Java and C/C++) the method is since different programming languages have corresponding static analysis tools to analyze them. For example, we leverage Soot [3] to obtain the CFG of a Java method while others can use Joern [4] to extract the CFG of a C method.

To better illustrate the different phases involved in our system, we choose the method in List 2 as an example and present a more clear description in Figure 3 about the three main steps.

### 4.3 Centrality Analysis

Instead of using traditional graph matching to measure the graph similarity of two graphs, we treat a CFG as a social network and conduct centrality analysis to excavate the graph details for more efficient similarity computation. Centrality concepts were first developed in social network analysis which quantifies the importance of a node in the network. There has been proposed different centrality measures in a social network, such as degree centrality [18], closeness centrality [18], betweenness centrality [17], and katz centrality [29]. Prior work [54] has suggested that degree centrality can achieve the highest efficiency while maintaining high effectiveness on graph analysis among the listed centrality measures. Therefore, in order to conduct more scalable code clone detection, we choose degree centrality to develop our proposed system. The degree centrality [18] of a node is defined as the fraction of nodes it is connected to. The degree centrality values are normalized by dividing by the maximum possible degree in a graph $N - 1$ where $N$ is the number of nodes in the graph. Note that $deg(v)$ is the degree of node $v$.

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

### 4. SYSTEM ARCHITECTURE

In this section, we introduce our proposed system, namely SCDetector (Semantic Clone Detector).

#### 4.1 System Overview

As shown in Figure 2, SCDetector consists of three main phases: Static Analysis, Centrality Analysis, and Clone Detection.

- **Static Analysis**: This phase aims to extract the CFG of a method based on static analysis, where each node is a basic block. The input in this phase is a method while the output is the CFG of the method.

- **Centrality Analysis**: In this phase, we first dig out the centrality of each basic block of the CFG obtained from static analysis. Then we assign the centrality to each token

![Figure 2: System architecture of SCDetector](image-url)
degree centrality of all tokens in a CFG. We call these tokens with total degree centrality as semantic tokens.

In brief, the input of centrality analysis is a CFG and the outputs are tokens with graph details (i.e., centrality), called semantic tokens.

4.4 Clone Detection

Deep learning generally refers to a series of machine learning algorithms built on a neural network structure that contains multiple layers of nonlinear transformations to abstract and learn the representation of data. These methods provide effective solutions due to their powerful feature learning ability. They have greatly promoted the progress of image processing, language recognition, and other fields, and have also attracted wide attention in the field of natural language processing.

After centrality analysis, the CFG of a method is transformed into certain semantic tokens. As a matter of fact, these tokens are indeed words. If we sort these words according to the first letter, then these words can be regarded as a sentence. Therefore, we can apply techniques in the field of natural language processing to encode the sentence into vector representation for efficiently similarity computation.

Long Short Term Memory (LSTM [56]) and Gated Recurrent Unit (GRU [8, 47]) are the most widely used deep learning models in natural language processing because of the high effectiveness on text processing. Prior studies [57] have validated that GRU can achieve almost the same performance as LSTM while requires less training time on processing the same task. Therefore, we prefer GRU in our deep learning model.

To detect clone pairs, we propose to use a Siamese architecture neural network [10] which is best suited for similarity comparison of two objects. Siamese neural network is a class of neural network architectures that contains two identical subnetworks, which means that they have the same configuration with the same parameters and weights. It has been widely applied in many areas, such as paraphrase scoring, where the inputs are two sentences and the output is a score of how similar they are. One important characteristic of using Siamese neural network is that the data set can be enlarged by using pairs of inputs instead of single ones. Given $n$ samples of a class, there will be $n + (n - 1)/2$ positive pairs and many negative pairs. Another advantage of Siamese network is that sharing weights across subnetworks making it require fewer parameters to train for than a plain architecture with the same number of layers.

