Gitor: Scalable Code Clone Detection by Building Global Sample Graph

Junjie Shan*
shanjunjie@westlake.edu.cn
Westlake University
Hangzhou, China

Shihan Dou*
shihandou@foxmail.com
Fudan University
Shanghai, China

Yueming Wu†
wuyueming21@gmail.com
Nanyang Technological University
Singapore

Hairu Wu
hrwu20@fudan.edu.cn
Fudan University
Shanghai, China

Yang Liu
yangliu@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Code clone detection is about finding out similar code fragments, which has drawn much attention in software engineering since it is important for software maintenance and evolution. Researchers have proposed many techniques and tools for source code clone detection, but current detection methods concentrate on analyzing or processing code samples individually without exploring the underlying connections among code samples.

In this paper, we propose *Gitor* to capture the underlying connections among different code samples. Specifically, given a source code database, we first tokenize all code samples to extract the pre-defined *individual information* (*e.g., keywords*). After obtaining all samples' individual information, we leverage them to build a large *global sample graph* where each node is a code sample or a type of individual information. Then we apply a node embedding technique on the global sample graph to extract all the samples' vector representations. After collecting all code samples' vectors, we can simply compare the similarity between any two samples to detect possible clone pairs. More importantly, since the obtained vector of a sample is from a global sample graph, we can combine it with its own code features to improve the code clone detection performance. To demonstrate the effectiveness of *Gitor*, we evaluate it on a widely used dataset namely BigCloneBench. Our experimental results show that *Gitor* has higher accuracy in terms of code clone detection and excellent execution time for inputs of various sizes (1–100 MLOC) compared to existing state-of-the-art tools. Moreover, we also evaluate the combination of *Gitor* with other traditional vector-based clone detection methods, the results show that the use of *Gitor* enables them detect more code clones with higher F1.

---
*Equal contribution
†Yueming Wu is the corresponding author

CCS Concepts

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

Keywords

Clone Detection, Node Embedding, Global Sample Graph

1 INTRODUCTION

Code clone, also known as duplicate code or similar code, refers to the existence of two or more identical or similar source code fragments. Numerous empirical studies [6, 23, 42] have shown that code cloning widely exists in different open source or closed source code bases. For example, [6, 37] detected 22.3% of code clones in Linux system, Kamiya et al. found 29% code clones in JDK, and even up to 50% code clones in some software systems [50]. Widespread code cloning has helped the development of software systems to a certain extent and can have positive benefits [21, 40]. However, many studies have pointed out that a large number of code clones can have a negative impact on software systems maintenance [19, 31, 49], since it may introduce bugs or vulnerabilities. Therefore, the automatic detection of code clones has attracted wide attention in the field of software engineering.

According to the syntactic or semantic similarity of code clones, Bellon et al. classified code clones into four types [8, 41]: textual similarity (type 1), lexical similarity (type 2), syntactic similarity (type 3), and semantic similarity (type 4). From type 1 to type 4, the similarity of cloned codes gradually decreases and the difficulty of detection gradually increases. A number of code clone detection method has been proposed [17, 20, 44, 46, 51, 55, 59]. For example, a state-of-the-art token-based method namely *SourcerCC* [44] is designed to capture the tokens' overlap similarity among different methods to detect Type-1 to Type-3 clones. In practice, token-based techniques are unable to handle Type-4 clones (*i.e.,* semantic clones) due to a lack of respect for program semantics. To mitigate the issue, researchers use program analysis to distill the semantics of code

fragments into tree or graph representations (*e.g.,* abstract syntax tree and control flow graph) and apply tree or graph matching to quantify the similarity between different codes. Empirical studies [24, 26, 51] have shown that tree-based and graph-based code clone detectors can achieve better performance on semantic code clone analysis. However, due to the complexity of tree and graph structures, they are unable to scale to large programs. Given that large-scale clone detection is essential to daily software engineering activities such as code search [22], mining library candidates [16], and license violation detection [12, 25], there is an increasing need for a scalable technique to detect code clones.

In this paper, we propose a novel code clone detection method leveraging global graph built across code samples. We find that almost all current code clone detection methods focus on extracting the features from source code directly while ignoring the potential underlying connections among different code samples. To achieve scalable and accurate code clone detection, we consider extracting these connections to build "bridges" between code samples (*i.e.,* global graph) and using them to detect code clones. Specifically, we mainly address two challenges in our paper.

- *How to build the global graph from source code and represent it properly to retain code details?*
- *How to utilize the global graph across different source code samples to efficiently and accurately detect code clones?*

To tackle the first challenge, we choose keyword tokens along with side information as the individual information to represent the source code samples. In detail, since the programming language of our experimental dataset is Java, we leverage the reserved words of Java as keyword tokens. Meanwhile, to better capture the code details, we also extract another kind of information (*i.e.,* side information) such as the maximum depth of brackets and the number of loops. Because the extraction of keywords and side information can be achieved by simple lexical analysis, we can complete scalable code clone analysis.

To address the second challenge, we use keywords and side information as the "bridge" to connect different code samples. Specifically, we build a global sample graph to represent the underlying connections between all samples. Each node in the graph represents a code sample or a kind of individual information (*i.e.,* keywords or side information). Each edge indicates whether a code sample contains such individual information. After constructing the global graph, we perform a node embedding technique on it to convert all code samples into corresponding vector representations. Given generated vectors, we can calculate the *cosine similarity* of two samples and quickly identify whether they are clone pairs.

We implement a prototype system, *Gitor*, and evaluate it on a widely used dataset, namely BigCloneBench [1, 46]. Our evaluation results show that *Gitor* is superior to six state-of-the-art comparative systems including *SourcererCC* [44], *CCFinder* [20], *Nicad* [42], *Deckard* [17], *CCAligner* [52], *Oreo* [43], *LVMapper* [56], and *NIL* [34]. Moreover, we can also combine the code sample representation vector generated by *Gitor* with feature vector obtained from source code directly by three traditional vector-based tools (*i.e.,* word2vec [33], doc2vec [27], and code2vec [5]), the results indicate that the combination make them detect more clones with higher F1. Finally, we examine the scalability of *Gitor* on various sizes of code.

Evaluation results report that *Gitor* has the ability to analyze 100 million lines of code, with the shortest execution time compared to *SourcererCC*, *CCFinder*, *Nicad* , *Deckard*, *CCAligner*, *Oreo*, *LVMapper*, and *NIL*.

In summary, this paper makes the following contributions:

- We propose a novel method to detect code clones by building a global sample graph using keywords and side information. The constructed global graph can capture the underlying connections between different source code samples.
- We design a prototype system namely *Gitor* and conduct evaluations on a widely used dataset (*i.e.,* BigCloneBench [1]). Experimental results suggest that *Gitor* outperforms *SourcererCC*, *CCFinder*, *Nicad* , *Deckard*, *CCAligner*, *Oreo*, *LVMapper*, and *NIL* and *Gitor* is adept at handling the challenges posed by the big scale of code.

Paper organization. The remainder of the paper is organized as follows. Section 2 explains the background and motivation. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 discusses the future work. Section 6 describes the related work. Section 7 concludes the present paper.

