CCGraph: a PDG-based code clone detector with approximate graph matching

Yue Zou[*]
School of Computer Science and Technology, University of Science and Technology of China
Hefei, China
zy1996@mail.ustc.edu.cn

Bihuan Ban[*]
School of Data Science, University of Science and Technology of China
Hefei, China
banbihua@mail.ustc.edu.cn

Yinxing Xue[†]
School of Computer Science and Technology, University of Science and Technology of China
Hefei, China
yxxue@ustc.edu.cn

Yun Xu[†‡]
School of Computer Science and Technology, University of Science and Technology of China
Hefei, China
xuyun@ustc.edu.cn

ABSTRACT

The software clone detection is an active research area, which is very important for software maintenance, bug detection etc. The two pieces of cloned code reflect some similarities or equivalents in the syntax or structure of the code representations. There are many representations of code like AST, token, PDG etc. The PDG (Program Dependency Graph) of source code can contain both syntactic and structural information. However, most existing PDG-based tools are quite time-consuming and miss many clones because they detect code clones with exact graph matching by using subgraph isomorphism. In this paper, we propose a novel PDG-based code clone detector, CCGraph, that uses graph kernels. Firstly, we normalize the structure of PDGs and design a two-stage filtering strategy by measuring the characteristic vectors of codes. Then we detect the code clones by using approximate graph matching algorithm based on the reforming WL (Weisfeiler-Lehman) graph kernel. Experiment results show that CCGraph retains a high accuracy, has both better recall and F1-score values, and detects more semantic clones than other two related state-of-the-art tools. Besides, CCGraph is much more efficient than the existing PDG-based tools.

CCS CONCEPTS

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

KEYWORDS

Clone detection, Program dependence graph, WL graph kernel

[*]Also with Key Laboratory on High Performance Computing, Anhui Province.
[†]Yinxing Xue and Yun Xu are the corresponding authors.
[‡]Also with Key Laboratory on High Performance Computing, Anhui Province.

ACM Reference Format:
Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416541

1 INTRODUCTION

Software development often has the phenomenon of copying similar or identical code fragments from the existing source codes, called *code clones*. Existing research shows that code cloning is very important in the software development and maintenance [1, 2], like software refactoring [3], bug detection [4], copyright plagiarism detection [5], code evolution analysis. In the various kinds of code clone detection approaches, PDG (Program Dependency Graph)-based approaches can find the code clones both in syntactic similarity and semantic similarity. As Bellon et al. [6] noted, PDG-based approaches can also report non-contiguous clones that cannot be perceived by other techniques. However, most PDG-based approaches consume too much time beacuse they match graphs exactly by using subgraph isomorphism, since this is an NP-hard problem [7]. Besides, in existing PDG-based approaches, many of the code clones are missed due to using exact or very close subgraph isomorphism. Therefore, not only the PDG-based code clone detector with approximate graph matching can find more code clones, but also make the processing time faster. In addition, there are some deep learning method, such as Oreo [15], which encode software metrics including PDG metrics into vectors and achieve good results. However, these methods are dependent on the initial training data and lack interpretability of output results.

Among these PDG-based approaches, Liu et al. [5] proposed a classic PDG-based clone detection by using the VF subgraph isomorphism algorithm. And they did not perform any filtering operations on PDG candidate sets. Since VF subgraph isomorphism algorithm belongs to the method of exact graphs match, this results in the expensive time cost. For the issue of high complexity of subgraph isomorphism, Gabel et al. [8] proposed a method of mapping PDG to AST and used the similarity of AST to replace the similarity of PDGs. Although this method speeds up the clone detection, it loses a lot of semantic information and results in many missing clones.

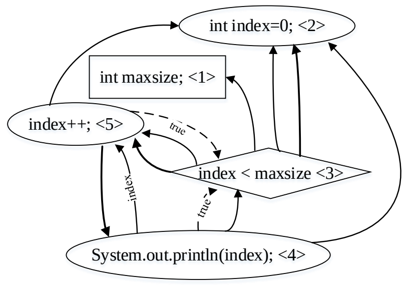
```
1: public static void loop1(int maxsize){
2:     int index=0;
3:     while(index<maxsize){
4:         System.out.println(index);
5:         index++;
6:     }
7: }
```

(a) The Source Code of loop1

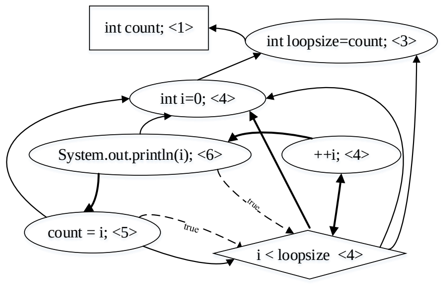
```
1: public static void loop2(int count){
2:     int loopsize=count;
3:     for(int i=0;i<loopsize;++i){
4:         count=i;
5:         System.out.println(i);
6:     }
7: }
```

(b) The Source Code of loop2

Figure 1: The PDG-based Clone Pair for Loop Printing.



(a) PDG for loop1



(b) PDG for loop2

Figure 2: Two PDGs for Loops in Figure 1.

Futhermore, for the issue of large PDG candidate sets, both Li et al. [4] and Wang et al. [9] adopted some strategies to modify the PDG and reduce the size of PDG candidate sets, like removing irrelevant nodes and edges. However, they still do not avoid solving the issue of subgraph isomorphism directly.

Besides the processing time, there are many clones missed for existing PDG-based approaches. For example, the PDGs of the original and the clone code are given in Figure 1. Both of the code pieces implement the same function of loop printing. However, one uses a *while* loop and the other one uses a *for* loop, and there are some other different statements. From the PDGs shown in Figure 2, there is no exact subgraph isomorphism between the two PDGs, but there are some local similarities. As we can see, the diamond nodes in two graphs have similar outgoing and inbound edges. Moreover, they have similar neighbor nodes which are assignment or declaration statements, and they have similar control dependencies with with their neighbors. Also, there are some other local similarities in the two PDGs, which are not described in details here. However, the existing PDG-based methods cannot detect these local similarities. Therefore, we adopt the approximate graph matching algorithm to detect these local similarities which is actually a kind of approximated PDG-based code clones.

The approximate graph matching algorithms are mainly divided into two classes, graph embedding and graph kernel methods [10]. Graph embedding methods embed a graph into a low-dimensional vector based on the characteristics of the graph, which can quantitatively measure the similarity between nodes and is more convenient to apply. However, this dimensionality reduction process loses a lot of graph structural information and severely affects the accuracy of the graph matching. The graph kernel methods [11] divide a graph into several sub-structures as kernels, and then calculate the similarity of the two graphs through their sub-structures. Different decomposition methods and substructures correspond to different types of graph kernel methods. For example, the WL (Weisfeiler-Lehman) [11] graph kernel collects the labels of the adjacent nodes of each node in two graphs, and measures the similarity of two nodes according to the similarity of the labels. The more similar nodes in two graphs, the closer the two graphs are. This kind of methods not only retain the advantages of low computational cost of the kernel function, but also contain various types of graph information such as directed edges and labels. Therfore, the WL graph kernel is very suitable for calculating the similarity between PDG pairs. Besides, the approximate graph matching based on graph kernels has been successfully applied in the fields of human face recognition [12], network analysis [13], and compound classification [14].