Figure 4 presents the Siamese architecture neural network used in SCDetector. The two identical subnetworks are two GRU neural networks sharing the same weights. The inputs of a GRU network are certain semantic tokens (i.e., tokens with total degree centrality) obtained by centrality analysis. We first train embeddings of tokens using word2vec [38] to convert a token into a fix-length vector whose dimension is set to be 100. After multiplying by the corresponding total degree centrality, the semantic vectors are then fed into two identical GRU subnetworks. The purpose of a one-layer GRU subnet is to learn the mapping from the
variable-length sequence space of 100-dimensional vectors to 50-dimensional. Finally, a comparator network takes inputs of these two subnetworks’ outputs to compute the distance. In practice, the distance measure can be adopted from several different measures, such as Manhattan distance and Cosine distance. In the paper, we leverage Cosine distance as our distance measure while others can use other measures. Moreover, the loss function used in these two subnetworks to penalize the incorrect classification is cross-entropy. The Siamese network is trained using Root Mean Square Prop (RMSProp) with a learning rate of 0.0001. The output of the trained Siamese network is the probability that two input methods are a clone pair. We claim that two methods are a pair of clones if the value is above 0.5.

5 EXPERIMENTS

In this section, we aim to answer the following research questions:

- **RQ1**: What is the effectiveness of SCDetector on detecting different types of code clones?
- **RQ2**: How the use of semantic tokens and Siamese network contribute to the effectiveness of SCDetector on detecting semantic code clones?
- **RQ3**: What is the time performance of SCDetector compared to other state-of-the-art clone detectors?

5.1 Experimental Datasets

We conduct our evaluations on two datasets: Google Code Jam [1] and BigCloneBench [2]. Programs in Google Code Jam [1] are collected from an online programming competition held by Google. In our experiment, we use the same dataset collected by [58], which consists of 1,669 projects from 12 different competition problems. As discussed in [58], projects for solving the same problem are functionally similar while those for different problems are dissimilar. Moreover, very few projects within a competition problem are syntactically similar. Therefore, we can assume that code clone pairs in the same problem are most likely to be semantic clones (i.e., Type-4 clones). The total number of similar and dissimilar method pairs are 275,570 and 1,116,376, respectively.

The second dataset used in our experiment is BigCloneBench [2] dataset, which consists of over 6,000,000 tagged clone pairs from 25,000 systems. The code granularity of clone pairs in BigCloneBench [2] is also function-level, and each clone pair is manually assigned a corresponding clone type. Because of the ambiguous boundary between Type-3 and Type-4, these two clone types are further divided into three subcategories by a similarity score measured by line-level and token-level after Type-1 and Type-2 normalizations, as follows: i) **Strongly Type-3** (ST3) with a similarity between [0.7, 1.0), ii) **Moderately Type-3** (MT3) with a similarity between [0.5, 0.7), and iii) **Weakly Type-3/Type-4** (WT3/T4) with a similarity between [0.0, 0.5).

5.2 Experimental Settings

For static analysis, we leverage a java optimization framework (i.e., Soot [3]) to generate the CFG of each method. Before obtaining the graph representation, we first need to compile the java source code into the corresponding .class. Then Soot is able to generate the CFG from the .class. In Google Code Jam dataset, we find that several programs report compilation errors such as ‘no such file or directory’. In order to correctly compile these programs, we then manually check them and find that the functionality of these programs is not process and analyze the data in an input file. After creating a file with the corresponding name in a program, we are able to generate the CFG. For BigCloneBench [2], because it does not provide the dependency libraries for most of the source code files, we select these successfully compiled files as our final dataset. The total number of final clone pairs is 280,390 including 8,139 Type-1 clones, 3,292 Type-2 clones, 3,469 Strongly Type-3 clones, 7,606 Moderately Type-3 clones, and 256,857 Weakly Type-3/Type-4 clones. Because of the lack of false tagged clone pairs in our BigCloneBench dataset, we randomly choose 280,000 dissimilar pairs from Google Code Jam dataset to complete the training and testing phase. For centrality analysis, we use a python library, networkx [6] to extract the degree centrality of all basic blocks in a CFG. Moreover, the Siamese neural network is implemented with PyTorch [6].

