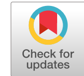
The Journal of Systems & Software 191 (2022) 111355

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Hierarchical semantic-aware neural code representation ☆



Yuan Jiang [a], Xiaohong Su [a,*], Christoph Treude [b], Tiantian Wang [a]

[a] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
[b] School of Computing and Information Systems, University of Melbourne, Melbourne, VIC, Australia

ARTICLE INFO

Article history:
Received 5 May 2021
Received in revised form 31 March 2022
Accepted 28 April 2022
Available online 6 May 2022

Keywords:
Code representation
Graph-LSTM
Hierarchical semantics
Program classification
Clone detection
Vulnerability detection
Deep learning

ABSTRACT

Code representation is a fundamental problem in many software engineering tasks. Despite the effort made by many researchers, it is still hard for existing methods to fully extract syntactic, structural and sequential features of source code, which form the hierarchical semantics of the program and are necessary to achieve a deeper code understanding. To alleviate this difficulty, we propose a new supervised approach based on the novel use of Tree-LSTM to incorporate the sequential and the global semantic features of programs explicitly into the representation model. Unlike previous techniques, our proposed model can not only learn low-level syntactic information within each statement but also the high-level semantic information between statements over the constructed semantic graph. Besides, considering that the sequential semantics is also critical for developers to understand the dependency path and data flow transmission, we propose a DFS-based method to generate the topological order of statements being processed, and then feed them as well as their in-neighboring information and syntactic embeddings into the proposed model to learn richer statement-level semantic features. Extensive experiments on multiple program comprehension tasks, e.g., code clone detection, demonstrate that our method achieves promising performance compared with other existing baselines.

© 2022 Elsevier Inc. All rights reserved.

1. Introduction

The application of deep learning in software engineering has achieved significant success, such as program classification (Mou et al., 2016; Zhang et al., 2019a), clone detection (White et al., 2016; Wei and Li, 2017; Fang et al., 2020), code completion (Liu et al., 2016b; Li et al., 2017) and code summarization (Hu et al., 2018a; Alon et al., 2018; Wan et al., 2018), which have been widely used in various types of software development and maintenance processes to assist human beings in reducing time and labor costs. For these tasks, it may be beneficial or even necessary to learn a rich semantic representation to understand the code snippet. Therefore, the deep code representation methods have attracted increasing attention due to the fact that they may achieve significantly better performance than the traditional code analysis methods.

Most work on code representation focuses on capturing local semantic information by processing the code directly or using a syntactic tree representation, treating it like sentences written in human-readable language (Ben-Nun et al., 2018). For example, several studies (Huo et al., 2016; Russell et al., 2018; Allamanis et al., 2016; Iyer et al., 2016) first directly lexed source code into a token sequence of variable length, and then encoded the entire sequence using two types of neural networks known as Long Short-Term Memory (LSTM Hochreiter and Schmidhuber, 1997) Networks and Convolutional Neural Networks (CNNs LeCun et al., 1998). There are also some studies (Wei and Li, 2017; Wan et al., 2018; Dam et al., 2018) that transformed code into tree-based abstract data (e.g., Abstract Syntax Trees (AST)) and applied recurrent neural network (RNN) variant Tree-LSTM (Tai et al., 2015) to construct code representation. Recent work (Dam et al., 2016; Hellendoorn and Devanbu, 2017) also demonstrates that LSTM and Tree-LSTM can achieve fairly satisfactory results for source code, similar to natural language. However, even so, these networks are still not competent enough to comprehend deep code semantics robustly due to the fact that source code usually contains richer and more explicit structural features (e.g., function calls, loops, control jumps between basic blocks, and interchangeable order of statements) than natural language text (Karmakar, 2019). Therefore, several advanced networks (Mou et al., 2016; Zhao and Huang, 2018; Allamanis et al., 2017; Fang et al., 2020; Zhang et al., 2019a; Feng et al., 2020; Guo et al., 2020) targeting the programming language domain have been proposed recently

☆ Editor: Alexander Serebrenik.

* Correspondence to: School of Computer Science and Technology, Harbin Institute of Technology, No.92, Xidazhi Street, Nangang District, Harbin City, Heilongjiang Province, China.

E-mail addresses: jiangyuan@hit.edu.cn (Y. Jiang), sxh@hit.edu.cn (X. Su), christoph.treude@unimelb.edu.au (C. Treude), wangtiantian@hit.edu.cn (T. Wang).

https://doi.org/10.1016/j.jss.2022.111355
0164-1212/© 2022 Elsevier Inc. All rights reserved.

to attempt to facilitate deep semantic understanding of the source code.

Inspired by the linguistic distributional hypothesis (Harris, 1954; Pantel, 2005) that semantically similar words tend to co-occur in the same contexts, a lot of researchers, like Ben-Nun (Ben-Nun et al., 2018) believed that this hypothesis holds in source code and reinterpreted it as "*Two statements that occur in the same contexts tend to have similar semantics*" for source code, where *context* is defined as a bag of semantically related statements which may be non-adjacent but are usually dependent on each other in terms of control or data flow. For example, if a variable is defined in the first line of a function but only used in the last line, there exists a "*context*" that is composed of the two lines of source code. Directly learning from consecutive statements in their appearance order in source code does not adequately satisfy this definition, as it only captures local sequential features of code at the statement level, while deep semantic features existing in "*context*" cannot be understood well enough.

To meet this need, one solution is to propose a context graph that incorporates structural information from both data- and control-flow of the code and then generate neighboring statement pairs with a fixed size of context, which is used for training statement embeddings using the skip-gram model (Ben-Nun et al., 2018). However, this method used unsupervised language modeling objectives to learn vector representation of each statement. Although the unsupervised training shows effectiveness and robustness on the final performance, it does not optimize the desired task directly; thus the improvement is limited (Liu et al., 2016a). Another solution is to employ graph neural networks (GNNs) to encode the semantic or syntactic information available with the Program Dependence Graphs (PDG Ferrante et al., 1987 for short) or augmented AST (Allamanis et al., 2017). However, as depicted in the previous work, each node in PDG represents a basic code block (i.e., a straight-line piece of code Wotawa and Krenn, 2007 that is at least a statement), which is too coarse-grained compared with other code representations (e.g., AST or token sequence) (Fang et al., 2020; Wagner et al., 1994). Therefore, the fine-grained semantic information located in a statement cannot be well learned from PDG combined with GNNs. In contrast, each node in augmented AST represents an element (e.g., a token) in the program, which is too fine-grained compared with other semantic-based representations (e.g., CFG and PDG). Therefore, the higher-level semantics based on dependencies between statements cannot be well captured from the augmented ASTs combined with GNNs.

In addition, current graph-based methods focus on capturing the structural information over the given graph (PDG or augmented AST), while neglecting the sequential semantics of statements. However, we believe the order of statements being processed is also critical for code understanding (Shido et al., 2019) since it specifies how statements interact with previous ones along the dependency path and data flow transmission (Huo and Li, 2017).