In this paper, we propose a novel PDG-based code clone detector, called CCGraph, which can find more PDG-based clones than other tools and is much faster than those PDG-based tools. This approach reduces the size of original PDGs at first, and then filters PDG pairs by a two-stage strategy with characteristic similarities. Finally, CCGraph identifies clone pairs by an approximate graph matching algorithm based on WL (Weisfeiler-Lehman) [11] graph kernel algorithm. To sum up, our study makes the following contributions:

1) We adopt approximate graph matching algorithm based on WL graph kernel. WL graph kernel sorts and compresses the labels of

all adjacent vertices of each node in each iteration [11], and we can calculate the similarity of two graphs by comparing the number of similar nodes in graphs. We design the iteration times according to the graph diameter, and add the weight of each iteration considering the distance of the nodes. Our tool can accelerate the comparison of graph similarity and detect more clones than the state-of-the-art tools.

2) In the preprocessing stage, we design a two-stage filtering algorithm. First, we count some numerical characteristics of PDG for rough filtering, such as the size of PDG, and classify the similar PDGs into the same category. Then, for each category, we calculate the similarity of string characteristics like function names based on $JaroWinkler$ distance [20]. In addition, we optimize the structure of PDGs by eliminating and merging some irrelative nodes. These strategies can reduce the size of candidate clone pairs greatly.

3) We present the whole PDG-based clone detector called CCGraph, reduce the candidate clone pairs scale in the preprocessing stage and adopt the approximate graph matching algorithm. It can detect more PDG-based clones than the latest clone detector Oreo [15] based on deep learning for Java and be much faster than the latest PDG-based clone detector CCsharp [9] for C code.

The rest of this paper is structured as follows. Section 2 introduces some concepts and definitions used in our research. Section 3 describes the details of methods proposed in our clone detection process. Section 4 shows the implementation and experiment of our tool against some other state-of-the-art tools on some codebases. Section 5 and 6 summarize the related work and discuss the limitations of our approach. Finally, Section 7 concludes the present work with a discussion of future work of CCGraph.

2 PRELIMINARIES AND DEFINITIONS

In this section, we give the definitions of some important concepts and notations used in the detection work, like PDG, subgraph isomorphism, and the WL graph kernel which is used in our proposed method. We also give the definitions of PDG-based clones we need to detect.

2.1 Program Dependency Graph

A program dependency graph (PDG) is a labeled and directed garph of the source code to show some dependencies like Figure 2. The nodes in PDG can be classified into the system and statement nodes. The system nodes are generated from PDG generation tools, like function entrance or exit nodes, which is not the specific code statement. The statement nodes include each statement and its type for them.

The edges show some dependencies between statement nodes, such as the data and control dependencies. The control dependency edge is from a control node to a next node if the condition controls that the next node will be executed. The data dependency edge is between two nodes that they all use the same variable or one of them assigns the variable and the another uses it either directly or indirectly like pointers. There is one more dependency edge called execution dependency, which is between two nodes if one of the nodes may only be executed after the another one.

Definition 2.1. Program Dependency Graph: The PDG for a code program is represented as $G = (V, E, \mu, \delta)$, where V is the set of nodes in graph, E is the set of edges in graph, $\mu : V \longrightarrow S$ is a function assigning types to nodes in graph, $\delta : E \longrightarrow S$ is a function assigning dependency types to edges.

2.2 Subgraph Isomorphism

Recently, many PDG-based clone detections define the clone pair of program p and p' if and only if there is subgraph isomorphism relationship among their PDGs. The mathematic definition of subgraph isomorphism is defined as below.

Definition 2.2. Graph Isomorphism: $G_1(V_1, E_1, \mu_1, \delta_1)$ is a graph isomorphism to $G_2(V_2, E_2, \mu_2, \delta_2)$ if and only if there is a bijective function $f : V_1 \longrightarrow V_2$ and $f^{-1} : V_2 \longrightarrow V_1$ satisfying:

- $\mu_1(v) = \mu_2(f(v))$ for any $v \in V_1$.
- $\forall e = (v_1, v_2) \in E_1, \exists e' = (f(v_1), f(v_2)) \in E_2$ such that $\delta(e) = \delta(e')$.
- $\forall e = (v_1, v_2) \in E_2, \exists e' = (f^{-1}(v_1), f^{-1}(v_2)) \in E_1$ such that $\delta(e') = \delta(e)$.

Definition 2.3. Subgraph Isomorphism: $G_1(V_1, E_1, \mu_1, \delta_1)$ is a subgraph isomorphism to $G_2(V_2, E_2, \mu_2, \delta_2)$ if and only if there is an injective function $f : V_1 \longrightarrow V_2$ that there is a subgraph $S \subseteq G_2$ such that f is a graph isomorphism from G_1 to S.

2.3 Weisfeiler-Lehman Graph Kernel

Mathematically, the method of graph kernels maps graphs to a vector feature space, and calculates the graph similarity by their inner product in the vector feature space. In the process of calculating, it divides a graph into substructures, and any combination of graph decomposition and substructure isomorphism judgment can be defined as a new graph kernel.

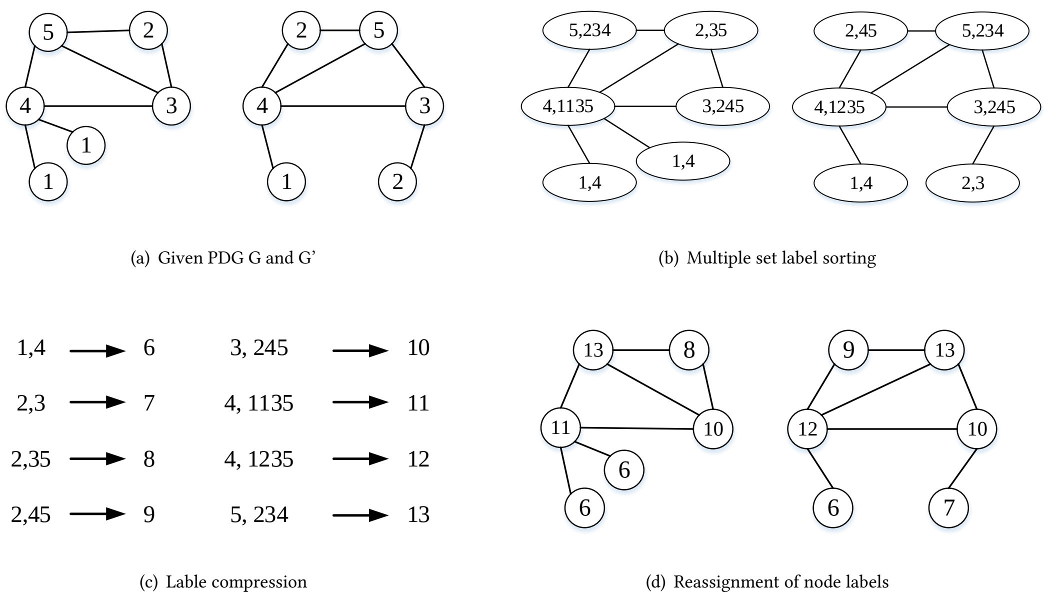
Definition 2.4. Graph Kernel: $\mathcal{G}$ is a finite set of graphs, R is a point product space and the function $k : (\mathcal{G} \times \mathcal{G}) \longrightarrow R$ is a graph kernel, then there is a Hilbert space F and a mapping function $\gamma : \mathcal{G} \longrightarrow F$, for all $G_1,\ G_2 \in \mathcal{G}$, $k(\ G_1, G_2) =< \gamma(G_1), \gamma(G_2) >$, where $< \cdot, \cdot >$ means the inner product in Hilbert space.