There has been proposed many approaches to detect code clones such as Iman [31], Rochelle [15], Toshihiro [27], Lingxiao [25], Abdullah [44], Raghavan [34], Jens [32], and Min [50]. However, most of them are not open-source. Therefore, We only compare SCDetector with the following state-of-the-art clone detection approaches:

- **SourcererCC** [43]: a state-of-the-art token-based clone detection tool.
- **Deckard** [24]: a popular AST-based clone detector.
- **RtvNN** [53]: a RNN-based clone detector that operates on source code tokens and ASTs.
- **ASTNN** [57]: a state-of-the-art deep learning-based functional clone detection tool that applies GRU on ASTs.

We run all experiments on a server with 8 cores of CPU and a GTX 1080 GPU. For both datasets, we first randomly divide them into ten subsets, then the seven subsets are used to train a model, the other two subsets are used to validate, and the last subset is used to test. We totally conduct five times and report the average results in our evaluations. Moreover, the widely used metrics to measure the detection performance are illustrated in Table 2.

### Table 2: Descriptions of used metrics in our experiments

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Abbr</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>TP</td>
<td>#samples correctly classified as clone pairs</td>
</tr>
<tr>
<td>True Negative</td>
<td>TN</td>
<td>#samples correctly classified as dissimilar pairs</td>
</tr>
<tr>
<td>False Positive</td>
<td>FP</td>
<td>#samples incorrectly classified as clone pairs</td>
</tr>
<tr>
<td>False Negative</td>
<td>FN</td>
<td>#samples incorrectly classified as dissimilar pairs</td>
</tr>
<tr>
<td>Precision</td>
<td>P</td>
<td>TP/(TP+FP)</td>
</tr>
<tr>
<td>Recall</td>
<td>R</td>
<td>TP/(TP+FN)</td>
</tr>
<tr>
<td>F-measure</td>
<td>F1</td>
<td>2<em>P</em>R/(P+R)</td>
</tr>
</tbody>
</table>

5.3 RQ1: Overall Effectiveness

5.3.1 Results on Google Code Jam. As mentioned before, projects in Google Code Jam for solving the same problem are functionally similar, and very few projects within a competition problem are
syntactically similar. Therefore, code clone pairs in the same problem are most likely to be semantic clones (i.e., Type-4 clones).

In the paper, we assume that Google Code Jam dataset is a semantic code clone dataset and conduct experiments to examine the effectiveness of SCDetector on semantic clones detection. Since the dataset used in this evaluation is the same as in [58], we directly adopt the results of Deckard [24] and RtvNN [53] as reported in [58]. Table 3 shows the detection results of SourcererCC [43], Deckard [24], RtvNN [53], and SCDetector. We ignore the result of ASTNN [57] because it takes a lot of time and some errors occur when processing code pairs in Google Code Jam dataset.

Table 3: Results on Google Code Jam dataset

<table>
<thead>
<tr>
<th></th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourcererCC</td>
<td>0.11</td>
<td>0.43</td>
<td>0.17</td>
</tr>
<tr>
<td>Deckard</td>
<td>0.44</td>
<td>0.45</td>
<td>0.44</td>
</tr>
<tr>
<td>RtvNN</td>
<td>0.90</td>
<td>0.20</td>
<td>0.33</td>
</tr>
<tr>
<td>SCDetector</td>
<td>0.87</td>
<td>0.81</td>
<td>0.82</td>
</tr>
</tbody>
</table>

SourcererCC achieves low recall and precision. It is reasonable that SourcererCC only considers the overlap similarity of tokens between two methods. As discussed in section 2, given two methods \(M_1\) and \(M_2\), the overlap similarity \(S(M_1, M_2)\) is calculated as the ratio between the number of same tokens shared by \(M_1\) and \(M_2\) and the maximum number of tokens in \(M_1\) and \(M_2\). Therefore, it can not handle semantic clones because of the lack of consideration of program semantics.