To tackle these challenges, in this paper, we propose a new approach to learn the program representation based on the novel use of Tree-LSTM. The key steps of our approach include a new scheme to embed the syntax and semantics of each statement contained in PDG nodes into a low dimensional vector space, and a DFS (Depth-First Search) based algorithm to map graph units into unit sequences to enable our model to capture additional sequential information. More specifically, given a code fragment, we first perform program dependency parsing to construct the PDG as the global context and then replace the nodes in PDG with the corresponding sub-ASTs to represent the local context. Next, we learn the syntactic embedding for each statement and employ a DFS-based approach to generate the sequences of statements being processed. Then, the syntactic embeddings and the neighboring information of statements as well as the statement sequences are used as input to our proposed model to learn the global statement-level semantic features. We make modifications to the Tree-LSTM model (named Graph-LSTM in this paper) to accommodate inputs from both the syntactic representation of each statement as well as its in-neighborhood information. Finally, we apply a max-pooling to condense the information of a variable length sequence of statement vectors into a single program vector.

Through the above process, our model is able to capture all the syntax, semantic, and long-range sequential dependencies from the AST and PDG as well as the order of statements being processed. Hence, the proposed method combines the advantages of previously proposed graph-based and sequence-based methods, and exhibits strong representative power for programs.

The main contributions of this paper include:

- **Semantic graph.** We introduce a novel representation named semantic graph to integrate syntactic and semantic information of source code.
- **Graph-LSTM.** We propose Graph-LSTM, a new code representation model based on deep learning, which embeds both low-level syntactic information and high-level semantic information over the constructed semantic graph of source code into a comprehensive code representation. The design of Graph-LSTM is based on a key insight that by matching the shape and structure of program graph traversals, the model can focus on learning the essence of program semantics at different levels of granularity, which is expected to alleviate the weaknesses (e.g., low efficiency and low effectiveness problems) introduced by the message-passing procedure in the standard GNN.
- **New method.** We design a novel method based on DFS- and topological sort algorithms to incorporate sequential information of source code into the code representation learning process, which can improve the feature representation capability of the proposed model.
- **Extensive experiments.** Evaluation results on multiple program comprehension tasks show that our strategies to account for capturing comprehensive semantic information improve the performance of the two tasks, and achieve the state-of-the-art performance on each.

Our paper is structured as follows. Section 2 presents a brief background on code representation and fundamental concepts about deep learning. Section 3 describes the overall framework of our method. Section 4 presents the details of the proposed network. Section 5 describes our experimental setting and the evaluation results. Section 6 presents a discussion and threats to validity. Finally, Section 7 introduces related work, and Section 8 concludes.

2. Background and motivations

2.1. Code representation

Various program analysis techniques have been proposed to explore the properties or behaviors of programs. According to prior research, the three most common representations, i.e., AST, control flow graphs (CFG for short) and PDG, are often chosen for the analysis of syntactic and semantic information of programs. To understand these techniques intuitively, the straightforward notions are outlined below.

(1) AST: represents the structure of program code. An AST of a program is an ordered tree where internal nodes represent operators or statements, and leaf nodes correspond to constants, identifiers or operands.

```c
1    int gcd(int a, int b)
2    {
3        int temp;
4        while (b != 0)
5        {
6            temp = a%b;
7            a = b;
8            b = temp;
9        }
10       return a;
11   }
```

Fig. 1. C code example to compute the greatest common divisor (GCD) of two positive integers a, b assuming $a > b$ for simplicity.

(2) CFG (Tip, 1994): explicitly describes the possible decisions of predicates and the possible sequences of execution of statements (Yamaguchi et al., 2014). A CFG of a program is a graph $G = (N, E)$, where N is the set of nodes that represent statements or control predicates of the program and E is a set of directed edges that represent the transfer of the control between two nodes. In particular, two distinguished nodes $entry \in N$ and $exit \in N$ represent the beginning (i.e., entry) and end (i.e., exist) of the control flow of the program, respectively.

(3) PDG (Tip, 1994): explicitly represents the dependencies among different statements and control predicates. A PDG of a program is a graph $G = (N, E)$, where N is the set of nodes that are the same as those of the CFG, and the edges in E denote the data and control dependencies between nodes.

As described in previous studies (Fang et al., 2020), AST is one of the commonly used syntax-based code representations which can represent the syntax of the program, while CFG and PDG are semantics-based representations which can store semantic information of the program, e.g., program execution and dependency information. To make it more clear, we give a simple code example of C language as shown in Fig. 1 to illustrate the differences of the above three types of representations. The corresponding representations of code for the example in Fig. 1 are shown in Fig. 2.

As can be seen from Fig. 2, AST gives us very useful information about the structure of the source code, which can detect some programming patterns for the same functionality from diverse forms of codes. For example, the statements "int a, b;" and "int a; int b;" are not quite same in the token sequences but have the same AST structure to implement the same functionality. However, there are also substantial code fragments that implement the same functionality, but their AST structures are noticeably different. For example, "for" and "while", with different syntax structures, are semantically similar because they are both loop statements. It is obvious from the aforementioned analysis that AST can well capture the local and structure information of the source code but cannot capture the functionality semantics of code fragments. To exploit the syntactic knowledge in AST, neural network-based methods, e.g., Tree-LSTM (Tai et al., 2015; Wei and Li, 2017), Tree-based CNN (Mou et al., 2016) and Recursive Neural Network (RvNN) (White et al., 2016) have been proposed recently, and used for modeling source code. These methods have proven to be quite powerful in many software engineering tasks, e.g., program classification, clone detection and code completion. But, there are some disadvantages that are not solved as of now. For example, keeping the original tree structure of the source code unchanged, AST's elements are much longer than those of raw source code, which makes it difficult for the neural network-based models to learn long-distance correlations in a much bigger tree due to the Vanishing Gradient (VG) problem and a bounded network memory given a fixed number of parameters. Moreover, some important semantic features cannot be well captured by these methods when modeling source code at the AST level as described above.