2 DEFINITION AND MOTIVATION

2.1 Definitions

The paper utilizes the well-accepted definitions of code clones and clone types as follows:

```
public static int fib(int i){
    int f1=0, f2=1, c=0;
    if((i == 0) || (i == 1)) return i;
    for (int j =2; j<=i; j++){
        c=f1+f2; f1=f2; f2=c;
    }
    return c;
}
```

Listing 1: Original (Func #0)

```
public static int fib(int i){
    int f1=0, f2=1, c=0;
    if((i == 0) || (i == 1)) return i;
    for (int j =2; j<=i; j++){
        c=f1+f2; f1=f2; f2=c;
    }
    return c;
}
```

Listing 2: Type-1 (Func #1)

```
public static int fib(int num){
    int f1=0, f2=1, c=0;
    if((num == 0) || (num == 1)) return num;
    for (int j =2; j<=num; j++){
        c=f1+f2; f1=f2; f2=c;
    }
    return c;
}
```

Listing 3: Type-2 (Func #2)

```
public static int calFib(int num){
    int fib1=0, fib2=1;
    int t=0;
    if((num == 1) || (num == 0)) return num;
    for (int k =2; k<=num; k++){
        t=fib1+fib2; fib1=fib2; fib2=t;
    }
    return t;
}
```

Listing 4: Type-3 (Func #3)

```
1   public static long calFib(long number){
2       long f1=0, f2=1, c=0;
3       switch(number){
4           case 0:
5               return 0;
6           case 1:
7               return 1;
8           default:
9               break;
10      }
11      while(number>=2){
12          c=f1+f2; f1=f2; f2=c;
13          number--;
14      }
15      return c;
16  }
```

Listing 5: Type-4 (Func #4)

2.1.1   Clone Types In our paper, we use the following widely used definitions [8, 41] of code clone types.

- **Type-1 (textual similarity)**: Identical code fragments, except for minor differences in white-space, layout, or comments.
- **Type-2 (lexical similarity)**: Structurally identical code fragments, in addition to Type-1 clone differences, there might be some differences in identifier names and literal values.
- **Type-3 (syntactic similarity)**: Modified similar code fragments that differ at the statement level. Besides the Type-1 and Type-2 clone, the fragments might have statements added, modified and/or removed compared to each other.
- **Type-4 (semantic similarity)**: Dissimilar code fragments with the same functionality but implemented in a syntactically different way.

To elaborate on different types of clones, listings 1 to 5 present examples from Type-1 to Type-4 clones. The original code is used to compute the Fibonacci number given the order. The Type-1 clone (starting in line #11) is identical to the original code. The Type-2 clone (starting in line #21) 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 #31) is syntacally similar but differs at the statement level. The first line in Type-3 (line #42) is totally different from the origin code. 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.

2.2   Motivation

To illustrate the key insight of our proposed method, we leverage Fun #0 and its corresponding type 3 and type 4 clones (*i.e.,* Fun #3 and Fun #4) as our analysis targets. As shown in Listing 1, those examples are all used to calculate the Fibonacci number of the given order. According to the definition of code clone, the clone pair $Fib_0.java$ and $Fib_3.java$ are classified as Type-3 clone (*i.e.,* syntactic similarity) since they differ at the statement level. The clone pair $Fib_0.java$ and $Fib_4.java$ are classified as a Type-4 clone (*i.e.,* semantic

clone) because they have syntactically dissimilar code to implement the same functionality.

2.2.1   SourcererCC We start with illustrating how the widely used clone detection tool *SourcererCC* [44] (*i.e.,* one of the state-of-the-art token-based clone detectors) detects possible clone pairs by calculating the similarity of each pair. *SourcererCC* [44] utilizes the *Overlap* of two source code blocks to compute the similarity since it intuitively captures the notion of overlap among different code blocks. For example, given two code blocks C_1 and C_2, the overlap similarity $S(C_1, C_2)$ is calculated as the number of tokens shared by C_1 and C_2.

$$S(C_1, C_2) = |C_1 \bigcap C_2|$$

Given the threshold θ and the maximum number of tokens $T = max(|C_1|, |C_2|)$, a pair of code blocks is considered as a clone pair when the ratio of overlap similarity and T is greater than the threshold θ.

$$\frac{S(C_1, C_2)}{T} \geq \theta$$

2.2.2   Keywords To achieve a more accurate clone detection, we need to extract reliable information to represent the source code, preferably some kind of global information that can reflect the connections between source code samples rather than analyze the information from code samples individually. So, we will extract the individual information from each code sample by extracting keywords and build a global sample graph that connects all code samples.
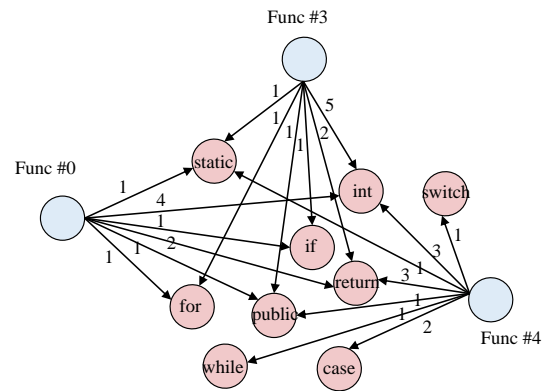
Figure 1: A global graph of Func#0, Func#3, and Func#4.

We first tokenize the source code to get the sequences of tokens of Func#0, Func#3, and Func#4. Then we choose only the reserved words in Java as the keywords instead of all tokens to represent code samples since the reserved words are used in all Java source code samples. After extracting keywords from the above code samples, we construct a weighted directed graph with the frequency of keywords as weight as illustrated in Figure 1. Each blue node represents a code sample, each red node represents a keyword, and the weight from blue nodes to red nodes is the frequency of keyword in the corresponding code sample.

2.2.3   Node Embedding In order to obtain the similarity of Fun#0, Fun#3, and Fun#4 in Figure 1, we first use node embedding methods to convert them into their vector representations. These node
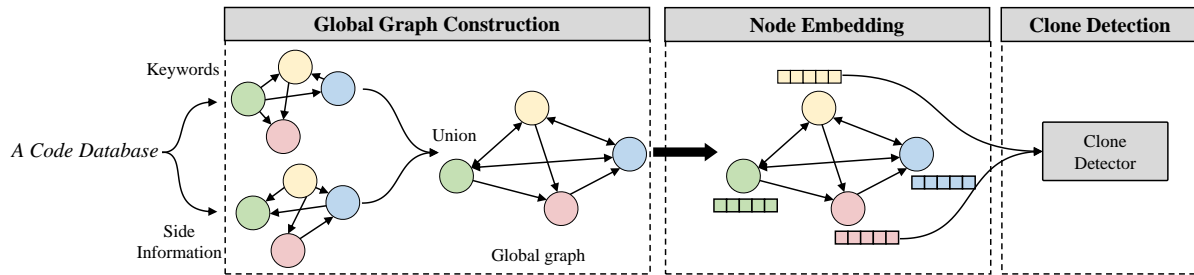
Figure 2: System overview of *Gitor*

embedding algorithms typically aim to capture the structural information and relationships between nodes and covert the graph structure and node attributes into representation vectors, which can preserve the underlying similarity among nodes [14, 36]. In this paper, we mainly consider two different embedding methods, namely node2vec [14] and ProNE [58], since they support the embedding of weighted graph. However, to achieve the scalability, we choose ProNE as our embedding method because it is faster, more scalable, and more effective than node2vec [30]. So, we use ProNE [58] to map the code samples into vectors, which can be used to calculate the similarity among different functions.