Deckard clusters the characteristic vectors of each AST subtree using predefined rules of two methods to detect clones. However, we find that more than half of the code clone pairs for solving the same competition problem have diverse parser tree structures, resulting in low precision and recall when detecting clones in Google Code Jam dataset.

RtvNN is able to achieve the highest recall but very low precision. After we manually check the detected pairs, we find that almost all the code pairs (i.e., similar pairs and dissimilar pairs) are detected as clones. It is because two functionally dissimilar methods may share syntactically similar components (i.e., IO operations) while RtvNN can not handle these issues. As discussed in [58], the distances between most methods calculated by RtvNN are in the range of [2.0, 2.8]. By lowering the distance threshold, the precision of RtvNN can be increased to 90%, however, its recall also drops quickly (down to less than 10%). As a result, it can only achieve 0.325 F1 score at the highest.

In conclusion, SCDetector is able to handle most of semantic code clones in Google Code Jam dataset compared to SourcererCC, Deckard, and RtvNN.

5.3.2 Results on BigCloneBench. On the one hand, prior work has validated that graph-based clone detectors can handle certain semantic clones. On the other hand, experimental results in a recent study (i.e., ASTNN [57]) have verified that ASTNN [57] is superior to program dependency graph-based (PDG-based) methods. Therefore, we only conduct comparative experiments to ASTNN instead of other PDG-based methods [32, 34, 50] in this evaluation.

Table 4 and 5 present the evaluation results of SourcererCC, Deckard, RtvNN, ASTNN, and SCDetector. SCDetector outperforms all the other detectors for both recall and precision. The F1 scores are encouraging as they show that SCDetector is able to handle different clone types. For example, when detecting Weakly Type-3/Type-4 clones, the F1 scores of SourcererCC, Deckard, RtvNN, and ASTNN are 2%, 2%, 0%, and 92% while SCDetector is able to maintain 97% of F1.

Table 4: F1 for each clone type in BigCloneBench

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>MT3</th>
<th>WT3/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourcererCC</td>
<td>1.00</td>
<td>1.00</td>
<td>0.65</td>
<td>0.20</td>
<td>0.02</td>
</tr>
<tr>
<td>Deckard</td>
<td>0.73</td>
<td>0.71</td>
<td>0.54</td>
<td>0.21</td>
<td>0.02</td>
</tr>
<tr>
<td>RtvNN</td>
<td>1.00</td>
<td>0.97</td>
<td>0.60</td>
<td>0.03</td>
<td>0.00</td>
</tr>
<tr>
<td>ASTNN</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
<td>0.92</td>
</tr>
<tr>
<td>SCDetector</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.97</td>
</tr>
</tbody>
</table>

As a matter of fact, the recall, precision, and F1 scores of SCDetector are higher than those on Google Code Jam dataset. To find out the reason why SCDetector performs better on BigCloneBench dataset, we manually examine several Type-4 clone pairs from BigCloneBench dataset and Google Code Jam dataset, respectively. The inspection results show that many of clone pairs in BigCloneBench share similar code structure while only differ on the sequence of the invoked API calls because these clone pairs are intentionally constructed by several experts. However, programs in Google Code Jam dataset are all implemented by different students or other programmers from scratch. Therefore, they are more difficult to be detected as a clone pair in Google Code Jam dataset compared to BigCloneBench dataset.

In conclusion, SCDetector has the ability to detect different types of code clones.