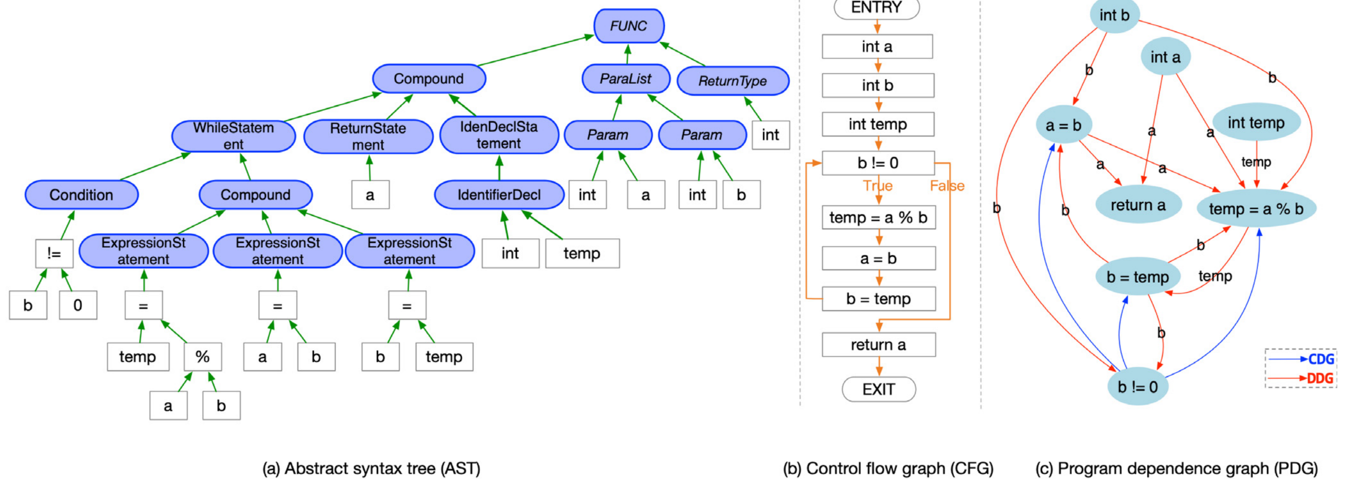
Due to the limitations of syntax-based representation in capturing code semantics, more and more researchers have drawn increasing attention in modeling source code at the CFG or PDG level. Since both CFG and PDG can be seen as a typical graph, it is natural to consider graph embedding techniques for code representation. For example, Fang et al. (2020) presented a fusion approach for simultaneously learning the syntactic and semantic features based on AST and CFG using word2vec and graph2vec respectively. Currently, for most of the existing graph embedding techniques (e.g., Graph2vec Narayanan et al., 2017, HOPE Ou et al., 2016, SDNE Wang et al., 2016 and Node2vec Grover and Leskovec, 2016), they aim to transform each node of the graph into a vector with fixed length. However, each node in CFG and PDG is a basic code block (i.e., a straight-line piece of code with at least a statement), which may be so coarse (Wagner et al., 1994) that some useful semantic information inside the code block could be lost (Fang et al., 2020). For example, following standard practice, the initialized feature of the statement "b = temp" is generated by taking the mean across all its tokens' embedding, which is identical to another statement "temp = b" consisting of the same tokens with the former (i.e., temp, =, b). That means if we replace the node "b = temp" of the PDG (see the right of Fig. 2) with "temp = b", we can obtain the same graph embedding since the replacement has no effect on both the initialized node features and the graph structure. This is not in agreement with actual practice, because the replacement changes the semantic meaning of the program.

Because of these limitations and shortcomings of the current syntax-based and semantics-based representations, some recent researches made attempts to mitigate the above mentioned challenges by carefully combining different types of representations. For example, Allamanis et al. (2017) proposed to augment ASTs by adding extra edges among tree nodes to represent a variety of code dependencies. Taking the code snippet in Fig. 1 as an example, the augmented AST generated by their method is shown in Fig. 3. As can be seen, the structure of the augmented AST might reflect data flow and control flow between variables. However, this approach also has its downsides, i.e., (1) it would still fail to identify the similar semantic units that implement the same functionality but have different forms (e.g., *for* and *while* statements mentioned above); (2) it may contain some redundant information and noisy data, and thus adversely hurt the generalization of the representation model (Jayasundara et al., 2019). In addition, considering that the constructed graph is much more complicated than the AST of the same program, it may also suffer from the VG problem and then cause ineffective model training of GNNs (Bieber et al., 2020).

Inspired by limitations in existing program representation techniques, our goal is to produce a new way of representing code to capture more comprehensive code semantics from both the syntactic information inside each statement and program dependencies among statements.

2.2. Neural networks for code representation

Deep Learning is based on neural networks which connect many small neurons into a complex network structure, enabling them to extract richer features of training data. So they can fully utilize the extracted features to accomplish specific tasks, which have proven to generally perform better than purely traditional machine learning algorithms (Kumar et al., 2016). The application of deep learning in software engineering has also achieved significant success, e.g., using CNN variant tree-based CNN (Mou et al.,



(a) Abstract syntax tree (AST)   (b) Control flow graph (CFG)   (c) Program dependence graph (PDG)

Fig. 2. Representations of code for the example in Fig. 1. Control flow is indicated by orange lines in the control flow graph. Control and data dependencies are indicated by red and blue lines in the program dependence graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

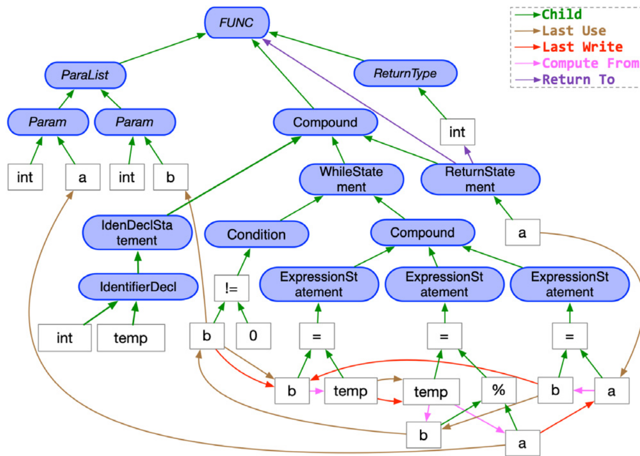


Fig. 3. The augmented AST with data and control-flow edges. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2016) to construct the program representation according to the abstract syntax tree.

However, with the increasing number of layers, the RNN and CNN may suffer from the VG problem, which can lead to ineffective and inefficient model training (Bengio et al., 2009). Therefore, variants such as LSTM (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Cho et al., 2014) networks are often deployed in practice to learn long-term dependencies (Zhang et al., 2019b). These networks have shown massive success in many NLP tasks, but there has been limited work in applying LSTM and GRU in representing source code. This is because source code involves too much structural information (e.g., CFG, and PDG) which is hardly captured by these networks due to the fact that they are especially designed for sequential data. To address this limitation, various graph-based neural networks (e.g., GGNN Li et al., 2015; Allamanis et al., 2017) have been recently employed for the code representation task and have achieved state-of-the-art results. However, these models also have their drawbacks: (1) they focus only on capturing information related to the graph structure of programs, and ignore the sequential semantics between statements; (2) they cannot handle the local syntactic information inside each node (e.g., a basic code block in PDG) well, as we detailed in Section 1. Thus, there is still room for improvement.

In this work, to overcome the above-mentioned limitations, we develop a novel neural network based on Tree-LSTM, called Graph-LSTM that is designed for representing the syntactic information within each statement and the global structural information as well as the sequential naturalness among statements.

2.3. Downstream tasks of code representation model

Our work focus on two common prediction tasks in software engineering: program classification and code clone detection, following previous studies.