2.2.4 Similarity Evaluation We calculate the similarity of above two code blocks using the method mentioned in *SourcererCC*. It shows that the number of tokens in Func#0 and Func#3 is 73 and 74, respectively. Then the same tokens shared by Fun#0 and Func#3 are obtained for computing the overlap similarity. We observe that there are 18 same tokens shared by these two code blocks, which means the overlap similarity of Func#0 and Func#3 is 18/74=0.24. If similarity threshold in *SourcererCC* is set to 70%, which means that *SourcererCC* reports two methods as a clone pair only when the ratio of number of shared tokens and maximum number of tokens of them is larger than 70%. In this case, *SourcererCC* will cause a false negative by reporting Func#0 and Func#3 as a none clone pair. Also, the similarity between Func#0 and Func#4 according to *SourcererCC* is 0.23. Now, we conduct node embedding on the graph shown in Figure 1, and then we get the vector representations of Fun#0, Fun#3, and Fun#4, which are used to calculate the similarity. After node embedding, the similarity between Func#0 and Func#3 is 0.99 and the similarity between Func#0 and Func#4 is 0.65, suggesting that the similarity among these clone pairs is significantly improved.

Based on the observation, we propose a novel code clone detection framework by considering the global relationships between different functions.

3 APPROACH

In this section, we introduce our proposed system, namely *Gitor*.

3.1 System Overview

As shown in Figure 2, *Gitor* consists of three main phases: *Global Graph Construction*, *Node Embedding*, and *Clone Detection*.

- **Global Graph Construction**: We first apply lexical analysis to extract the *individual information* including *keywords* and *side information* of a code sample with corresponding

weights. Then a global sample graph is built by using these information where each node represents a sample or a type of *individual information*.
- **Node Embedding**: Given a graph of the whole code base, we use a node embedding technique on the global graph and output the vectors of each node with chosen dimension. The input is a weighted global sample graph, and the outputs are vectors of all samples in the code base.
- **Clone Detection**: After the generation of vectors, we have two ways to detect potential clone pairs. First, we can simply calculate the cosine similarity of a pair of samples to identify code clones. Second, we can combine *Gitor* with other vector-based clone detection methods, which will boost the performance of clone detection.

3.2 Global Graph Construction

3.2.1 Individual Information Extraction. In this paper, we aim to combine the connection capture capability of graph embedding methods with the scalability of token-based methods. Therefore, we first conduct tokenization on the source code to extract the keywords and side information from the source code. Since our experiments are done on the BigCloneBench dataset [46], we tokenize the *java* source code based on a java parse tool, namely *javalang* [2]. We choose the *Java* reserved words as *keywords* along with five types of *side-information* as *individual information*. For example, take the Func #0 from Listing 1, the keywords and corresponding weights of Func #0 is *{public: 1, static: 1, int: 4, if: 1, return: 2, for: 1}*. Moreover, the five different types of *side information* are as follows:

- Maximum Nesting Depth of the Curly Brackets (*MNDCB*): The number of maximum depth of nested curly brackets. For example, the *MNDCB* of Func #0 is 2.
- Maximum Number of Parallel Curly Brackets (*MNPCB*): The number of maximum parallel curly brackets with depth 2. For example, the *MNPCB* of Func #0 is 1.
- Loop-Repetition Information (*LRI*): The number of loop functions used in the code, including for-loop and while-loop. For example, the *LRI* of Func #0 is 1.
- Flow-Control Information (*FCI*): The number of flow-control functions used in the code, including if-else and switch-case. For example, the *FCI* of Func #0 is 1.
- Numerical Declaration Information (*NDI*): The number of numeric variables declared in the code, including int, double, float, byte, short and long declaration. For example, the *NDI* of Func #0 is 4.

For *MNDCB*, we utilize a depth counter, adjusting it with every encountered curly bracket—incrementing for each opening and decrementing for each closing, subsequently noting the peak depth. For *MNPCB*, we discern parallel curly brackets at distinct depths by counting sequential opening and closing pairs. Loop-related tokens like for and while contribute to the *LRI* tally. Similarly, flow-control tokens such as if and switch are counted for *FCI*. The *NDI* is ascertained by enumerating numeric variable declaration tokens like int and double. This token-based methodology offers a nuanced perspective on the code's structure and semantics.

In this paper, we chose these types of *side information* because they can provide additional information about the structure and complexity of the code samples and can help to identify clones that might not have been detected by keyword-based methods alone. Also, using such information can improve the scalability of code clone since it reflects the code structure without processing and comparing the whole code sample. For example, *LRI* represents the number of for-loop declarations and while-loop declarations since they have similar functionality in Java, and the substitution of these two loop functions for each other is often found in clone samples. Also, the types of *side information* might differ according to the programming language of code samples. For instance, the functionality of curly brackets in Python is different from that of curly brackets in Java. So, the types of *side information* should be carefully selected according to the different programming languages of code samples. Moreover, different combinations of *side information* may affect the detection performance slightly and the main goal of this paper is not about finding the optimal combination of different types of *side information*, so we will use all of these five types in the following paper.

In our study on code clone detection, we discern the importance of both *keywords* and *side information*. Keywords are the reserved words directly extracted as tokens from the code, serving as foundational markers of code content. On the other hand, *side information* delves deeper, encapsulating structural metadata such as bracket depth and loop count. To maximize the potential of both elements, we construct individual graphs for each. These are then amalgamated into a singular, comprehensive global graph, establishing diverse connections between code samples. This method not only merges lexical content with structural nuances but also provides a more robust framework, enhancing the precision in detecting code clones by highlighting intricate relationships and similarities between code snippets.

After extracting *keywords* and *side information*, we can get a sequence of tokens with corresponding weight, which is the frequency in this case.

3.2.2 Global Graph Construction. Nowadays, the graph is an important kind of representation to encode relation structure, which is used in many domains (*i.e.,* social networks, citation networks, function call diagrams, etc.). The nodes and edges can represent the objects and relationships respectively. Evaluation of similarity between two nodes based on the graph structure has a wide range of applications, such as social networks analysis, knn, graph clustering, etc. Therefore, instead of simply comparing the similarity using the overlap *keywords* and *side information* of two samples, we first use *keywords* extracted from the last step to build a graph
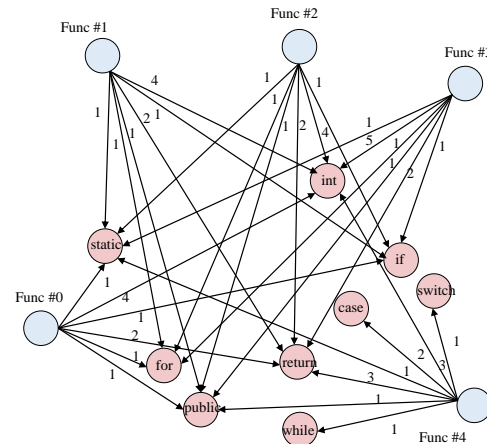