5.4 RQ2: Semantic Tokens and Siamese Network

In order to check the effectiveness of semantic tokens and Siamese network on detecting semantic clones, we conduct several single factor experiments. In the first experiment, we take inputs of the original tokens obtained from the source code of a method by lexical analysis into a GRU encoder. Given two methods, the outputs of the GRU encoder are two vectors. After obtaining the similarity (i.e., normalization) of two vectors by analyzing the Cosine distance between them, we claim that the two methods are a clone pair when the similarity is greater than 70%. In our second experiment, the original tokens are fed into a Siamese network including two identical GRU subnetworks. The output of the Siamese network is the probability that two input methods are a clone pair. Two methods are detected as a clone pair if the probability is larger than 0.5. In our final experiment, we implement SCDetector, which means that methods are first transformed into certain semantic tokens (i.e., tokens with the total degree centrality) and then fed into a Siamese GRU network to train and test. Similarly, if the probability
is larger than 0.5, these two methods are treated as a clone pair. GRU networks used in these three experiments are the same networks.

Table 6: Results on Google Code Jam dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRU-OriginalTokens</td>
<td>0.29</td>
<td>0.25</td>
<td>0.27</td>
</tr>
<tr>
<td>Siamese-GRU-OriginalTokens</td>
<td>0.72</td>
<td>0.55</td>
<td>0.53</td>
</tr>
<tr>
<td>Siamese-GRU-SemanticTokens</td>
<td>0.87</td>
<td>0.81</td>
<td>0.82</td>
</tr>
</tbody>
</table>

Table 6 presents the detection results including recall, precision, and F1 of our three single factor experiments on Google Code Jam dataset. Such results indicate that SCDetector is able to perform best because of the preservation of graph semantics (i.e., CFG) and the utilization of a Siamese network. For instance, when we only input the original tokens obtained from the source code of a method by lexical analysis into a GRU encoder to detect clones, the F1 is only 27%. However, the F1 is able to maintain 53% when we adopt a Siamese network to further process the original tokens. We find that the improvements are mainly because the Siamese network can adjust the differences between functional similar methods. Given two methods, the Siamese network first maps the input pair to the same feature space. If the pair is a clone pair, the distance between them will be adjusted to be smaller as possible with training. On the contrary, if it is not a clone pair, the distance will be adjusted larger and larger. As a matter of fact, clones in Google Code Jam dataset are implemented by different programmers from scratch, thus these semantic clones are almost syntactically different. If we only use a GRU network to encode a method into a vector representation and then compute the similarity to identify clones, it is generally difficult to detect such clones since they are almost syntactically different. However, when we apply a Siamese network to train and detect clones, the distance of these semantic clone pairs will be adjusted to become smaller as the training progresses, making it possible to detect such clones.

Additionally, if we first transform the graph details into semantic tokens, and then feed these semantic tokens as the input pair to the Siamese network, the F1 is able to increase to 82%. This happens mainly due to the consideration of graph details. As discussed in Section 2, we present a pair of input methods which is to calculate the greatest common divisor of two integers. They belong to semantic clone since they implement the same functionality with syntactically dissimilar code. When we compute the overlap similarity by using original tokens, the pair cannot be detected by SourcererCC. This is because these two methods are implemented in a completely different way, thus it is difficult to be detected if we only consider the original tokens. Figure 1 shows the two CFGs and we can see that these two graphs are structurally similar since they are designed to achieve the same functionality. In other words, semantic clones may share some similar subgraphs when we distill the semantics into a graph representation. Therefore, when we attach the graph details to tokens, these semantic tokens can be more effective on detecting code clones.

In conclusion, the attachment of graph details to tokens and the adoption of a Siamese network are both effective on detecting code clones.

Figure 5: Time performance of SourcererCC, Deckard, RtvNN, ASTNN, and SCDetector

5.5 RQ3: Scalability

In this part, we pay attention to the runtime performance of SCDetector and four comparative systems. In order to test the scalability of these clone detectors, we randomly select 1,000,000 code pairs from Google Code Jam dataset as our test objects. We run all tools on these randomly selected code pairs three times and report the average runtime. For deep learning-based methods, the complete procedure consists of model training and model testing. For example, the clone detection procedure of ASTNN consists of GRU training and GRU testing while SCDetector composes of Siamese network training and Siamese network testing. Therefore, in Figure 5, we present the training time and prediction time of RtvNN, ASTNN, and SCDetector separately.