Generally, given two graphs G_1, G_2, and the substructures sequences $\{S_{1,1}, S_{1,2}, \ldots, S_{1,n_1}\}$ of G_1 and $\{S_{2,1}, S_{2,2}, \ldots, S_{2,n_2}\}$ of G_2, where $S_{1,i}$ is the i^{th} substructure of G_1 and $S_{2,j}$ is the j^{th} substructure of G_2, the graph kernel of G_1 and G_2 can be expressed as

$$k_r(G_1, G_2) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \sigma(S_{1,i},\ S_{2,j})$$

where $\sigma(S_{1,i}, S_{2,j}) = 1$ when $S_{1,i}$ and $S_{2,j}$ are isomorphism else $\sigma(S_{1,i}, S_{2,j}) = 0$.

Definition 2.5. Weisfeiler-Lehman Graph Kernel: Given two graphs $G_1(V_1, E_1, \mu_1, \delta_1)$ and $G_2(V_2, E_2, \mu_2, \delta_2)$ and iteration number h, after the h^{th} times iteration of WL subtree isomorphism algorithm, we get the structures of two graphs as $G_1(h) = (G_1, l_h(G_1))$ and $G_2(h) = (G_2, l_h(G_2))$. And $l_h(G_1)$ is the set of node labels for G_1 in



(a) Given PDG G and G'

(b) Multiple set label sorting

(c) Lable compression

(d) Reassignment of node labels

Figure 3: Computation of the WL graph kernel for one iteration

the iteration h, the WL graph kernel is expressed as

$$k_{WL}^{(h)}(G_1, G_2) = k(G_1(0), G_2(0)) + ... + k(G_1(h), G_2(h))$$

$k(G_1(h), G_2(h))$ is the base kernel function of G_1 and G_2, which calculates the number of same label pairs in node label sets $l_h(G_1)$ and $l_h(G_2)$.

Weisfeiler-Lehman (WL) graph kernel [11] is an advanced graph kernel in recent years. As shown in Figure 3, the WL graph kernel sorts the labels of all adjacent vertices of each node, compresses these labels into new shorter label based on a hash algorithm, and then counts the same original and compressed labels in two graphs. After one iteration of computation, the compressed labels denote subtree patterns or sub-structures in graph. For instance, in the Figure 3 (a), the node (denoted as v) with label 5 in G and the same label node (denoted as v') in G' have same neighbor nodes with label 2, 3 and 4. After collecting the neighbor node labels, both of the label sequences of v and v' are 2,345, which is shown in Figure 3 (b). Then after the label compression shown in Figure 3 (c), the new label of v and v' is 13. After one iteration of the WL graph kernel calculating, in the Figure 3 (d), the v and v' have the same label, which means that the local graph structures around v and v' are same. When most of the nodes of the two graphs have this kind of similarity, we think that the two graphs are very similar.

2.4 Similarity of Graphs Based on WL Graph Kernel

After the approximate graph matching, we need to distinguish whether the original code fragments are clones according to the kernel value we calculated. Hence, we need to define a measure of similarity based on the WL graph kernel value after normalizing.

Definition 2.6. Similarity of PDG pairs : Given two program dependency graphs $G_1(V_1, E_1, \mu_1, \delta_1)$ and $G_2(V_2, E_2, \mu_2, \delta_2)$, we measure the similarity of PDG pairs by the h times iterations of WL graph kernel value as

$$similarity(G_1, G_2) = \frac{k_{WL}^{(h)}(G_1, G_2)}{|V_1 \cup V_2|}$$

, the similarity value is between 0 and 1, and the code fragments of G_1 $andG_2$ are *code clone* when the $similarity(G_1, G_2)$ is bigger than the predefined threshold T.

3 PROPOSED METHOD

In this section, we introduce the overview of the proposed method of clone detection firstly, and then describe the design of the simplification of PDG structures, the characteristic vector extraction, the filtering strategies and the approximate graph matching algorithm based on the WL graph kernel in details.

3.1 Overview

The overview of our proposed method is shown in Figure 4. It consists of two process phases, code preprocessing and clone detection. Code preprocessing analyzes the source code files, extracts PDGs in function level, simplifies the PDG structures and filters the candidate PDG pairs by characteristic vectors. Clone detection measures the similarity by calculating the WL graph kernel values of PDG pairs and identifies the code clone if PDG pairs satisfy the threshold condition.

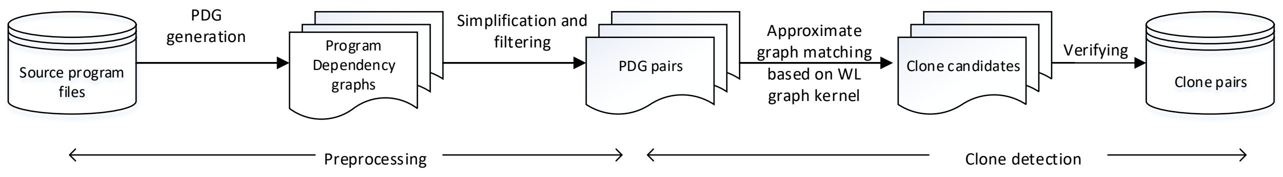


Figure 4: The overview of the proposed method.

3.2 Simplification of PDG Structures

In the PDG generation, we observed that there are a large number of system nodes and edges which are not related to code semantic information. Also, there are some sub-PDGs which link to other functions because of function calls in source code. Therefore, we proposed two strategies to simplify the structure of PDGs, eliminating meaningless nodes and merging function call sub-PDGs.

3.2.1 Eliminate meaningless nodes. In the process of PDG generation, system nodes and some assistant parameter nodes are genreated automatically by the PDG generation tools. These nodes do not affect the original logical structure of the source code. Therefore, we simply remove these nodes and the edges connected to them to reduce the size of PDGs.