Program classification aims to classify newly added source files into different categories according to their functionalities in the software development process (Mou et al., 2016), which is an important research area as orderly management of code resources makes code easier to reuse and then helps with speeding up coding. More formally, given the representation $r \in \mathbb{R}^k$ of code fragment X_i, and a set of categories of length M, the program classification task is defined as finding the M-dimension predicted result $\hat{y}$, such that:

$$\hat{y} = sigmoid(Wr + b) \tag{1}$$

where $W \in \mathbb{R}^{M*k}$ is the weight matrix and $b \in \mathbb{R}^M$ is the bias item. In this paper, we explore the use of deep learning techniques to address the program classification task. During training, we employ the widely used cross entropy as the loss function for optimization.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[p(y_j|X_i)log(\hat{p}(y_j|X_i))\right] + \gamma\,\|\theta\|^2 \tag{2}$$

where N is the total number of training instances, M is the number of program categories, $\hat{p}(y_j|X_i)$ is the output of the neural network model and $p(y_j|X_i)$ is the desired output whose value is 1 if y_j is the ground truth, and otherwise is 0. The last term of the formula is the sum of squares of all parameters in the network, which can effectively alleviate the overfitting and slow updating problems.

Clone detection aims to look for exact or similar code pairs or groups, called clones, in given code corpora by measuring the similarity between two code fragments, which can help to reduce the cost of software maintenance and prevent faults (Sheneamer

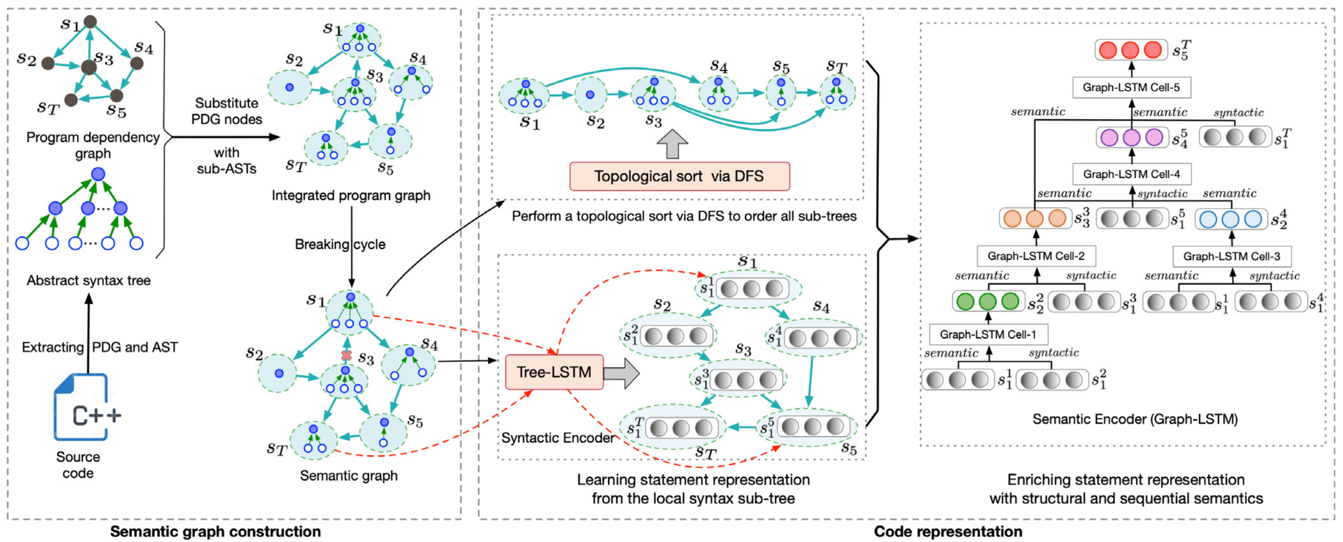


Fig. 4. An overview of the proposed deep learning framework for code representation. The left subplot describes the process of the semantic graph construction, where the input code fragment is first represented as a PDG, which can be further combined with the corresponding AST to obtain an integrated program graph. Next, the semantic graph can be constructed by removing cycles in the integrated program graph. The right subplot illustrates the detailed architecture of the proposed code representation framework, where the syntactic encoder first learns statement syntactic representation from the local syntax tree, and then the semantic encoder learns the global statement-level semantic features by taking the syntactic embedding of statements as well as the statement sequences as inputs.

and Kalita, 2016; Wang et al., 2020a; Yu et al., 2019). Given a pair of code clones, they may be semantically similar (in terms of their target functionality) but differ syntactically to various degrees. Therefore, following a common taxonomy (Sajnani et al., 2016; White et al., 2016; Zhang et al., 2019a; Fang et al., 2020; Wang et al., 2020a), code clones can be classified into four categories based upon the syntactical dissimilarity.

- Type 1: Identical code fragments except for differences in whitespace, blanks, comments, and layout.
- Type 2: In addition to clone differences in Type 1, syntactically identical code fragments except for differences in variable names, type names, literal values and function names.
- Type 3: Syntactically similar code fragments except for several statements added, modified, or removed with respect to each other.
- Type 4: Syntactically dissimilar code fragments that implement the same functionality.

Suppose there are two code fragments and let v_1 and v_2 be the corresponding vector representations. The semantic relevance between v_1 and v_2 is measured according to their distance, i.e., $r = v_1 - v_2$ (Tai et al., 2015; Zhang et al., 2019a). Then their similarity score can be calculated by $\hat{y} = sigmoid(Wr+b)$, where $W \in \mathbb{R}^{2*k}$ is the weight matrix and $b \in \mathbb{R}^2$ is the bias item. Similar to program classification task, we use the cross-entropy as loss function for optimization.

3. Our approach

In this paper, we propose a novel framework which first constructs a semantic graph for the input of source code and then encodes the semantic graph by introducing a novel Graph-LSTM. Fig. 4 shows the overview of the proposed approach, containing the stages of (1) semantic graph construction, and (2) code representation.

3.1. Semantic graph construction

We introduce a new code representation, which we refer to as semantic graph, to integrate syntax and semantics of source code. For clarity, the semantic graph is defined as follows:

Definition 1. A semantic graph is defined as $G = (S, E)$, which is a directed, acyclic graph with a set of sub-trees, $S = \{s_1, s_2, \ldots, s_n\}$ and a set of edges, $E = \{e_{i,j}\}$. Each sub-tree $s_i \in S$ represents the syntax tree rooted at each PDG node (i.e., a statement). Each edge $e_{i,j}$ is an ordered pair, connecting s_i to s_j, which represents the control or data dependencies between two sub-trees.

In this phase, we parse the source code to generate a semantic graph in the following steps:

Step 1. **Extracting dependencies among statements**. To capture the data and control dependencies among statements, we parse each method in source code to create its PDG. To this end, there are some standard algorithms (e.g., Ferrante et al. (1987)). As a running example, the second column of Fig. 5 shows the PDG corresponding to the code snippet in the first column, where each number represents the node id whose value is generated according to the order of statements appearing in code. Since the PDG represents source code at the statement level, it is a higher level of abstraction than token-based representations or ASTs, and thus only serves as a skeleton in representing source code.