For SourcererCC and Deckard, they do not need to conduct the training phase so the training runtime overheads are both zero. In addition, SourcererCC takes the least time to detect clones because it is a pure token-based clone detector. Compared to RtvNN, SCDetector takes less time to train while requires more time to detect code clones. However, from the experimental results in Table 3, 4, and 5, we can see that the ability of RtvNN on semantic clone detection is very low. Moreover, as for ASTNN, results in Figure 5 report that SCDetector is more scalable than ASTNN not only on training phase but also on testing phase. The training time and testing time of ASTNN are 16,096 seconds and 2,894 seconds while SCDetector only needs to take 3,076 seconds (i.e., 2,937 seconds for training and 139 seconds for testing) to complete the whole clone detection. It is reasonable because the number of hidden layers in SCDetector is only one which requires less time to train and test.

SCDetector aims to balance the ability of token-based and graph-based techniques to detect semantic clones. As for token-based methods, SourcererCC is the most scalable tool that can scale to very big code and we have presented the runtime overhead in Figure 5. As for graph-based approaches, we select one state-of-the-art traditional graph-based clone detection tool namely CCSharp [50] to compare the scalability. CCSharp aims to solve the problem of PDG-based tools’ high time cost. It adopts two strategies (i.e., PDG’s structure modification and characteristic vector filtering) to
decrease the overall computing quantity. CCSharp is not open-source and we can not test it on Google Code Jam dataset. However, they present the scale of their dataset and report the time performance in their paper. Table 7 shows the details of two datasets including the total lines of code (LOC), the total number of methods, and the time performance. Obviously, our randomly selected 1 million pairs are larger than the dataset in CCSharp. CCSharp does not need to train, thus the training time is zero. In reality, the training phase of SCDetector is the most time-consuming process, however, it is a one-time offline phase. Once the model is trained, it can be reused to compute the code similarity between two given methods. Compared to CCSharp, the prediction time of SCDetector is extremely less than CCSharp. The runtime overhead on detecting clones is 1,995.9 seconds for CCSharp while is 139 seconds for SCDetector when given a trained model. In other words, given a trained model, SCDetector is 14 times faster than CCSharp.

Table 7: The scale of datasets used in CCSharp and our evaluation

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LOC</th>
<th>#methods</th>
<th>Training Time</th>
<th>Prediction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL in CCSharp [50]</td>
<td>86,096</td>
<td>1,134</td>
<td>0</td>
<td>1,995.9</td>
</tr>
<tr>
<td>Google Code Jam (1 million pairs)</td>
<td>98,117</td>
<td>1,669</td>
<td>2,937</td>
<td>139</td>
</tr>
</tbody>
</table>

In summary, because of the consideration of graph details and the adoption of deep learning, SCDetector requires more time to detect clones compared to a token-based method (i.e., SourcererCC). However, compared to a graph-based method (i.e., CCSharp), SCDetector is 14 times faster due to the transformation of graph details.

6 DISCUSSIONS

Differences from the most similar systems. The most similar related methods to SCDetector are Oreo [42] and Centroid [12]. Oreo also applies a Siamese network to detect clones, however, the inputs of its Siamese network are 24 method-level software metrics while are certain semantic tokens in SCDetector. Moreover, since SCDetector takes inputs of semantic tokens, we leverage two identical one-layer GRU subnetworks to measure the distance between them. As for Oreo, the Siamese network consists of two four-layer DNN subnetworks, which are different from SCDetector. Centroid aims to detect application clones on Android markets, it uses a geometry characteristic, called centroid, of CFGs to measure the similarity of methods in two apps. The graph representation of Centroid is the same as SCDetector, that is, a CFG of a method. In physics, especially when analyzing forces on a physical object, people usually use the center of mass (i.e., centroid) to represent an object. When two objects are identical, their centroids are also the same. In SCDetector, we regard the CFG of a method as a social network and apply social-network-centrality analysis to extract the degree centrality of all basic blocks. Although the graph representations of Centroid and SCDetector are the same, the perspective of graph analysis is completely different, that is, the physical analysis of Centroid and the social network analysis of SCDetector.