Figure 3: A global graph of Func #0-4 constructed by keywords.
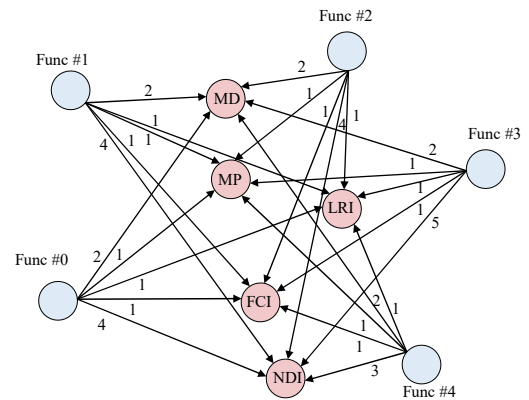
Figure 4: A global graph of Func #0-4 constructed by side information.

representing the whole code base. To better illustrate this phase in *Gitor*, we choose the samples in Listing 1 as an example and present a clearer description in Figure 3. As illustrated in Figure 3, the blue nodes represent Function #0-4 and the red nodes represent the keywords. The weight on the directed edges from functions to keywords is the frequency of keywords appearing in the functions.

To better capture the program details of source code, we also select another kind of information. Specifically, we construct another graph using *side information* defined above, where the blue nodes represent function as well, but the red nodes represent *side information* types, and the weight is the count of corresponding *side-information*, as illustrated in Figure 4. After obtaining two graphs using *keywords* and *side information*, we merge them by merging the nodes that have the same labels to build one larger global sample graph which will be embedded and used to calculate the similarity of any two functions.

In short, the input of graph construction is a code database containing many code samples (*i.e.,* functions) and the output is a large global sample graph.

Figure 5: Two applications of *Gitor*. The first is to detect code clones using global features and the second is to combine global features with individual features to detect code clones.
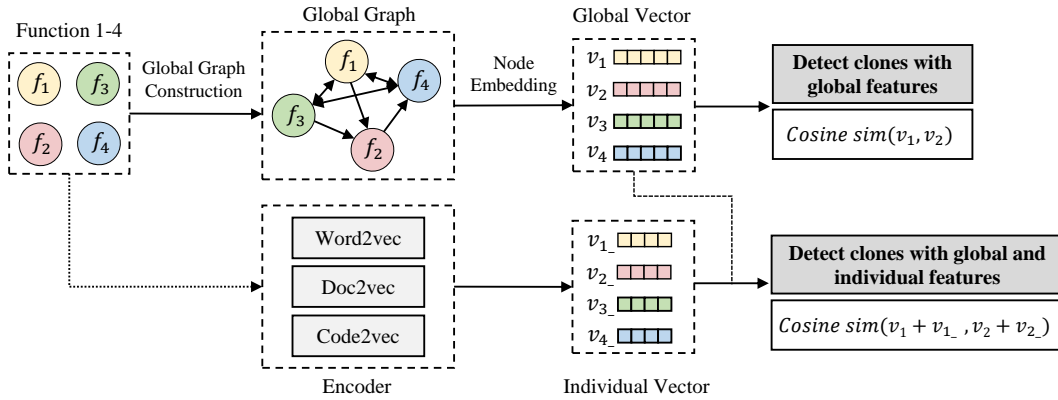
3.3 Node Embedding

Graph is a commonly used type of information representation in complex systems and can represent many complex relationships in real-life scenarios, such as social networks [35], crime networks [15], traffic networks [61], etc. Graph analysis is used to dig deeper into the intrinsic features of graph data, however, since the graph is non-Euclidean data, traditional data analysis methods generally have high computational effort and spatial overhead. Graph embedding is an effective method to solve the graph analysis problem, which transforms the original graph data into a low-dimensional space and preserves key information, thus improving node classification, link prediction, and graph analysis. It can improve the performance of the tasks like node classification, link prediction, and node clustering by retaining key information from the graph. Deep Learning-based methods among different graph embedding methods have demonstrated promising results due to their capability of automatically discovering underlying connections and identifying useful representations from the complex graph structures. For instance, deep learning with random walk (*i.e.,* DeepWalk [39] and Node2vec [14]) can leverage the neighborhood structure by sampling paths on the graph automatically.

Graph embedding methods are feature representation learning methods, exploiting the graph structure to transform each node of the graph into a low-dimensional vector while preserving neighborhood similarity, semantic information, and community structure among nodes [10]. The obtained vector representations can be utilized by a wide range of tasks such as link prediction [45], node classification [48]. So, the node embedding method can capture the global connections among nodes in the graph, which means it can capture the underlying similarity property among functions from a holistic perspective than analyzing them individually. In this paper, we choose ProNE [58] since it is a fast and effective method that combines the benefits of various embedding methods while remaining time-efficient [30]. Moreover, we conduct the embedding with different vector sizes (*i.e.,* d = 16, 32, 64, 128) on our chosen dataset, BigCloneBench[46], to find the optimal embedding dimension for clone detection.

In brief, the input of node embedding is the graph constructed before, and the outputs are vectors of all nodes in the graph with the pre-defined dimension.

3.4 Clone Detection

After collecting the vectors of all functions, we have two applications to use them for code clone detection. The first is to apply code clone detection by directly computing the similarity of these vectors. More importantly, these vectors can also be used to enhance the detection effectiveness of other vector-based code clone detectors. Figure 5 describes an example of the two applications of *Gitor*.

3.4.1 Application 1: Detect clones with global features. Cosine similarity is a commonly used metric, which measures similarity between two vectors, especially in high-dimension space. It measures similarity as the cosine of the angle between two vectors. Two similar vectors are expected to have a small angle between them. The cosine similarity of two vectors x and y is defined as follows:

$$\cos \theta = \frac{\Sigma_i^d x_i y_i}{\sqrt{\Sigma_i^d x_i^2} \sqrt{\Sigma_i^d y_i^2}}$$

Our first application is simply calculating the cosine similarity between two vectors, If the similarity is greater than a certain threshold (*e.g.,* 0.7), they are identified as a clone pair, as illustrated in Figure 5.

3.4.2 Application 2: Detect clones with global and self features. We choose the graph as the representation of the whole code base since the natural structure of the graph can capture the underlying global connections among different code samples better than analyzing them individually. Instead of using the *Gitor* alone, we can combine it with other self-features-based (*i.e.,* individual-features-based) methods, which is generated by individual analysis on each code sample. Nowadays, there are numerous vector-based code clone detection methods, such as [53] and [57], and the current methods all focus on detecting clones utilizing individual features. In other words, our proposed *global graph* based clone detection method

Table 1: Detection performance of *Gitor* with different cosine similarity thresholds.