Step 2. **Substituting PDG nodes with sub-ASTs**. To capture the syntactic features within statements contained in the PDG nodes, we firstly parse the source code into AST. Then we establish correspondence between AST nodes and PDG nodes based on their locations in source code. Finally, we replace the PDG nodes with sub-trees accordingly to generate an integrated program graph which can not only represent the syntax information within statements but also the data- and control-dependence information among statements. Taking the code snippet in Fig. 1 as an example, the generated integrated program graph is shown in the third column of Fig. 5, where the syntactic sub-trees are outlined in green dashed line while the semantic relationships are annotated in blue line and red line.

Step 3. **Breaking cycles in the constructed graph.** This step aims to break cycles from a directed integrated program graph, while preserving the underlying dependencies between statements as much as possible. In this paper, we utilize the most commonly used DFS-based approach to simplify a directed integrated program graph modeling syntactic information and dependency relationships into a directed acyclic semantic graph, which only includes acyclic relationships and has a hierarchical structure.

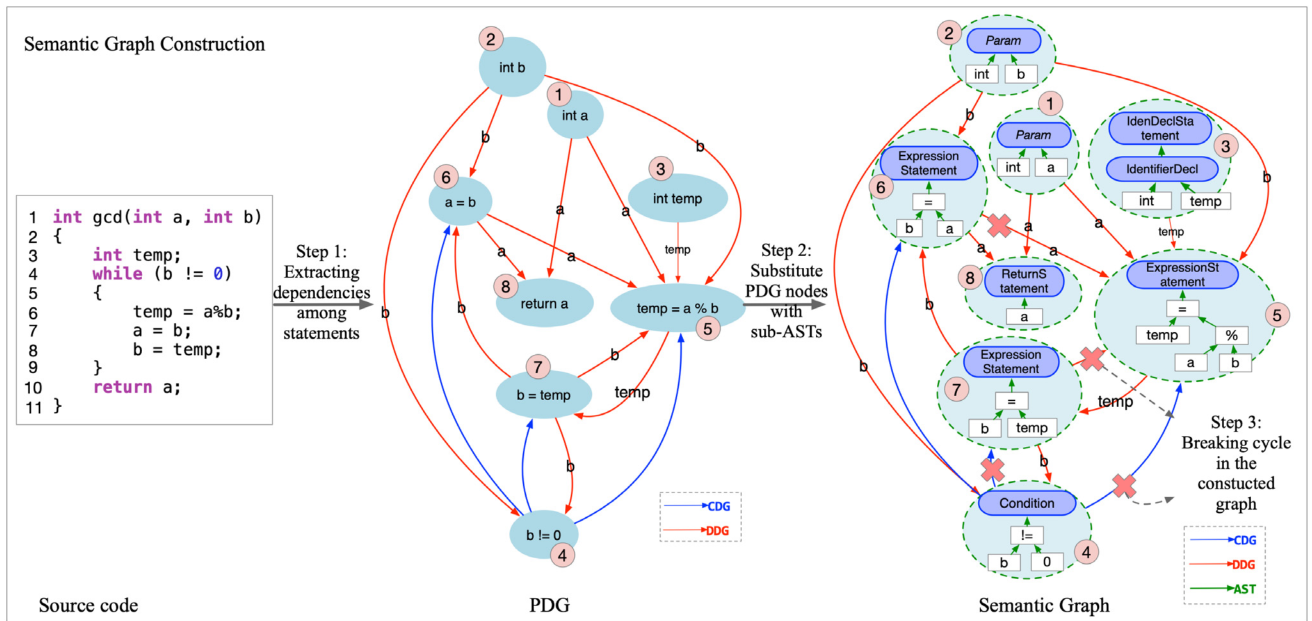


Fig. 5. A running example to illustrate the process of semantic graph construction, which consists of three steps: (1) Extracting dependencies among statements; (2) Substituting PDG nodes with sub-ASTs; (3) Breaking cycles in the constructed graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Afterwards, we can perform a topological sort of statements (i.e., sub-trees) in the constructed graph and then recursively apply our proposed code representation model along the sequences of sub-trees to capture the deep sequential and structural semantics in the following steps. Note that the breaking cycle process is inevitable since our approach requires the topological order of statement processing as input, and topological-sort can only be performed on graphs with no cycles.

3.2. Code representation model

Both syntax-based and semantics-based code representation approaches have their own limitations as we discussed in Section 2.1. Therefore, different from others directly learning the code representation at the token level (e.g., token-based or AST-based methods) or statement level (e.g., CFG-based or PDG-based methods), we propose a novel code representation framework which first learns the low-level syntactic information from the local context, and then learns the high-level semantic information from the global context.

To obtain more accurate statement representations, we conduct the following steps.

Step 1. **Learning statement representation from the local syntax sub-tree.** To better capture the semantics in the local context, we employ Tree-LSTM as the Syntactic Encoder, which takes the AST of each statement as input and outputs a continuous, real-valued vector for each statement, to generate a highly non-linear semantic feature for each minimal unit of semantic information (i.e., statement). Since syntactic encoder (SynEncoder for short) learns from local contexts, it cannot capture the global characteristics associated with program execution or dependency information between statements, and thus only serve as the previous step in producing feature vectors that represent the deep semantics of source code.

Step 2. **Enriching statement representation with structural and sequential semantics**. Encoding global semantic features for statements written in high-level programming languages (e.g., C/C++) can be nontrivial, precisely because of a lack of modeling the sequential and structural information simultaneously. To enrich the local semantic features of statements, which have been represented by continuous vectors in the first step of this phase, we propose a graph-based LSTM (Graph-LSTM) approach to learn the global features of a statement from its in-neighbor units over a given graph structure in topological order. Fig. 6 shows the general overview of our framework to model the global context.