Why SCDetector outperforms the other approaches. The reasons are mainly two-fold. First, SCDetector considers the program semantics by transforming the CFG of a method into corresponding semantic tokens. These tokens are used to measure code similarity of different methods. However, traditional token-based techniques (e.g., CCFinder [28] and SourcererCC [43]) have no ability to handle semantic clones since they only care about syntactic level’s code features rather than semantic level’s details because of the high time cost of the semantic analysis process. Second, we apply a Siamese network in SCDetector to train and detect code clones. Siamese network is best suited for similarity comparison of two objects. Given two methods, it first maps the input pair to the same feature space. If they are not a clone pair, the distance between them will be adjusted larger and larger as the training progresses. On the contrary, if the pair is a clone pair, the distance will be adjusted to become smaller with training, making it possible to detect semantic clones. Through our experimental results, we can see that the Siamese network can indeed improve the detection effectiveness. However, as for RtvNN [53] and ASTNN [57], after obtaining the vector representations, they compute the similarity directly, making them perform worse than SCDetector.

7 LIMITATIONS

First, the key insight of SCDetector is to transform the CFG of a method into tokens with graph details. Therefore, we need to obtain the graph representation first by static analysis. Because the experimental datasets are implemented in Java, we leverage a Java optimization framework, namely Soot [3] to complete our static analysis phase. However, Soot [3] requires to successfully compile the given codes first and then the CFG can be extracted. It is the reason why we cannot use all the files in BigCloneBench dataset to commence our evaluations. In our future work, we plan to implement a static analysis tool or leverage other static analysis tools (e.g., WALA [7]) to generate the CFG of methods from source code directly.

Second, SCDetector extracts the CFG of a method and detects clone by measuring the similarity of methods. Therefore, SCDetector can only detect method-level code clones and can not handle clones in other code granularity units (e.g., line-level). Moreover, copying a method and then pasting with large number of edits can cause false negatives by SCDetector since the CFGs of the original method and the pasted method are significantly different. Such clones are considered as large-gap clones in [51]. We plan to combine other network properties with centrality to mitigate the issue.

Third, SCDetector is based on the degree centrality of tokens and relies on the common tokens between two programs. Although the extraction of a CFG from a method source code by Soot can normalize several tokens, SCDetector may cause false negatives when the same functionality is implemented using different APIs and different graph structures. We plan to normalize the source code first and then conduct static analysis to generate abstracted CFGs for more effective comparison.

Fourth, we generate semantic tokens by analyzing the degree centrality of all basic blocks in a CFG since the extraction of degree centrality is the most efficient among several different centrality measures [54]. However, degree centrality reflects the
relative number of connections of a node in a CFG and is limited in representing the graph context. We plan to use more different centrality measures to find the most suitable centrality that can balance the effectiveness and the efficiency on code clone detection.

Fifth, since the input of our Siamese network in SCDetector is semantic tokens, SCDetector may cause false positives when methods realize different functionalities based on some unique tokens while having a very similar structure. The most critical operations of these methods are different, resulting in completely different semantics. SCDetector can not handle this type of code clones. We plan to combine the attention [49] with the centrality of all tokens to mitigate the situation.

Sixth, the degree centrality can quantify the importance of tokens in a CFG. In SCDetector, the purpose of degree centrality extraction is to maintain the graph details to achieve a semantic code comparison. High degree centrality does not indicate high impact on our final comparison results. In our GRU network, the inputs of semantic tokens are simply sorted according to the first letter. We plan to analyze the tokens in the source code first to obtain more accurate orders of all tokens.

8 RELATED WORK

In this part, we introduce studies related to code clone detection, which can be classified into five main categories, as follows: the text-based, the token-based, the tree-based, the graph-based, and the metrics-based methods.