Cosine = 0.60	Keywords				Side Information				Both			
	16	32	64	128	16	32	64	128	16	32	64	128
T-1 Recall	1	1	1	1	1	1	1	1	1	1	1	1
T-2 Recall	1	1	1	1	1	1	1	1	1	1	1	1
VST-3 Recall	0.986	0.972	0.991	0.990	1	1	1	0.999	0.988	0.989	0.993	0.992
ST-3 Recall	0.858	0.822	0.868	0.847	0.995	0.995	0.995	0.994	0.843	0.829	0.922	0.906
MT-3 Recall	0.638	0.617	0.560	0.568	0.959	0.958	0.956	0.952	0.672	0.683	0.780	0.690
Type-4 Recall	0.168	0.186	0.099	0.089	0.606	0.595	0.559	0.550	0.162	0.199	0.204	0.114
Precision	0.858	0.874	0.926	0.919	0.687	0.694	0.712	0.711	0.867	0.903	0.912	0.936
F1	0.651	0.655	0.625	0.621	0.747	0.749	0.752	0.750	0.659	0.684	0.715	0.669
Cosine = 0.70	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**
T-1 Recall	1	1	1	1	1	1	1	1	1	1	1	1
T-2 Recall	1	1	1	1	1	1	1	1	1	1	1	1
VST-3 Recall	0.979	0.930	0.958	0.946	1	1	1	0.999	0.964	0.954	0.988	0.989
ST-3 Recall	0.811	0.753	0.777	0.761	0.995	0.995	0.994	0.993	0.755	0.750	0.840	0.803
MT-3 Recall	0.537	0.486	0.401	0.402	0.945	0.950	0.945	0.943	0.548	0.537	0.601	0.492
Type-4 Recall	0.101	0.099	0.045	0.039	0.498	0.486	0.446	0.436	0.097	0.103	0.090	0.046
Precision	0.905	0.922	0.955	0.958	0.728	0.730	0.738	0.739	0.917	0.939	0.951	0.964
F1	0.612	0.597	0.557	0.554	0.748	0.747	0.742	0.740	0.613	0.616	0.637	0.589
Cosine = 0.80	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**
T-1 Recall	1	1	1	1	1	1	1	1	1	1	1	1
T-2 Recall	1	1	1	1	1	1	1	1	1	1	1	1
VST-3 Recall	0.936	0.871	0.918	0.907	1	0.999	0.999	0.999	0.897	0.939	0.963	0.931
ST-3 Recall	0.721	0.655	0.620	0.634	0.995	0.994	0.994	0.993	0.681	0.672	0.700	0.666
MT-3 Recall	0.425	0.312	0.233	0.231	0.916	0.911	0.903	0.898	0.422	0.370	0.378	0.290
Type-4 Recall	0.051	0.040	0.015	0.013	0.343	0.329	0.296	0.287	0.048	0.040	0.027	0.014
Precision	0.954	0.960	0.981	0.982	0.769	0.766	0.770	0.769	0.954	0.973	0.979	0.985
F1	0.563	0.518	0.480	0.480	0.729	0.723	0.716	0.712	0.558	0.541	0.541	0.504

can be combined with current vector-based detection methods to boost the performance of clone detection.

In this paper, we choose Doc2vec [27], Word2vec [33], and Code2vec [5] as the vector-based detection methods [53, 57]. Word2vec [33] and Doc2vec [27] are well-known natural language processing baseline methods for extracting feature vectors from source code. Code2vec [5] parses a code fragment into an AST path collection. To predict the method name, the core idea is to use a soft-attention mechanism on the paths and aggregate all vector representations into a single vector. This combined method is illustrated in Figure 5. The global vector and individual vector are added to calculate the similarity between two code samples.

4 EXPERIMENTS

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

- *RQ1: What is the effectiveness of Gitor in detecting different types of code clones when used alone?*
- *RQ2: How does the use of global features contribute to the effectiveness of boosting individual-features-based clone detection?*
- *RQ3: What is the effectiveness of Gitor compared to other state-of-the-art code clone detectors?*
- *RQ4: What is the runtime performance of Gitor compared to other state-of-the-art clone detectors?*

4.1 Experimental Settings

4.1.1 Dataset We conduct our evaluations on the dataset: Big-CloneBench [1], which consists of more than 8,000,000 labeled clone pairs from 25,000 systems. The code granularity of clone pairs in BigCloneBench [1] is function-level, and each clone pair is manually assigned a corresponding clone type. Type-3 and Type-4 types are usually further divided into four subcategories based on their syntactical similarity score, as follows: i) *Very Strongly Type-3* (VST3) with a similarity between [0.9, 1.0), ii) *Strongly Type-3* (ST3) with a similarity between [0.7, 0.9), iii) *Moderately Type-3* (MT3) with a similarity between [0.5, 0.7), and iv) *Weakly Type-3/Type-4* (WT3/T4) with a similarity between [0.0, 0.5). The total number of these clone pairs used in our experiments is 8,446,574 including 8,139 Type-1 clones, 3,292 Type-2 clones, 4,577 VST3 clones, 3,469 ST3 clones, 7,606 MT3 clones, and 8,424,068 WT3/T4 clones. In the following experiment results, we use Type-4 (T4) to denote WT3/T4.

4.1.2 Implementation For *individual information* extraction, we leverage a java parser (*i.e., javalang* [2]) to extract the keywords and *side information* from source code samples. For global sample graph construction, we use a Python library, *networkx* [3], to build a weighted and directed graph. For node embedding, we employ a

widely used embedding method, *ProNE* [58], to conduct the embedding of the graph. The output of embedding is a series of vectors of all nodes in the graph.

We also select certain state-of-the-art code clone detection tools as our comparative systems, including *SourcererCC* [44], *CCFinder* [20], *NiCad* [42], *Deckard* [17], *CCAligner* [52], *Oreo* [43], *LVMapper* [56], and *NIL* [34]. All experiments are conducted on a server with Intel Xeon E5-2678 v3 @ 2.50GHz, 32 Gig-bytes memory, GeForce RTX 2080 TI Graphics Card and Ubuntu 18.04.5 LTS.

4.1.3 Metrics We make use of the following widely used metrics to measure the detection performance of *Gitor*. Precision is defined as $P = TP/(TP + FP)$. Recall is defined as $R = TP/(TP + FN)$. F1 is defined as $F1 = 2 * P * R/(P + R)$. Among them, *true positive* (TP) represents the number of samples correctly classified as clone pairs, *false positive* (FP) represents the number of samples incorrectly classified as clone pairs, and *false negative* (FN) represents the number of samples incorrectly classified as non-clone pairs.

4.2 RQ1: Effectiveness of Gitor Used Alone

To examine the capability of *Gitor* on clode clone detection, we conduct experiments from two perspectives, one is testing the performance of *Gitor* alone, and another is testing the performance of *Gitor* combined with other individual-features-based methods. In this part, we focus on checking the ability of *Gitor* alone. Specifically, we select different cosine similarity thresholds (*i.e.,* 0.6, 0.7, and 0.8) and different dimensions (*i.e.,* 16, 32, 64, and 128) of node embedding vectors to commence our evaluations. The results are illustrated in Table 1, including the recall, precision, and F1 scores of our experiment on BigCloneBench dataset. As for the measurement of precision, similar to other previous works [43, 52], we randomly sample 400 clone pairs from clone reports in each tool and conduct manual analysis to validate them. Each clone pair is checked independently by two experts. If there is a conflict, a final decision will be made after discussion with another expert. The principle rule for judging is based on the overall similarity between the two clone fragments and on whether they perform similar functionality.