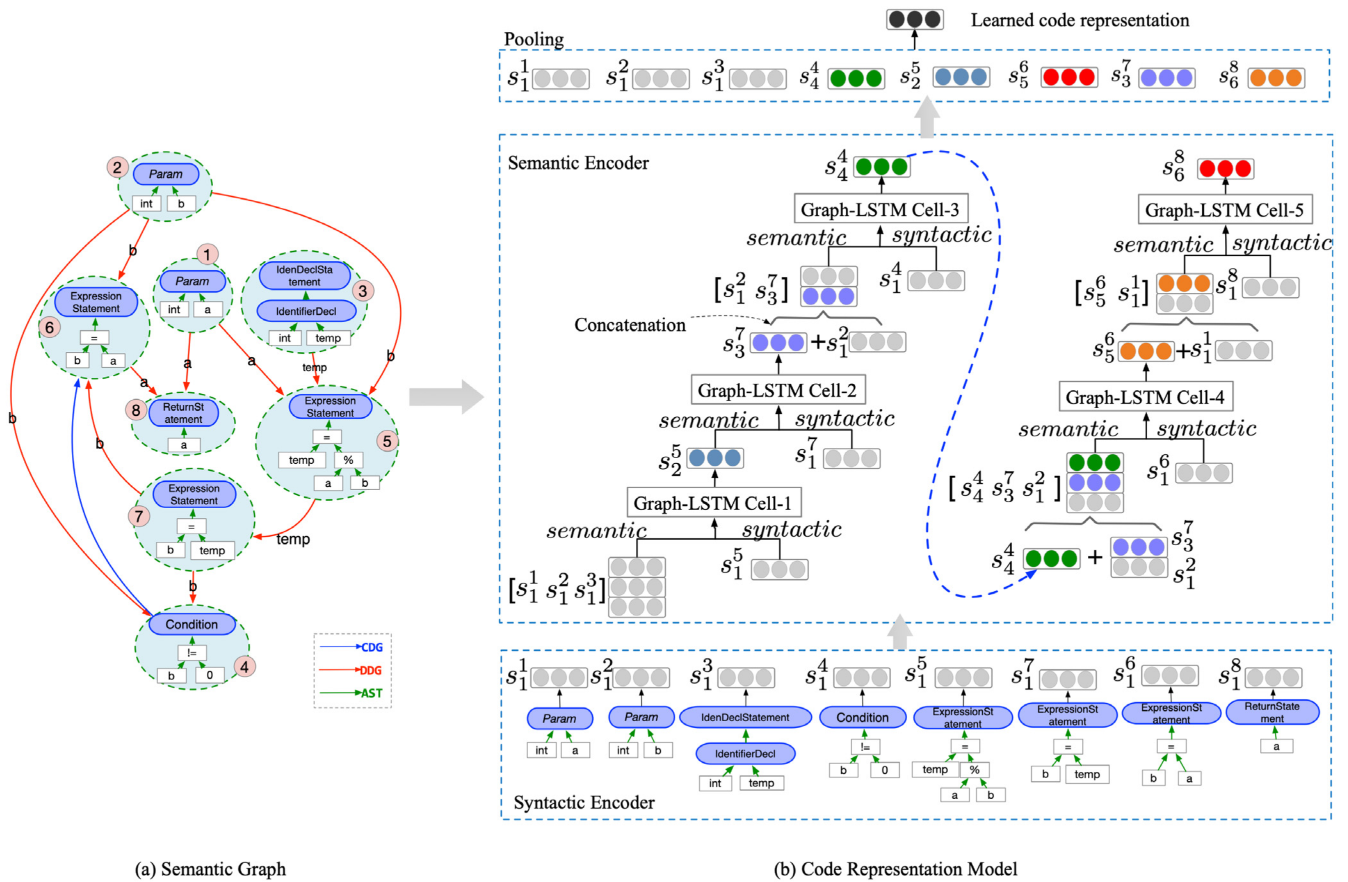
Our method for capturing the structural semantics is motivated by the following observation.

Observation 1: When humans comprehend a program, they particularly focus on the program structure besides the syntax information. This is because the program structure specifies how different statements interact with each other to accomplish certain functionality, and thus conveys insights into the semantics of source code (Huo et al., 2016).

Therefore, for a statement s, it is important to capture the structural information from its direct neighbors. To this end, we make modifications to the Tree-LSTM model to accommodate inputs from both the syntactic representation of each statement and its in-neighborhood information. In this paper, we refer to the modified Tree-LSTM as the graph-based LSTM (Graph-LSTM) since it is designed for graph data. Different time steps in Graph-LSTM generate the global representation of different statements. For example, the second time step is to enrich the 5th statement features (i.e., $\boldsymbol{s}_2^5$ where the subscript and superscript indicate the time step and the id of sub-trees in the semantic graph, respectively) according to its in-neighbors (i.e., $\boldsymbol{s}_1^1$, $\boldsymbol{s}_1^2$ and $\boldsymbol{s}_1^3$) and its syntax representation (i.e., $\boldsymbol{s}_1^5$). Obviously, the aforementioned learning process for statement $\boldsymbol{s}_i$ considers both the global context semantic information (i.e., dependencies between statements) and its local context syntax information.

In addition to the structural semantics described above, we also consider the sequential semantics in our proposed method. This is inspired by the following observation.

Observation 2: When humans comprehend a program, they generally do it in an incremental and sequential manner (Hendrix et al., 2002; Sneed and Dombovari, 1999). That means the order of statements being processed in source code is critical to



(a) Semantic Graph (b) Code Representation Model

Fig. 6. The overall architecture of our proposed Graph-LSTM. The input code fragment is first represented as a semantic graph as we described in Section 4. Then each sub-tree in the semantic graph is encoded into a fixed-length vector via the Syntax Encoder. Next, the syntactic embedding of each statement and its in-neighborhood information as well as the statement sequences are used as input for the proposed Graph-LSTM model (i.e., Semantic Encoder) to learn the global statement-level semantic features. Finally, a max-pooling operation is used to condense the information of a variable length sequence of global statement vectors into a single program vector. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

deep semantic understanding of the source code since it specifies how statements interact with previous ones along the dependency path and data flow transmission, and hence provides additional semantics to program functionality aside from syntax and structural information (Huo and Li, 2017).

Therefore, it is important to determine an appropriate ordering of the sub-trees. To this end, in this paper, we propose to leverage the DFS-based method to generate a reasonable topological ordering of the subtrees and then recursively apply Graph-LSTM along the sequences of sub-trees to capture both the sequential and structural semantic features.

Overall, in the first step (i.e., step 1), we treat each sub-tree as an independent unit, we can use Tree-LSTM to compute the "local" feature of each statement w.r.t. the sub-tree itself. Then, in the second step (i.e., step 2), these local features can be used to construct the final "global" features of statements by further considering their semantic relations and sequential information.

The main advantages of our proposed method are: (1) compared to traditional ones (e.g., CNN and LSTM), it naturally deals with extracting features from non-consecutive but semantically related statements by leveraging program dependency information; (2) compared to those only considering graph-structure, it can better capture both sequential and structural information through the proposed Graph-LSTM; (3) compared to previous methods based on ASTs or PDGs, it first learns the low-level syntax information over sub-trees within each statement, and then learns the high-level semantic information based on the dependencies between statements. All these advantages enable our approach to learn more accurate code representations.

The specific network architecture and learning procedure are described in Section 4.

4. Detailed model architecture

In this section, we will provide details for the key steps of our proposed code representation method.