3.2.2 Merge function call sub-PDGs. In the PDGs of source code, the calls to other functions are plotted as a sub-PDGs. Considering that the codes corresponding to these sub-PDGs are not in the current file, we simplify to merge the sub-PDG part to one function call node. This merging operation does not lose any structure information of the original PDG but greatly reduce the size of PDGs. We name the new node type *Fun_Call* to represent the function call sub-PDGs in original PDGs.

3.3 Filtering of PDG Pairs

Clone detection makes PDG pair-wise comparison, and the scale of PDG pairs increases exponentially with the number of PDGs. Therefore, it is necessary to filter PDG pairs for reducing the time consuming of clone detection.

We mainly filter PDG pairs from two aspects. On the one hand, we consider the size of PDG directly, such as the number of nodes and edges in PDGs. And for the case that the scale ratio of two graphs is too big, we filter these PDG pairs as other cloning detection methods. On the other hand, we adopt the filtering strategies according to some important informations in PDGs, such as the data type of input and output. The overall of PDG's characteristics are listed in Table 1 and the filtering process is structured as following subsections, which is also shown in algorithm 1.

3.3.1 Numerical similarity filtering. For the numerical part of the characteristic vector, we filter PDG pairs by size of PDG and scale ratio firstly, then calculate the cosine similarity between two numerical vectors. PDG pairs are divided into categories according to their numerical similarity.

Algorithm 1 Characteristic Vectors Filtering

Input: P is characteristic vector of code PDG G. P' is characteristic vector of code PDG G'. L is threshold of node number in PDG characteristic. T is threshold of PDG pair's scale radio. G_S is threshold of the string similarity and G_N is threshold of the numerical similarity of the characteristic vectors.

Output: R is the candidate PDG pairs for the clone detection

```
function FILTER($G, G', P, P', L, T, G_S, G_N$)
    for each code PDG $G$ and $P$ in clone codes do
        for each code PDG $G'$ and $P'$ in clone codes do
            if $sizeof(G) < L$ or $sizeof(G') < L$ then
                PDG pair $(G, G')$ is filtered
            else if $min(G, G')/max(G, G') < T$ then
                PDG pair $(G, G')$ is filtered
            else if number similarity of $(P, P') > G_S$ then
                $R \leftarrow R \cup (G, G')$
                CONTINUE
            else if string similarity of $(P, P') > G_N$ then
                $R \leftarrow R \cup (G, G')$
            end if
        end for
    end for
end function
```

Table 1: The Characteristic Vector of PDG

Numerical characteristics	Description
NumOfCtrlEdges	The control dependency edges.
NumOfExcEdges	The excute dependency edges.
NumOfDataEdges	The data dependency edges.
NumOfDeclNodes	The variable declaration nodes.
NumOfAssignNodes	The assignment nodes.
NumOfCtrlNodes	The control nodes.
NumOfStateNodes	The function state nodes.
NumOfOtherNodes	other types nodes.
NumOfReference	The number of reference variables.
String characteristics	**Description**
ReturnPara	The return parameters.
IntroPara	The incoming parameters.
Name	The function name.

- Size filtering: In real code dataset, there are many functions that only have a return statement. This kind of clones have no reference value for practical application. So, we focus on those PDG pairs which have meaningful size. And in the detailed implementation, we set the size threshold as 6 lines, which is also a common filter standard of other clone detections like CCAligner [17].
- Scale ratio filtering: We need to match PDGs in pairwise, and the pair have no point in real-world if the scale ratio of two PDGs is too big, like one PDG size is 10 times than the other one. We filter the pairs if the scale ratio of two PDGs is bigger than the specific threshold.
- Numerical similarity filtering: We extract the numerical vectors according to the characteristics of PDGs like the number of different kinds of nodes and edges. Node types are divided into variable declaration, assignment, control statement, function call and other kind of statements. Connecting edges are classified as control dependency edge, data dependency edge and execution dependency edge according to the dependency relationship. Then we filter those PDG pairs if the cosine similarity of two vectors do not meet the threshold we set. According to the results of repeated experiments, we set a threshold of 0.9 for numerical similarity filtering.

3.3.2 String similarity filtering. In the characteristic vectors of PDGs, some dimensions are string variables, like the function name, input parameters, output variables. For each category classified by numerical similarity, we need to calculate the string similarity between these string part of vector to filter some PDG pairs. Since the string parts of vectors are short strings and the $Jaro\ Winkler$ algorithm has a great effect on short text similarity calculations [14, 15], we choose the $Jaro\ Winkler$ distance ratio [20] to measure the string similarity. The Jaro-Winkler algorithm gives the starting part a higher score for the same string, which defines a prefix range p. For the two strings to be matched, the prefix part has the same length as the partial string of length L. Considering the diversity of string variables in reality, we only filter those PDG pairs which $Jaro\ Winkler$ distance ratio less than 0.5.

- Jaro Distance: Jaro distance d_j of given strings a, b is $d_j = \frac{1}{3}(\frac{m}{|a|} + \frac{m}{|b|} + \frac{(m-t)}{m})$, the m is the number of matching characters, and t is half the number of transpositions, $|a|$ and $|b|$ is the length of string a and b.
- Jaro Winkler Distance: Jaro Winkler distance d_w of give strings a, b is $d_w = d_j + L * P(1 - d_j)$, dj is the jaro distance of given strings a and b, L is the length of the prefix partial match, P is a range factor constant used to adjust the weight of the prefix match, but the value of P cannot exceed 0.25, because the final score may exceed 1 point. The standard default setting of $Jaro\ Winkler$ is $P = 0.1$.

3.4 Approximate Graph Matching Based on WL Graph Kernel

After the simplification of PDG structures and filtering of PDG pairs, we adopt the approximate graph matching algorithm to measure the similarity between the PDG pairs.

Considering the specialty of PDGs, we choose to implement the approximate graph matching algorithm based on WL graph kernel. In all kinds of graph kernels, WL graph kernel can be applied to directed and labeled graphs, and it is able to process the attributes of nodes in graphs. WL graph kernel has better time-consuming performance than the traditional kernels like Random-Walk or Shortest-Path graph kernel [11]. Also, there are open source implementations of WL[1] graph kernel used in bioinformatics to compare protein structure and we can easily apply it to our tool with a secondary development. Since WL graph kernel has great classification performance and it is one of the best graph kernels at present, we choose to adopt our approximate graph matching algorithm based on WL graph kernel.