Through the results in Table 1, we find several interesting phenomena. First, when the similarity threshold is different, the detection performance of *Gitor* is also different. Basically, the larger the threshold, the higher the precision, but the lower the recall. It is reasonable because the larger the threshold, the higher the similarity of the detected clones, and the higher the similarity is, the greater the probability of clones. But at the same time, some pairs whose similarity is slightly lower than the threshold will be filtered out, resulting in lower recall. Second, when the vector dimensions are different, the detection performance of *Gitor* is also different. This is normal because the dimensions of the vectors are different, the degree of retention of graph information will also be different. Basically, when we combine *keywords* with *side information*, the larger the dimension of the vector, the higher the precision. Third, the features obtained when selecting *keywords* to construct a sample graph are more accurate than when selecting *side information*. In other words, when using *keywords* to construct the global graph, the precision of *Gitor* is higher than when selecting *side information*. This is because *keywords* represent the key tokens in the programming language, these key tokens are not allowed to

be changed, and different key tokens describe different program information. *Gitor* can preserve more program semantics when all keywords are considered. Forth, after combining *keywords* with *side information*, *Gitor*'s precision is mostly improved. It shows that the combination of the two information allows *Gitor* to retain more program semantics. At this time, when the vector dimension is 64, the average F1 under the three thresholds (*i.e.,* 0.6, 0.7, and 0.8) is the highest.

Based on the above findings, we suggest that if researchers want to detect more clones, they can set the threshold to 0.6 when using *Gitor* with *keywords* and *side information*. In addition, if researcher like to detect clones with higher accuracy, they can set the threshold to a higher value, such as 0.7 or 0.8.

Table 2: Detection performance of individual-features-based detectors

Method	Individual-features-based detector		
	Doc2Vec	W2V-avg	Code2Vec
Type-1 Recall	1	1	1
Type-2 Recall	0.95	1	1
VST-3 Recall	0.86	0.99	0.998
ST-3 Recall	0.57	0.85	0.995
MT-3 Recall	0.21	0.53	0.979
Type-4 Recall	0.02	0.11	0.928
Precision	0.98	0.98	0.619
F1	0.47	0.63	0.753

Table 3: Detection performance of individual-features-based detectors combined with *Gitor*

Method	With Gitor (Ours)		
	Doc2Vec	W2V-avg	Code2Vec
Type-1 Recall	1	1	1
Type-2 Recall	1	1	1
VST-3 Recall	0.947	0.973	0.998
ST-3 Recall	0.671	0.817	0.996
MT-3 Recall	0.421	0.573	0.979
Type-4 Recall	0.042	0.131	0.939
Precision	0.971	0.981	0.65
F1	0.558	0.649	0.778

4.3 RQ2: Combination with Other Individual Features-based Methods

In this part, we pay attention to the effectiveness when *Gitor* is combined with other detection methods. Since our system is purely based on global features, we first want to explore how would it contribute to current individual-features-based detection methods. In order to check the effectiveness of detection using the combination of global features and individual features, we pick several widely used methods [53, 57], which have been proved effective on clone detection, then we test the effectiveness when they are combined with *Gitor*.

Table 4: Detection performance of *SourcererCC* [44], *CCFinder* [20], *NiCad* [42], *Deckard* [17], *CCAligner* [52], *Oreo* [43], *LVMapper*[56], *NIL*[34], and *Gitor* on detecting different types of code clones.

Tool	SourcererCC	CCFinder	NiCad	Deckard	CCAligner	Oreo	LVMapper	NIL	Gitor
Type-1 Recall	1	1	1	0.6	1	1	0.99	0.99	**1**
Type-2 Recall	0.97	0.93	0.99	0.58	0.99	0.99	0.99	0.96	**1**
Very Strongly Type-3 Recall	0.93	0.62	0.98	0.62	0.97	1	0.98	0.93	**0.988**
Strongly Type-3 Recall	0.6	0.15	**0.93**	0.31	0.7	0.89	0.81	0.67	0.84
Moderately Type-3 Recall	0.05	0.01	0.008	0.12	0.1	0.3	0.19	0.1	**0.601**
Type-4 Recall	0	0	0	0.01	-	0.007	-	-	**0.09**
Precision	0.978	0.72	**0.99**	0.348	0.8	0.895	0.58	0.94	0.951

In this experiment, we choose Doc2Vec [27], Word2Vec [33], and Code2Vec [5] as the individual-features-based detection methods [53, 57]. We define the average vectors of Word2Vec as W2V-avg, and the Doc2Vec extends the word vectors to entire document vectors. The Code2Vec can embed the entire code sample into a single vector. We choose *cosine similarity* as the similarity metric in this part of experiments, and we use the default parameters for Doc2Vec, Word2Vec, and Code2Vec.

To evaluate the detection performance of these three methods, we test them on the BigCloneBench dataset. We use three methods to get the embeddings of all code samples and compare the *cosine similarity* to detect possible clone pairs, where we set the similarity threshold as 0.9, and embedding dimension as 128 since these tools reach their best performance under this setting [57]. Table 2 shows the detection results including recall, precision and F1-score on the BigCloneBench dataset. Then we combine the vectors generated by the above methods with *Gitor* generated vectors and conduct the similarity comparison on the BigCodeBench dataset, where the similarity threshold is set to 0.9 as well and the dimension of *Gitor* is 32 since *Gitor* performs the best with dimension as 32 when the similarity threshold set to 0.9. The results are illustrated in Table 3, where we can see that the overall detection performance, including recall, precision, and F1-score, is significantly improved compared to the original results, so it suggests that *Gitor* can boost the effectiveness of other individual-features-based detection methods.

In short, the *Gitor* is not only effective when used alone, but also able to boost the performance of other individual-features-based detection methods.

4.4 RQ3: Comparative with Other Detectors

In order to evaluate *Gitor*'s performance comprehensively, we compare the performance of *Gitor*'s clone detection against the latest versions of several publicly available clone detection tools, such as *SourcererCC* [44], *CCFinder* [20], *NiCad* [42], *Deckard* [17], *CCAligner* [52], *Oreo* [43], *LVMapper* [56], and *NIL* [34]. Since most of traditional code clone detection tools (*e.g., SourcererCC* [44] and *NiCad* [42]) select 0.7 as their thresholds to identify code clones, we also choose 0.7 as the threshold to commence our comparative evaluations. Through the results in Table 1, we observe that *Gitor* can maintain the best overall performance (*i.e.,* F1) when the dimension of node embedding vectors is 64. Therefore, we use the corresponding detection results as the comparative performance of *Gitor, SourcererCC, CCFinder, Nicad , Deckard, CCAligner, Oreo,*

LVMapper, and *NIL*, where the recall numbers are summarized per clone category. As Table 4 shows, *Gitor* outperforms every other tool on most of the clone categories, except for ST3. Although NiCad performs the best on ST3, *Gitor*'s performance on ST3 clone is still quite comparable to the state-of-art since there is only a 9 percent difference. The recall results are promising since they suggest that, in addition to recognizing easier-to-find clones like T1, T2, and VST3, *Gitor* also detects clones that other tools miss. In comparison to other methods, where Oreo's highest recall is 0.3, 0.601 recall in the MT3 category is a significant improvement. Table 4 also shows the precision results of all tools. The precision of *Gitor* is 0.951, and only SourcererCC and NiCard perform marginally better than *Gitor* (by 5 percent).