4.1. Breaking cycles when constructing semantic graphs

Currently, owing to its simplicity and efficiency, the Depth-First search (DFS) method has been widely used to remove cycles in the directed graph. Specifically, supposing that the DFS method starts at sub-Tree s, we traverse the integrated program graph (detailed in Section 3.1) in a depth-first manner from s and then remove the back edges (dependencies) between sub-trees. Note that removing cycles from the integrated program graph is equivalent to that from the PDG since the cycle (or loop) structure only occurs in the PDGs and not in sub-trees of statements. Therefore, in what follows, we denote the sub-tree of each statement by using the corresponding PDG node's id for simplicity in representation. The detailed procedure is shown in Algorithm 1, where G stands for the integrated program graph of the source code and E represents the removed edges that are obtained during the DFS traversal. First, for each sub-tree s, we assign a property symbol "color", whose value is initialized to "white", to indicate its state during the search (Line 1–2). Second, we initialize an empty set E to store the removed edges (Line 3) and reset the global time

counter *time* in order to record the discovery timestamp d and the finished timestamp f for each sub-tree in the next steps (Line 4). Thirdly, we loop over the sub-trees in S and check if the color of the sub-tree is "white" (Line 5–6). If so, we visit it using DFS-VISIT (Algorithm 2). Whenever *DFS-VISIT(G, s, E)* is called in line 7, sub-tree s becomes the root of a new tree in the depth-first forest. When *DFS-VISIT* returns, the removed edges have been added into E, and the current sub-tree s has been assigned a discovery time $s.d$ and a finishing time $s.f$.

Algorithm 1: Breaking cycles in the integrated program graph using the deep first search

```
   Input: Semantic graph G
   Output: Edges E need to be removed
 1 for each sub-tree s ∈ G.S do
 2     s.color = WHITE ;
 3 E ← ∅ {Initialize a set to store the removed edges } ;
 4 time = 0 ;
 5 for each sub-tree s ∈ G.S do
 6     if s.color = WHITE then
 7         DFS-VISIT(G, s, E)
 8 return E
```

Algorithm 2 shows pseudocode for the process of *DFS-VISIT(G, s, E)*. At every call to *DFS-VISIT*, the "color" attribute of sub-tree s is changed from "white" to "gray" (Line 1). Next, the global variable time is increased by 1, whose value is treated as the discovery time $s.d$ of s (Line 2–3). Finally, we examine each sub-tree v adjacent to s and then recursively visit v if it is marked as "white" (Line 4–6), otherwise we add the edge (s, v) to the set of edges E if v is marked as "GRAY" (Line 7–8). It can be seen that all edges *(s, v)* connected with s is explored by the depth-first search since each neighboring sub-tree $v \in G : Adj[s]$ is considered in the loop of line 4. Finally, after every edge starting from s has been explored, we change the "color" of s from "gray" to "black" and then record the finishing time (i.e., time $+1$) in *s.f* (Line 11–13).

Algorithm 2: DFS-VISIT

```
   Input: Semantic graph G, sub-tree s, edges to be removed
          E
   Output: Edges E need to be removed
 1 s.color = GRAY;
 2 time = time + 1;
 3 s.d = time;
 4 for each sub-tree v ∈ G : Adj[s] do
 5     if v.color = WHITE then
 6         DFS-VISIT(G, s, E)
 7     else if v.color = GRAY then
 8         E ∪ edge(s, v);
 9     else
10         continue;
11 s.color = BLACK;
12 time = time + 1;
13 s.f = time;
```

Taking the integrated program graph in the second columns of Fig. 5 as an example, assuming that a DFS starts from a sub-tree s_1, the method will remove cycle edges (s_6, s_5), (s_7, s_5), (s_4, s_7) and (s_4, s_5). The detailed traverse and removing processes are depicted in Fig. 7.

4.2. Topological sort

After applying the DFS-based method to remove cycle edges in the integrated program graph, we can obtain a semantic graph with no cycles. Next, we perform a topological sort to order all sub-trees such that s_i appears before s_j when the dependency (s_i, s_j) holds. The advantage of this is that it provides intuitive insights into which statements should be processed first when the s_j is processed. More importantly, indicating precedence among statements can provide an opportunity for the code representation model to encode the sequential and structural information simultaneously, which will be depicted in Section 4.4. The topological-based ordering process can be accomplished by sorting the vertices (or sub-trees) chronologically by their finished timestamps. In particular, since the semantic graph is a directed acyclic graph, it must have a topological-sort in which all sub-trees are ordered along a horizontal line and their dependency relations go from left to right. Fig. 8 presents the topologically sorted semantic graph of code snippet in the first column of Fig. 5, where all dependency relations are from the left to the right. The ordering result is quite sensible because it is consistent with the human understanding of programs.

4.3. Syntactic encoder

Each statement of source code in the proposed semantic graph is represented in a tree structure, which provides opportunities to capture syntax information of programs. In this paper, we adopt the child-sum Tree-LSTM as the syntactic Encoder (i.e., SynEncoder) for modeling the tree structures of statements since it can be built on arbitrary tree structures. Formally, given a sub-tree, the encoder learns distributed representations of the AST nodes. At each node j in the sub-tree, let x_j denote the corresponding token and let the hidden state and memory cell of its lth child be h_{jl} and c_{jl} respectively. The hidden state of x_j is generated based on the states of its children in the sub-tree, which is calculated in the Tree-LSTM cell as follows:

$$
\begin{aligned}
\hat{h_j} &= \sum_{l \in C(j)} h_{jl} \\
f_{jl} &= \sigma(W^f \cdot \boldsymbol{x_j} + U^f \cdot h_{jl} + b_f) \\
i_j &= \sigma(W^i \cdot \boldsymbol{x_j} + U^i \cdot \hat{h_j} + b_i) \\
u_j &= tanh(W^u \cdot \boldsymbol{x_j} + U^u \cdot \hat{h_j} + b_u) \\
c_j &= \sum_{l \in C(j)} f_{jl} \odot c_{jl} + i_j \odot u_j \\
o_j &= \sigma(W^o \cdot \boldsymbol{x_j} + U^o \cdot \hat{h_j} + b_o) \\
h_j &= o_j \odot tanh(c_j)
\end{aligned}
\tag{3}
$$

where i_j, $f_{j(*)}$, o_j represent the input, forget and output gates at the jth step respectively, while c_j and h_j represent the memory cell and output vector respectively. u_j denotes a state for updating the memory cell and $\boldsymbol{x_j}$ represents the embedded vector of node j. σ denotes the sigmoid function and $\odot$ denotes element-wise multiplication. $W_{\cdot} \in \mathbb{R}^{e \times d}$ are weight matrices and $b_{\cdot} \in \mathbb{R}^d$ are bias terms. Through the iterative procedure of computing hidden states of all nodes, the syntax information within the sub-tree structure can be obtained. Note that, in this paper, bold and normal text are used to distinguish between a vector and a symbol, respectively, e.g., $\boldsymbol{x_j}$ represents an embedded vector and x_j represents a token symbol.