Algorithm 2 Clone Detection With WL Graph Kernel Testing of Two Graphs

```
Input: the PDG pair $(G, G')$, all nodes $v$ and labels $l(v)$ of $v$ in $G$
    and $G'$; $N(v)$ is the set of $v's$ neighbor nodes; $m$ is the number of
    iterations of WL algorithm; $T$ is the threshold of PDG similarity
    of $G$ and $G'$.
Output: $R$ is the result set of PDG pairs after the clone detection
 1: function CLONE($G, G', v, l(v), N(v), m, T$)
 2:     for each pair $(G, G')$ in candidate pairs do
 3:         for $i_{th}$ iteration in $m$ times do
 4:             for each node $v$ in $G$ and $G'$ do
 5:                 $M(v) \leftarrow \sum_( l(u)|u\exists N(v))$
 6:                 $s(v) \leftarrow M(v)$ sorted by ascending order
 7:                 $s(v) \leftarrow l(v) + s(v)$
 8:                 $l(v) \leftarrow f(s(v))$
 9:                 WHEN $f : \sum_( *) \rightarrow \sum$
10:                 STATISFY $f(s(v)) = f(s(w))$
11:                 if and only if $s(v) = s(w)$
12:             end for
13:             $w_i \leftarrow (m - i + 1)/m$
14:             $k_i = sizeof(samelabel(v))/sizeof(total(v))$
15:         end for
16:         $k \leftarrow \sum_( k_i * w_i)$
17:         if $k > T$ then
18:             $R \leftarrow R \cup (G, G')$
19:         end if
20:     end for
21: end function
```

The steps of our proposed algorithm is described in detail as following and shown in Algorithm 2:

- Hash the label of each node in the PDG according to the grammar category as the initial label of the node.
- Set the h iterations for the WL algorithm. In the h iterations, each iteration collects the labels of the current node's neighbors as a sequence, and compresses the label sequence into a new label by using local sensitive hashing algorithm.
- If the labels of the two nodes are same after the iteration, it is considered that the subtrees with the two nodes as the root node and the height of h are isomorphic.

[1]https://github.com/BorgwardtLab/GraphKernels.

- Calculate the number of same node pairs between two PDGs, which is also the graph kernel value. If the graph kernel values of the two PDGs satisfy the threshold range, the two PDGs are considered as PDG-based code clone candidates.

Furthermore, considering the different diameters of different clone PDGs, we dynamically set the iterations number according to the diameter of the smaller PDG, and count the number of nodes with the same label between PDGs after each iteration. In addition, we add the weight factor $(h-n+1)/h$ to the n^{th} iteration to give higher weight to the lower iterations because each node in source code is more affected by its neighbors than the nodes far away.

3.5 Verifying

After the approximate graph matching based on WL graph kernel, we get the kernel value of PDG pairs. This step is to vefity whether the candidates are code clones.

For each PDG pair, we calculate the WL graph kernel value $k_{WL}^{(h)}(G_1, G_2)$ in the approximate graph matching, and we measure the similarity by the ratio of the kernel value to the total number of nodes in two PDGs shown in *Section* 2. When the similarity exceed the threshold we set before, we consider the code fragments of the PDG pair are *code clone*. In order to further verify whether two PDGs are cloned or not, we use the *asymmetric coefficient* of *Dice similarity* [21] from another point of view, which is defined as follows:

$$sim(G_1, G_2) = \frac{k_{WL}^{(h)}(G_1, G_2)}{min(|V_1|, |V_2|)}.$$

where V_1 and V_2 are the node sets of PDG G_1 and G_2. The similarity threshold is chosen with best results among plenty of repeated experiments. Therefore, we verify the two PDGs are code clones when their similarity is greater than or equal to 0.9.

4 EVALUATION

In this section, we design and implement the experiments of our clone detection CCGraph and empirical evaluation against other state-of-the-art clone detectors. Firstly, we begin by introducing the environment configuration of our experiments. We introduce the datasets used in experiments in detail and explain the reasons for choosing these datasets.

4.1 Experimental Configuration

Since most of current PDG-based clone detectors are commercial that are not open source, we choose the open source and executable tools to be compared with our approach. For C code, we choose the latest PDG-based clone detector CCSharp [9] for comparison both on artificial datasets and real-world datasets. However, CCSharp can not be implemented to Java code. For Java code, there is no targeted PDG-based clone detector. There is a clone detector Oreo [15] based on deep learning to be regarded as the candidate. In the process of model training, Oreo uses some informations of PDG as a part of feature input, so it can find some PDG-based code clones. Therefore, we choose Oreo for comparison on Java datasets. We also compare CCGraph with other baselines such as token-based (SourcererCC, in order to find out more Type-3 clones, we set a 80% threshold), AST-based (Deckard,also with a 80% threshold) on Java datasets. In addition, there is no standard PDG tested benchmark to measure the accuracy and recall rate of clone detection methods. For this, we manually check the experiment results on the tested datasets.

For the PDG generation, we use Frama-C[2] [23] to generate PDGs of C code. For Java programs, we choose the Java PDG generator TinyPDG[3] [24] based on an open source library. All PDGs of C and Java code are unified to *.dot* file fomat. After the PDG generation, we used Python scripts to parse *.dot* files and transform these files into the specified format for subsequent graph matching. In the implementation of approximate graph matching, we used Python and C++ to calculate similarity based on some relative libraries. All experiments are executed on a same machine with Ubuntu 14.04LTS operation system with a quad-core CPU and 8GB of memory.

4.2 Experimental Datasets

We performed experiments on CCGraph against the CCSharp [9] on C code and Oreo [15] on Java code on several code datasets. These datasets include real-world and artificial code datasets. Since the PDG generation can only be applied to compilable code, we choose some high quality real-world code datasets and develop some artificial code datasets from the real-world codes by kinds of disguises to simulate the generation of code clones in daily programming [21, 22]. The details of these code datasets are shown in the Table 2, including their LoC (Line of Code), number of functions and description.

Table 2: The Details of Datasets

Datasets	LoC	Functions	Description
BCB-5	4374	186	Decompress zip archive.
BCB-11	8384	338	Initialize Java Eclipse projects.
BCB-15	9895	478	Load custom font.
BCB-38	9572	481	Get MAC address string.
Less	19233	286	Linux text viewer.
PostgreSQL	86096	1134	dataset server.
Artificial-1	1658	100	Generated artificial dataset.
Artificial-2	20157	384	Generated artificial dataset.

For C code, we choose the *less* and several modules of *PostgreSQL* datasets. The *less* dataset is used in CCSharp [9], GPLAG [5] and the *PostgreSQL* program is also common used in kinds of clone detector. And we also generate two artificial datasets based on some real-world dataset like leetcode or cJSON by using a mixture of four types disguises [25]: format changing (i.e. change spaces, comments and layouts), renaming/type chaning (i.e. rename varibles, functions), reordering statements, and control substitution (i.e. replacing *while* loops with for). These disguises do not affect the overall syntactic structure and PDG dependencies in original code, but may not be detected by those tools using subgraph isomorphism.

For Java code, we choose BigCloneBench, which is common used in all kinds of clone detectors. BigCloneBench [27] is a large benchmark of manually validated clones by mining the big real-world dataset IJaDataset-2.0 [28] (25,000 Java systems). The current

[2]http://frama-c.com/index.html.
[3]https://github.com/YoshikiHigo/TinyPDG

public release version of BigCloneBench contains 8 million clones of 43 distinct functionalities [29]. Considering the BigCloneBench is too big for us to calculate the accuracy and recall of our method, we randomly select several small functionalities.