For the text-based methods [14, 26, 40], the similarity between two code snippets are measured in the form of text or strings. [26] designs a fingerprinting technique to find out code clones. [14] presents a language-independent method to detect similar codes by simply line-based string matching. However, these two techniques do not support Type-3 clone detection. In order to detect more types of clones, Nicad [40] introduces a two-stage approach which consists of i) identification and normalization of potential clones using flexible pretty-printing and ii) similarity computation by simply text-line comparison using longest common subsequence algorithm. Although Nicad can detect several Type-3 clones, it has no ability to handle Type-4 clones since it ignores the program semantics of given code fragments.

For the token-based techniques [20, 28, 35, 43, 51], tokens are firstly obtained from program code by lexical analysis. CCFinder [28] extracts a token sequence from the input code and applies several rule-based transformations to convert the token sequence into a regular form for detecting Type-1 and Type-2 clones. In order to support Type-3 clone detection, SourcererCC [43] has been designed. It captures the tokens’ overlap similarity among different methods to detect near-miss Type-3 clones. SourcererCC [43] is the most scalable code clone detector which can scale to 250M line code clone detection. However, similar to text-based methods, token-based approaches can not handle Type-4 clones either.

For the tree-based tools [24, 52, 57], Abstract Syntax Tree (AST) is used as the code representation to detect code clones. The main idea of Deckard [24] is to compute characteristic vectors within ASTs and apply Locality Sensitive Hashing (LSH) to cluster similar vectors for code detection. CDLH [52] first transforms ASTs into binary trees and then adopts Tree-LSTM [46] on these trees to encode them as vector representations. Finally, these vectors are used to measure the similarity among different codes. Unlike CDLH [52], ASTNN [57] splits each large AST into a sequence of small statement trees. After encoding these statement trees into vectors, a bidirectional RNN model is used to produce the final vector representation of a code fragment to discover semantic code clones. These tree-based tools are able to detect semantic clones, however, they suffer from low scalability because of the large execution times.

For the graph-based methods [12, 32, 34, 50, 58], program semantics are firstly distilled into different graph representations, such as program dependency graph and control flow graph. [32] and [34] both extract the program dependency graphs of code fragments and identify similar codes by digging out isomorphic subgraphs to represent code clones. In an effort to improve the runtime performance of [32] and [34], CCSharp [50] adopts two strategies to mitigate the overall computing cost: graph structure modification and characteristic vector filtering. However, it still suffers from low scalability on large-scale code clone detection due to the complexity of graph isomorphism and heavy-weight time consumption of graph matching.

For the metrics-based approaches [9, 37, 39, 42], metrics can be obtained from tree or graph representations of source code or directly from source code. Both [9] and [37] extract metrics from AST to represent the source code and uses them to identify code clones. In addition, [39] uses different categories of metrics (e.g., classes, coupling, and hierarchical structure) extracted from source code to detect clones. These methods leverage features from code to measure the semantic similarity of two code fragments.

9 CONCLUSION

In this paper, we propose a novel method to measure the similarity of semantic codes, namely SCDetector. SCDetector is a combination of token-based and graph-based approach. Given a method source code, we first generate the CFG and then apply centrality analysis to transform the graph into certain semantic tokens (i.e., tokens with graph details). Finally, these semantic tokens are fed into a Siamese network to train a model and use it to detect code clone pairs. We evaluate SCDetector on two widely used datasets and experimental results show that SCDetector is superior to other four state-of-the-art clone detectors (i.e., SourcererCC [43], Deckard [24], RtvNN [53], and ASTNN [57]). Moreover, the time cost of SCDetector is 14 times less than a traditional graph-based method (i.e., CCSharp [50]).

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the National Science Foundation of China under Grant No. 61672249, by the Shenzhen Fundamental Research Program under Grant No. JCYJ201704131142156114, by the Guangdong Provincial Science and Technology Plan Project under Grant No. 2017B010124001 and the Key-Area Research and Development Program of Guangdong Province under Grant No. 2019B010139001.

References