The recall and precision experiments show that *Gitor* is a reliable and accurate clone detector that can detect Type-1, Type-2, and Type-3 clones efficiently and detect part of Type-4 clones. To address this issue, in the future we will improve *Gitor* with better chosen individual information to detect Type-4 clones more effectively.

4.5 RQ4: Scalability

In this section, we pay attention on the runtime performance of *Gitor*. As mentioned before, scalability is an important requirement for clone detection methods, and *Gitor* is designed as a scalable clone detection system. So, we will evaluate the efficiency and demonstrate the scalability of *Gitor* in two parts: training efficiency and classification efficiency.

Dataset for scalability experiments: We use the whole dataset of BigCloneBench (*i.e.,* IJaDataset [4]), which is a widely used dataset containing about 250 million lines of Java source code mined from SourceForge and Google Code. The full IJaDataset and its subsets are often used for evaluating execution time and scalability of clone detection tools [28, 44, 52]. We test *Gitor* using inputs with different sizes generated from this dataset.

Different sizes for scalability experiments: Execution time primarily depends on the size of the input in terms of the number of lines of code (LOC) needed to be processed and classified by the system. So, we build the inputs with varying convenient sizes (*i.e.,* 1K, 10K, 100K, 1M, 10M, and 100M LOC) by randomly selecting samples from IJaDataset.

Results: The execution is finished on a machine with Intel Xeon E5-2678 v3 @ 2.50GHz 12 cores CPU, 32GB of memory, GeForce RTX 2080 TI Graphics Card, and system is Ubuntu 18.04.5 LTS. For *Gitor*, it mainly consists of two phases, the first it to apply node

Table 5: Runtime performance of *SourcererCC* [44], *CCFinder* [20], *NiCad* [42], *Deckard* [17], *CCAligner* [52], *Oreo* [43], *LVMapper*[56], *NIL*[34], and *Gitor*.

LOC	SourcererCC	CCFinder	NiCad	Deckard	CCAligner	Oreo	LVMapper	NIL	Gitor
1K	3s	2s	1s	1s	1s	1s	1s	1s	**0.03s**
10K	5s	5s	2s	4s	2s	3s	-	-	**0.18s**
100K	7s	10s	5s	32s	3s	6s	-	-	**1.26s**
1M	37s	39s	12s	27m12s	11m52s	4m34s	29s	**10s**	13.10s
10M	12m21s	6m30s	19m49s	Killed	29m48s	36m6s	13m 38s	**1m 38s**	2m11s
100M	12h27m	9h49m	Killed	-	Killed	1d13h46m	17h 23m 39s	1h 38m 29s	**1h7min**

Killed means the tool fails to parse the code or report out-of-memory errors, "-" means no such data in previous study

embedding to extract all functions' vectors, and then these vectors will be used to compute cosine similarity one by one. In practice, it takes little time to complete the first phase (*i.e.,* 20 minutes for 100M LOC). However, when the code size becomes large, the number of functions will also be large, resulting in a massive number of code pairs to be analyzed. To mitigate the issue, we adopt matrix computation to calculate the similarity of all code pairs, where GPU is used to accelerate the computation process. The runtime of *Gitor* is the total time of two phases included. The runtime performance of all the above tools and corresponding LOC are listed in Table 5, which shows that *Gitor* outperforms the seven state-of-the-art clone detection tools in all sizes of inputs while *Gitor* is a bit slower than *NIL* in 1MLOC and 10MLOC size, but *Gitor* is still more efficient than the state-of-the-art detector *NIL* when it comes to larger size, 100MLOC, in this case.

In conclusion, *Gitor* is eight times faster than the token-based detection tool *CCFinder* [20] with the input size of 100 million LOC, which means it is highly scalable.

4.6 Summarization

Our experimental results demonstrate *Gitor*'s effectiveness as a code clone detection method. It achieves optimal accuracy using a combination of keyword and side information features (RQ1). *Gitor* improves the performance of individual feature-based detectors when used jointly (RQ2). In comparative evaluations, *Gitor* attains higher recall than eight state-of-the-art tools on the BigCloneBench dataset, with precision comparable to top techniques (RQ3). Moreover, *Gitor* analyzes 100 million lines of code efficiently in just 1 hour, and is the fastest tool on large code bases, running 100X faster than CCFinder (RQ4). In summary, through extensive evaluations, our results consistently highlight *Gitor*'s strengths in terms of effectiveness, enhancement capability, superior accuracy over current methods, and scalability to large code bases.

5 DISCUSSION

Why Gitor outperforms the other approaches. First, currently existing clone detection tools (*e.g., CCFinder* [20] and *SourcererCC* [44]) focus on analyzing code samples individually without considering the underlying connection among code samples. However, *Gitor* considers the connection among different code samples by extracting the *individual information* of a code sample as its representation, and the extracted *individual information* is used to build

a global graph to represent the whole code base which preserves the underlying connections of all code samples.

Why not compare with deep learning based methods. First of all, *Gitor* is not a deep learning-based method, and the experiment we conduct in Section 4.3 is only used to prove that *Gitor* can boost the performance of other detection methods, which does not suggest that *Gitor* is a deep learning based. Second, deep learning-based methods require training a detector on large labeled datasets, which is time-consuming and limits the practicability and scalability of deep learning-based clone detectors. In contrast, *Gitor* does not require time-consuming training on large labeled datasets, making it more practical for use in real-world applications.

Future work. The embedding process of *Gitor* is very efficient since we make use of *ProNE* [58], however, the code clone classification process is quite time-consuming due to its $O(n^2)$ complexity to get all clones detected. In future work, we consider techniques like filtering to improve our classification speed by filtering most of the unlikely code pairs according to the properties of the sample itself, such as lines of a sample, and the number of tokens in a sample. Besides, to achieve better detection performance, we will explore more types of *individual information* to represent code samples more properly and more accurately.

6 RELATED WORK

This section introduces related studies on code clone detection, which can be classified into five categories: text-based methods, token-based methods, tree-based methods, graph-based methods, and metrics-based methods.

The similarity between two code snippets is measured in the form of text or strings for the text-based methods [11, 18, 42]. [18] proposes a fingerprinting technique for detecting code clones. [11] develops a language-independent method for detecting similar codes using only line-based string matching. These two techniques, however, do not support Type-3 clone detection. To detect more types of clones, *Nicad* [42] introduces a two-stage approach that consists of i) identifying and normalizing potential clones using flexible pretty-printing and ii) computing similarity by simply text-line comparison using the longest common subsequence algorithm. Although *Nicad* can detect a number of Type-3 clones, it cannot detect Type-4 clones because it ignores the program semantics of given code samples.

For the token-based techniques [13, 20, 29, 44, 52], tokens are first collected from program code by lexical analysis. *CCFinder*

[20] extracts a token sequence from the input code and converts it into a regular form for finding Type-1 and Type-2 clones using numerous rule-based transformations, and *SourcererCC* [44] has been developed to support Type-3 clone detection, which is designed to capture the tokens' overlap similarity among multiple approaches for detecting Type-3 clones that are close to being detected. *SourcererCC* [44] is the most scalable code clone detector, capable of detecting 250 million lines of code. However, token-based detection methods, like text-based approaches, are unable to handle Type-4 clones.