4.3 Experimental Results

4.3.1 Efficiency of PDG Structure Simplification. We have tested the effect of PDG structure simplification on several typical datasets. These strategies effectively simplify PDGs and do not damage the original semantic information and the results of efficiency are shown in Table 3. According to the research [16], the time costed by graph matching grows exponentially with the growth of the numbers of nodes and edges in graph. Therefore, the reduction of PDG size by removing meaningless nodes and merging function call sub-PDGs is very beneficial to the following processes of clone detection.

Table 3: Efficiency of PDG Structure Simplification

Datasets	Language	Size of original files (A)	Size of simplification files (B)	Ratio of simplification (B/A)
Less	C	871KB	732KB	0.84
PostgreSQL	C	19.7MB	9.6MB	0.49
Artificial-1	C	436KB	408KB	0.94
Aritificial-2	C	1.1M	891KB	0.79
Datasets	Language	Size of original files (A)	Size of simplification files (B)	Ratio of simplification (B/A)
BCB-5	Java	559KB	482KB	0.86
BCB-11	Java	955KB	799KB	0.84
BCB-15	Java	1.3MB	1.1MB	0.85
BCB-38	Java	1.2MB	998KB	0.81

4.3.2 Efficiency of Filtering Strategies. The filtering phase is very important in the whole clone detection. We have implemented the heuristic filtering strategies and the results of filtering efficiency are shown in Table 4. The filtering ratio refers to the number of PDG pairs after filtering divided by the number of total PDG pairs. It can exclude the majority of candidate PDG pairs, and helps to accelerate the graph matching process later.

In order to ensure the accuracy of our filtering strategies, we conducted a sample check on the filtered PDG pairs. We randomly seletected 20 PDG pairs for three times from the filtered PDG pairs of each dataset, and determine the number of clone pairs misfiltered in the 20 PDG pairs. According to the results shown in Table 5, the number of our misfiltered clone pairs are very small, which also verifies the effectiveness of our filtering strategies.

4.3.3 Results of Code Clone Detection. We evaluate CCGraph with the total clone quantity, clone quality, time cost and scalability. The total clone quantity is the total number of clone pairs detected by tools, which is shown in Table 6. The clone quality includes the precision, recall rate and the F1-score we calculated, which is shown Table 7 and Table 8. The total clone result used in the recall rate is the union of the results generated by all tools. Also, we compare the time cost of the whole clone detection process with CCSharp

Table 4: Efficiency of Filtering Strategies on Datasets

Datasets	Language	Theoretical PDG pairs (A)	PDG pairs after filtering (B)	Ration of filtering (B/A)
Less	C	81796	1316	1.61%
PostgreSQL	C	1285956	22247	1.73%
Artificial-1	C	10000	562	5.62%
Aritificial-2	C	147456	2708	1.84%
Datasets	Language	Theoretical PDG pairs (A)	PDG pairs after filtering (B)	Ration of filtering (B/A)
BCB-5	Java	34596	596	1.72%
BCB-11	Java	114244	1805	1.58%
BCB-15	Java	228484	3309	1.45%
BCB-38	Java	231361	3424	1.48%

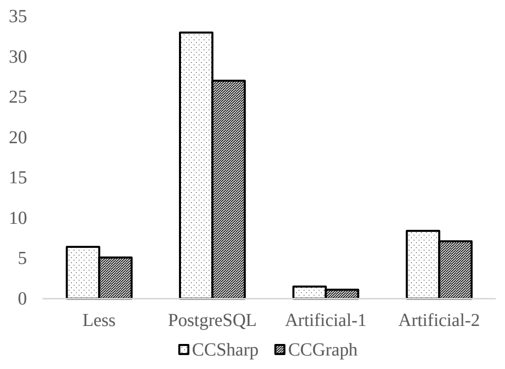
Table 5: The Number of Misfiltered Clone Pairs of 3 Sampling Examinations

Datasets	Language	1st experiment	2nd experiment	3rd experiment
Less	C	0/20	0/20	0/20
PostgreSQL	C	0/20	1/20	0/20
Artificial-1	C	0/20	0/20	0/20
Aritificial-2	C	0/20	0/20	0/20
Datasets	Language	1st experiment	2nd experiment	3rd experiment
BCB-5	Java	0/20	0/20	0/20
BCB-11	Java	0/20	1/20	0/20
BCB-15	Java	0/20	0/20	0/20
BCB-38	Java	0/20	0/20	1/20

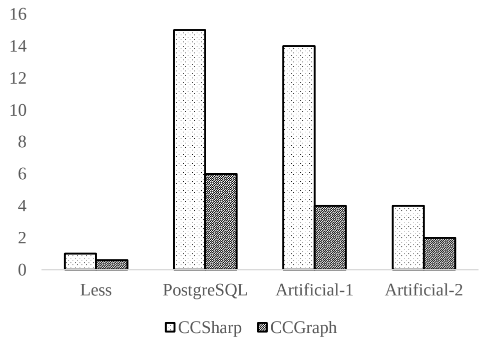
and measure the scalability on datasets of different scales, which results are shown in Figure 5 and Table 9.

Total number of clones. We generate PDGs and detect clones in function granularity. We measure the total number of clones by the number of clone pairs against other tools. In the PDG filtering stage, we exclude those code which number of lines are less than 6. And from the clone results shown in Table 6, we know that CCGraph has obvious advantages in the number of clone pairs than CCSharp [9] in all *C* datasets and Oreo [15] in all *Java* datasets. As shown in Table 6, our approach can find more than twice as many PDG-based clone pairs on Artificial-2 dataset as CCSharp. And even in the worst case, we can find 30 percent more PDG-based clone pairs than Oreo. The reason is that we adopt the approximate graph matching, reduce the requirement of clone matching and find some clone pairs missed by other tools.

Clone quality. Clone quality includes precision, recall rate and F1-score value. And we choose those small code datasets to make it easier to calculate the precision and recall rate. For precision, we manually check and evaluate all the detected clone pairs to judge if they are real clones, and calculate the ratio of all true positive code



(a) In preprocessing stage(min)



(b) In graph matching stage(s)

Figure 5: Time cost comparison between CCSharp and CCGraph

Table 6: The Number of Clones Detected by CCGraph Against Other Tools

Dataset	Language	CCSharp	CCGraph
Less	C	12	**23**
PostgreSQL	C	149	**297**
Artificial-1	C	105	**183**
Artificial-2	C	74	**162**
Dataset	Language	Oreo	CCGraph
BCB-5	Java	47	**76**
BCB-11	Java	129	**161**
BCB-15	Java	75	**138**
BCB-38	Java	102	**131**

clones to all clones we detected.

$$Precision = \frac{TP}{TP + FP}$$

where TP is the true positive PDG-based code clones reported by the tool, and FP refers the wrong clone pairs we detected. For recall rate, we define

$$Recall = \frac{TP}{TP + FN}$$

where TP is same as precision definition, FN refers the true clone pairs we missed, and the denominator $TP + FN$ is the union result of the two tools as the total detected clones (removing duplicate and FP elements). We also define

$$F1 - score = \frac{2 * Precision * Recall}{Precision + Recall}$$

From the results shown in table 7 and 8, we find that our results are slightly less than CCSharp and Oreo in accuracy, when we also maintain a high level of accuracy which is acceptable in reality. The little loss of accuracy is because of the examples that the two PDGs share almost the same data and control dependencies in their structure but the function of their programs are obviously different. But our recall rate is significantly higher than CCShrap and Oreo. In the best case, our recall rate is almost 40% higher than CCsharp and Oreo. In addition, we calculate the F1-score value, and CCGraph is at least 10% better than CCSharp and Oreo on each dataset.

Table 7: Clone Results on C Code Dataset

Datasets	Tools	Accuracy	Recall	F1-score
Less	CCSharp	**0.92**	0.52	0.64
	CCGraph	0.89	**0.81**	**0.85**
PostgreSQL	CCSharp	**0.99**	0.56	0.72
	CCGraph	0.91	**0.85**	**0.83**
Artificial-1	CCSharp	**0.94**	0.63	0.75
	CCGraph	0.90	**0.80**	**0.85**
Artificial-2	CCSharp	**0.99**	0.42	0.59
	CCGraph	0.92	**0.81**	**0.86**

Time cost and scalability. Considering the PDG-based clone detection is more time-consuming than other types clone detection tools and the time of training datasets in Oreo [15] is hard to measure, we only compare the time cost with CCSharp [9]. We divide the overall time cost into preprocessing and graph matching time cost. And the time cost results are shown in Figure 5. The preprocessing stage includes PDG generation, structure optimization and filtering. From the results shown in Figure 5, CCGraph reduces the time cost both on preprocessing and graph matching stages, and accelerates the graph matching greatly than CCSharp [9].

Also, to ensure that CCGraph can be extended to larger code datasets than CCSharp, we compare the runtime of the two tools on datasets of different sizes. From the results shown in Table 9,CCGraph is much slower than the token-based detector Oreo because of the high time-consuming of graph algorithms. However,

Table 8: Clone Results on Java Code Dataset

Datasets	Tools	Accuracy	Recall	F1-score
BCB-5	SourcererCC	**1**	0.02	0.03
	Deckard	0.22	0.35	0.27
	Oreo	0.94	0.63	0.75
	CCGraph	0.87	**0.94**	**0.90**
BCB-11	SourcererCC	0.88	0.08	0.14
	Deckard	0.19	0.32	0.24
	Oreo	**0.96**	0.73	0.80
	CCGraph	**0.96**	**0.91**	**0.93**
BCB-15	SourcererCC	0.75	0.03	0.05
	Deckard	0.24	0.22	0.23
	Oreo	**0.96**	0.56	0.71
	CCGraph	0.86	**0.92**	**0.89**
BCB-38	SourcererCC	**1**	0.09	0.16
	Deckard	0.29	0.30	0.29
	Oreo	0.98	0.66	0.79
	CCGraph	0.95	**0.83**	**0.89**

CCGraph is than CCSharp, the PDG-based detector on all datasets, and it can also run normally on large-scale datasets, while CCSharp can not be extended to large-scale datasets at the level of 50*M* lines of code. This also shows that CCGraph has better scalability than CCSharp at least.

Table 9: Time Cost of CCGraph and CCSharp on Datasets of Different Sizes

LoC	10K	1M	10M	30M	50M
CCSharp	14s	1h12m3s	8h1m18s	–	–
Oreo	2s	4m22s	24m12s	2h11m30s	4h36m12s
CCGraph	4s	15m10s	2h21m10s	7h22m12s	17h48m21s

5 RELATED WORK

There have been many kinds of code clone detection methods in the literature, Rattan et al. [30] summarized many tools in his research. These methods can be divided into text-based [31, 32, 33, 34], token-based [16, 35, 36, 37], AST-based [38, 39], PDG-based [5, 9, 41, 42, 43], metric-based [44, 45, 46] and others like deep learning [22, 47, 48, 49].

Among the text-based code clone detections, the representative work compared the code fragments in the form of text. Baker [31] proposed a parametric matching algorithm, regarded the code line as a long string, and used the string matching algorithm to detect completely consistent code clones. NiCad [33] detected the potential code clones using the longest common string subsequences comparison. Johnson [34] developed the clone detection by a fingerpringting technique. However, these clone detections based on text can only detect those exactly same code clones or clones with high text similarity. Considering the various practical clone detection scenarios, the industry pays more attention on PDG-based clone detections recently [30].

Among the token-based code clone detections, they extracted the tokens of source code by lexical analysis tools firstly, and then compared thosed tokens instead of keywords since tokens can tolerate more different identifiers. CCFinder [35] is a popular clone detector based on token, but it does not support detecting the structural similarity of code fragments. CCAligner [17] is good at those clones with relatively concentrated changes but it will misses many PDG-based clones which CCGraph can detect. SourcererCC [38] performed great in detecting format transformation and renaming changes, but only limited to the token similarity. This also means that they are not suitable for the detection of PDG-based clones.

Among the AST-based code clone detections, CloneDR [39] and Deckard [40] transformed the code into abstract syntax tree (AST) and detected the similar subtrees to find code clones. But the transformation lost much structure information of code and this may lead to the missing of many code clones.

Among the PDG-based clone detections. Duplix [41] and PDG-DUP [42] both detected PDG-based clones using program slicing and subgraph isomorphism matching. And these methods all suffered from high time consuming and missed many clones, which can not be applied to large scale code datasets. To improve the time performance, Sargsyan et al. [43] modified PDG generated by LLVM through removing isolated nodes, GPLAG [5] used a lossy filter to reduce the plagiarism scale by characteristic vectors, CCSharp [9] proposed some new PDG modification and filtering stragegies. However, these PDG-based clone methods only applied some simple filtering strategies, and used the exact graph matching algorithm like subgraph isomorphism to detect PDG-based clones. These methods still have a unacceptable time consumption and poor scalabilities. alse they have a big loss of code clones.

Among the metric-based clone detections, they extracted the charateristic vectors of source code and calculated the vector similarities to detect code clones. Patenaude et al. [44] divided the code to different categories using the metrics. Balazinska et al. [45] and Mayrand et al. [46] both extracted the vectors from the AST of source code. These methods based on metrics are very fast on detecting clones but they have high false positive rate and miss much clones because of the rough comparison.

There also exists some other techniques for clone detection, such as detections based on deep learning techniques and code behaviors. Wei et al. [47] proposed an approach to learn the representations and Hamming distance of source code and detected code clones. White et al. [48] proposed a model to detect code clones by learning the discriminating features of source code. Yu [50] used a new tree-based convolution to detect semantic clones, by capturing both the structural information of a code fragment from its AST and lexical information from code tokens. Oreo [15] and DeepSim [22] used the clones that are almost identical or very similar to train the deep learning model, the ability of detecting clones with large difference is limited. The deep learning methods always have too much dependency on the training data and lack interpretability, which may not suitable for some special scenarios. The experiments in Section 4 show that CCGraph can find more PDG-based code clones than Oreo.