To detect code clones, the tree-based tools [17, 54, 59] employ *Abstract Syntax Tree* (AST) as the code representation. *Deckard* [17]'s core idea is to compute characteristic vectors within ASTs and use *Locality Sensitive Hashing* (LSH) to cluster comparable vectors for clone detection. *CDLH* [54] first converts ASTs to binary trees, then uses Tree-LSTM [47] to encode these trees into vector representations. Finally, these vectors are utilized to compare distinct codes' similarity. *ASTNN* [59] separates each huge AST into a sequence of little statement trees, unlike *CDLH* [54]. To find semantic code clones, after encoding these statement trees into vectors, a bidirectional RNN model is utilized to construct the final vector representation of a code fragment. These tree-based methods can detect semantic clones, but their scalability is limited due to their long execution times.

For the graph-based methods [9, 24, 26, 51, 60], program semantics are first distilled into multiple graph representations, such as program dependency graph and control flow graph. [24] and [26] both extract program dependency graphs from code fragments and locate similar codes by excavating isomorphic subgraphs to represent code clones. *CCSharp* [51] employs two strategies to reduce the overall processing cost of [24] and [26]: graph structure modification and characteristic vector filtering. However, due to the complexity of graph isomorphism and the heavy-weight time consumption of graph matching, it still has low scalability on large-scale code clone detection.

Metrics can be obtained from tree or graph representations of source code or straight from source code for the metrics-based techniques [7, 32, 38, 43]. Both [7] and [32] use metrics extracted from the AST to describe the source code and to identify code clones. In addition, [38] detects clones using a variety of metrics collected from source code (*e.g.*, classes, coupling, and hierarchical organization). These approaches use code features to determine how similar two code fragments are in terms of semantics.

7 CONCLUSION

In this paper, we propose *Gitor* to achieve scalable code clone detection. Given a source code base, we first generate a global graph representing the whole code base, and then apply graph embedding to extract the vectors of all code samples in the code base. Finally, the code sample vectors can be simply used to compute the similarity of different code sample. We evaluate *Gitor* on a widely used dataset and compare *Gitor* with other widely used code clone detection methods. The results show that *Gitor* is superior to *SourcererCC* [44], *CCFinder* [20], *NiCad* [42], *Deckard* [17], *CCAligner* [52], and *Oreo* [43] on both effectiveness and scalability.

Moreover, *Gitor* requires only about one hour to analyze 100 million lines of code and is the most scalable among our comparative tools.

8 DATA AVAILABILITY

Our data are available on our website: https://github.com/Gitor-clone/Gitor.

REFERENCES

[1] 2020. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[2] 2020. javalang. https://github.com/c2nes/javalang.
[3] 2020. Software for complex networks (Networkx). http://networkx.github.io.
[4] 2022. IJaDataset. https://github.com/jeffsvajlenko/BigCloneEval.
[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[6] B. S. Baker. 1995. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95)*. IEEE Computer Society, USA, 86.
[7] 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 (ISMS'99)*.
[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* (2007).
[9] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.
[10] Quanyu Dai, Xiao Shen, Liang Zhang, Qiang Li, and Dan Wang. 2019. Adversarial training methods for network embedding. In *The World Wide Web Conference*. 329–339.
[11] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*.
[12] Daniel M German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2009. Code siblings: technical and legal implications of copying code between applications. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR'09)*.
[13] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (ECSMR'09)*.
[14] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.
[15] Chao Huang, Junbo Zhang, Yu Zheng, and Nitesh V Chawla. 2018. DeepCrime: Attentive hierarchical recurrent networks for crime prediction. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 1423–1432.
[16] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2012. Inter-project functional clone detection toward building libraries: an empirical study on 13,000 projects. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*.
[17] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*.
[18] J Howard Johnson. 1994. Substring matching for clone detection and change tracking.. In *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*.

[19] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. 485–495. https://doi.org/10.1109/ICSE.2009.5070547

[20] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* (2002).

[21] Cory J. Kapser and Michael W. Godfrey. 2006. Supporting the Analysis of Clones in Software Systems: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (mar 2006), 61–82.

[22] Iman Keivanloo, Juergen Rilling, and Philippe Charland. 2011. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*.

[23] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. 2005. An empirical study of code clone genealogies. In *ESEC/FSE-13*.

[24] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 2001 International Static Analysis Symposium (ISAS'01)*.

[25] Rainer Koschke. 2012. Large-scale inter-system clone detection using suffix trees. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (ECSME'12)*.

[26] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*.

[27] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[28] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 272–283.

[29] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: a deep learning-based clone detection approach. In *Proceedings of the 2017 International Conference on Software Maintenance and Evolution (ICSME'17)*.

[30] Xueyi Liu and Jie Tang. 2021. Network representation learning: A macro and micro view. *AI Open* 2 (2021), 43–64. https://doi.org/10.1016/j.aiopen.2021.02.001

[31] Angela Lozano and Michel Wermelinger. 2008. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*. 227–236. https://doi.org/10.1109/ICSM.2008.4658071

[32] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance (ICSM'96)*.

[33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *Computer Science* (2013).

[34] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–841.

[35] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. 2002. Random graph models of social networks. *Proceedings of the national academy of sciences* 99, suppl 1 (2002), 2566–2572.

[36] Enrico Palumbo, Giuseppe Rizzo, Raphaël Troncy, Elena Baralis, Michele Osella, and Enrico Ferro. 2018. Knowledge graph embeddings with node2vec for item recommendation. In *The Semantic Web: ESWC 2018 Satellite Events: ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers 15*. Springer, 117–120.

[37] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. 1999. Extending software quality assessment techniques to Java systems. In *Proceedings Seventh International Workshop on Program Comprehension*. 49–56. https://doi.org/10.1109/WPC.1999.777743

[38] 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 (IWPC'99)*.

[39] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[40] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[41] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* (2007).

[42] Chanchal K Roy and James R Cordy. 2008. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 International Conference on Program Comprehension (ICPC'08)*.

[43] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V 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 (FSE'18)*.

[44] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*.

[45] Jiankai Sun, Bortik Bandyopadhyay, Armin Bashizade, Jiongqian Liang, P Sadayappan, and Srinivasan Parthasarathy. 2019. Atp: Directed graph embedding with asymmetric transitivity preservation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 265–272.

[46] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME'14)*.

[47] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[48] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*. 1067–1077.

[49] 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.

[50] Andrew Walenstein and Arun Lakhotia. 2007. The software similarity problem in malware analysis. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[51] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: an efficient three-phase code clone detector using modified pdgs. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*.

[52] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*.

[53] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. HELoC: Hierarchical Contrastive Learning of Source Code Representation. *arXiv preprint arXiv:2203.14285* (2022).

[54] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 2017 International Joint Conferences on Artificial Intelligence (IJCAI'17)*.

[55] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*.

[56] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K Roy. 2020. Lvmapper: A large-variance clone detector using sequencing alignment approach. *IEEE access* 8 (2020), 27986–27997.

[57] Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, and Katsuro Inoue. 2018. Investigating vector-based detection of code clones using bigclonebench. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 699–700.

[58] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. ProNE: Fast and Scalable Network Representation Learning.. In *IJCAI*, Vol. 19. 4278–4284.

[59] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*.

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

[61] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. Gman: A graph multi-attention network for traffic prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1234–1241.