6 LIMITATIONS

There are also some limitations in our research work. One limitation is that there is no any opened and standard benchmark datasets for PDG-based clone detection methods. We can not compare our approach with other clone detectors on any unified datasets. Therefore, we need to perform our experiments on the code datasets to manually verify the accuracy and recall rate of clone results. Another limitation is that the existing PDG generators need to be improved. Now both of the PDG generation of C code and Java code can only be suited to the compliable programs. This requires that the test datasets must be a complete compilable program and also limits the scope of PDG-based clone detection. Therefore, we need to develop a PDG generator for code segments. In this way, our approach can be more widely processed.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a PDG-based clone detection work with approximate graph matching algorithm called CCGraph. We introduce the preprocessing stage to optimize the structure of PDGs and filter the scale of candidate clone pairs by a two-stage strategy. Moreover, we design the approximate graph matching algorithm based on WL graph kernel to detect the clone pairs. Finally, we evaluate CCGraph with clone quantity, clone quality and scalability against Oreo [15] and CCSharp [9]. The experiment results show that our method perform the better recall rate and F1-score value while maintaining a high accuracy.

In the future work, we plan to further improve the performance of CCGraph. And we will consider investigating the PDG generation of more types of programming languages. Besides, we want to integrate the CCGraph into the code development and management systems. Meanwhile, we can develop some downstream applications based on CCGraph, such as software structure analysis, bug detection and other software analysis applications.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grant No.61672480 and No. 61972373) and the Program for Excellent Graduate Students in Collaborative Innovation Center of High Performance Computing. The research of Dr. Xue is supported by CAS Pioneer Hundred Talents Program.

REFERENCES

[1] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development.* IEEE, 7-7.

[2] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. 2010. An empirical study on the maintenance of source code clones.*Empirical Software Engineering* 15, 1 (2010), 1–34.

[3] Yun Lin, Zhenchang Xing, Xin Peng, Yang Liu, Jun Sun, Wenyun Zhao and Jinsong Dong. 2014. Clonepedia: Summarizing code clones by common syntactic context for software maintenance. In *IEEE International Conference on Software Maintenance and Evolution.* IEEE, 341-350.

[4] Jingyue Li and Michael D. Ernst. 2012. CBCD: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 310-320.

[5] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 872–881.

[6] 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), 577-591.

[7] David S Johnson. 1987. The NP-completeness column: An ongoing guide. *Journal of algorithms*, 8(2), 285-303.

[8] Mark Gabel, Lingxiao Jiang and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering.* ACM, 321-330.

[9] Min Wang, Pengcheng Wang, Yun Xu. 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In *24th Asia-Pacific Software Engineering Conference (APSEC).* IEEE, 100-109.

[10] Navarin N, Sperduti A. 2017. Approximated Neighbours MinHash Graph Node Kernel. *European Symposium on Artificial Neural Networks Computational Intelligence and Machine Learning*, ESANN, 281-286.

[11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn and Karsten M. Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep), 2539-2561.

[12] Foggia Pasquale, Percannella G, Vento M. 2014. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01), 1450001.

[13] Ya Jun, Liu Z S, Chang, Q. 2016. The network attack graph analysis based on graph kernel. *Journal of Military Communications Technology*, Vol.37: 20-25.

[14] Gaüzère B, Brun L, Villemin D. 2011. Two New Graph Kernels and Applications to Chemoinformatics. In *International Conference on Graph-based Representations in Pattern Recognition.*

[15] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18).* ACM, 354–365.

[16] Krinke Jens. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering.* IEEE, 301-309.

[17] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering.* ACM, 1066-1077.

[18] William W Cohen, Pradeep Ravikumar, and Stephen E Fienberg. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks. *IIWeb*, Vol. 73-78.

[19] Van der Loo and Mark PJ. 2014. The stringdist package for approximate string matching.*The R Journal 6.1,* 111-122.

[20] Feigenbaum James. 2016. JAROWINKLER: Stata module to calculate the Jaro-Winkler distance between strings.

[21] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. 1999. Modern information retrieval.*ACM press New York,*Vol. 463..

[22] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 141-151.

[23] Cuoq P., Kirchner F., Kosmatov N., Prevosto V., Signoles J. and Yakobowski B. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods.* Springer,233-247.

[24] Higo Yoshiki, and Shinji Kusumoto. 2011. Code clone detection on specialized PDGs with heuristics. In *European Conference on Software Maintenance and Reengineering.* IEEE, 75-84.

[25] Gang Zhang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2012. Cloning practices: Why developers clone and what can be changed. In *IEEE International Conference on Software Maintenance (ICSM).* IEEE, 285-294.

[26] Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun. 2014. Detecting differences across multiple instances of code clones. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 164-174.

[27] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating Clone Detection Tools with BigCloneBench. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME 2015).* IEEE, 131-140.

[28] Ambient Software Evoluton Group. (2013). IJaDataset 2.0. http://secold.org/projects/seclone.

[29] Svajlenko Jeffrey and Chanchal K. Roy. 2016. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 596-600.

[30] Rattan Dhavleesh, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology 55.7 (2013),* 1165-1199.

[31] Baker Brenda S. 1997. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing 26.5 (1997),* 1343-1362.

[32] Baker Brenda S. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering.* IEEE, 86-95.

[33] Cordy James R. and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension.* IEEE, 219-220.

[34] Johnson J Howard. 1994. Substring Matching for Clone Detection and Change Tracking.*ICSM.* Vol. 94, 120-126.

[35] Kamiya Toshihiro, 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.

[36] Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.*OSdi.* Vol. 4. No. 19, 289-302.

[37] Göde Nils and Rainer Koschke. 2009. Incremental clone detection. In *2009 13th European Conference on Software Maintenance and Reengineering.* IEEE, 219-228.

[38] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K.Roy and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* IEEE,1157-1168.

[39] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance.* IEEE, 368–377.

[40] 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.* IEEE Computer Society, 96–105.

[41] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of Eighth Working Conference on Reverse Engineering.* IEEE, 301–309.

[42] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International Static Analysis Symposium.* Springer, 40–56.

[43] Sargsyan, Sevak, Kurmangaleev S, Belevantsev A and Avetisyan A. 2016. Scalable and accurate detection of code clones. *Programming and Computer Software* 42.1 (2016), 27-33.

[44] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. 1999. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension.* IEEE, 49–56.

[45] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 1999. Measuring clone based reengineering opportunities. In *Proceedings of the 6th International Software Metrics Symposium.* IEEE, 292–303.

[46] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. 2015. A comparative study on the bug-proneness of different types of code clones. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 15).* IEEE, 91–100.

[47] 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 26th International Joint Conference on Artificial Intelligence (IJCAI 17).* 3034–3040.

[48] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 87–98.

[49] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC:memory comparison-based clone detector. In *Proceedings of the 33rd International on Software Engineering.* ACM, 301–310.

[50] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC).* IEEE, 2019: 70-80.