4.4. Semantic encoder

As the statements contained in PDG nodes are represented in the form of AST sub-trees, the SynEncoder can capture the syntactic information of code via the Tree-LSTM network. However, it is not sufficient to extract the semantics of global context since the SynEncoder does not consider the relations (i.e., data and control

Fig. 7. The process of removing cycle edges in the integrated program graph by applying the DFS-based method. For simplicity in presentation, we denote the sub-tree of each statement by using the corresponding PDG node's id. The background color of each node (i.e., sub-tree) represents its state during the search. Timestamps surrounding vertices are marked as green, which indicates discovery and finishing times. As vertices are explored by the algorithm, the removing edges are plotted as red "X". As can be seen on the last subplot, the removed cycle edges are (s_6, s_5), (s_7, s_5), (s_4, s_7) and (s_4, s_5). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

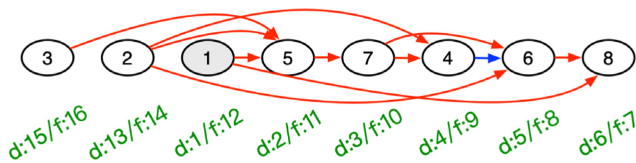


Fig. 8. The topologically sorted semantic graph, in which all vertices (sub-trees) are sorted along a horizontal line in the descending order of their finishing timestamps, and all edges are coming from the left vertices and pointing to the right.

dependencies) among statements. Therefore, we can define the goal of this phase as follows: Given a semantic graph $G = (S, E)$, where each statement in the form of sub-tree s_i has been encoded into a fixed-length vectorial representation via the SynEncoder, the semantic encoder module enriches statement representation with neighbor information derived from the graph structures.

Specifically, given a sub-tree $s_i \in S$ with a hidden representation h_i learned by SynEncoder and its in-neighbors $N_G^-(i)$ as well as their hidden representations $\boldsymbol{h}_{N_G^-(i)} = \{h_k | k \in N_G^-(i)\}$, the Graph-LSTM updates the node's hidden representation using the following equation.

$$h_i^e = f(x_i, h_i, \boldsymbol{h}_{N_G^-(i)}) \quad (4)$$

where x_i is the type of the root node of s_i (e.g., x_6 in the semantic graph shown in the first column of Fig. 6 is "*ExpressionStatement*")

and $f(\cdot)$ is the composite function made up of a series of cascade-operations, most of which have the same formulation with the Tree-LSTM model as we detailed in Eq. (3). The main difference between the Graph-LSTM and Tree-LSTM is that, in the Tree-LSTM, the internal gates (i.e., the input, output and intermediate cell states) are calculated based on the hidden states of the children of the node in the given tree, while in the Graph-LSTM, these gates are calculated using the hidden states of in-neighbors $N_G^-(i)$ and the hidden state of the current sub-tree. In this paper, the formula of $\hat{h}$ in Graph-LSTM is defined as follows:

$$\hat{h}_i = h_i + \sum_{k \in N_G^-(i)} h_{ik} \tag{5}$$

Next, using this modified hidden state $\hat{h}_i$, the internal gates, intermediate cell state and the updated hidden state h_i^e are calculated using the same formula described in Eq. (3). In addition, we can use the same network parameters as in the SynEncoder and thus reduce the total parameters of the entire model.

4.5. Program encoder

The obtained hidden vector h_i^e that is output in the previous phase can be regarded as the semantic feature of each statement. Then, an aggregation operation is required to condense the information of a variable-length sequence of statement vectors into a single vector. To this end, we utilize the max pooling operation on top of the hidden states to obtain the final program vector representation.

As we detailed in Section 2.3, the program vector can be used for the downstream target-specific tasks. For example, according to the program vector, we can use the logistic regression with sigmoid function, i.e., $y = sigmoid(W_1 r + b_1)$ as shown in the formula (1) to predict the probability of the program X belonging to the mth class.

5. Evaluation setting

5.1. Research questions

In order to show the performance of our method[1] clearly, in our experiments, we are interested in the following questions.

RQ1: How does our approach perform in program classification compared with the state-of-the-art approaches?

RQ2: How does our approach perform in code clone detection compared with the state-of-the-art approaches?

RQ3: What is the effectiveness of the proposed semantic graph and Graph-LSTM in our method?

RQ4: How efficient is our approach during the training and testing phases?

5.2. Dataset preparation

We evaluate the proposed method on OJ and BigCloneBench datasets, and demonstrate its efficacy at program classification and code clone detection tasks. The OJ benchmark (named OJ-Data-1) is collected from a pedagogic program online judge (*OJ*) system and made public by Mou et al. (2016), which mainly includes C/C++ programs for 104 programming tasks, and each task has 500 programs submitted by different students. Considering the relatively small size of the OJ benchmark, we collect more labeled data samples of other programming tasks from the OJ system.[2] Our collected dataset (named OJ-Data-2) has the same number of programming tasks as the OJ Benchmark, in which each programming task consists of 500 programs. Thus, we can merge the two datasets (i.e., OJ-Data-1 and OJ-Data-2) to create a large-scale one for our experimental evaluation. The merged dataset (named OJ-All) studied in this paper is accessible through the following link.[3]

For the program classification task, following previous work (Zhang et al., 2019a; Mou et al., 2016), we perform an extensive evaluation of our proposed method on OJ datasets. The target label of a program in OJ dataset is the *id* of the programming problem to which the program belongs. Experimental results on three datasets (i.e., OJ-Data-1, OJ-Data-2 and the merged dataset OJ-All) are provided. For each dataset, we randomly shuffle it, use the first 60% of the programs to train our model, and split the rest equally for validation (20%) and testing (20%).

For the code clone detection task, both OJ benchmark and BigCloneBench dataset are employed for an overall assessment of the proposed method. As the previous work in Zhang et al. (2019a), for the OJ benchmark, we choose the first 15 class labels. Each of them corresponds to a programming problem and consists of 500 instances (i.e., source programs). Since two programs that solve the same problem (i.e., implement the same functionality) form a clone pair, a huge number of pairs (i.e., more than 28 million clone pairs) are generated by randomly pairing two programs drawn from the same target class, which will result in a large consumption of resources and times for comparison. Thus, a common practice is to sample fewer program pairs (e.g., 50 thousand samples in Zhang et al. (2019a) and in this paper) instead. Note that the dataset of clone pairs sampled from the OJ benchmark is also called the OJClone dataset in the literature (Fang et al., 2020). Similarly, we randomly divide the OJClone datasets in a ratio of approximately 60%/20%/20% (training/validating/evaluating) to perform extensive experiments.

The BigCloneBench dataset is published by Svajlenko et al. (2014) and widely used for evaluating code clone detection tools, which is mined from the big data inter-project repository IJa-Dataset 2.0 (Ambient Software Evoluton Group, 2013) and consists of 6 million true positive clones of different clone types: Type-1, Type-2, Type-3 and Type-4, as well as 260 thousand false clone pairs. The similarity of clone pairs is defined as the minimum ratio of the lines one code fragment shares with another after normalization (Svajlenko et al., 2014). Concretely, the similarity of two fragments of Type-1 and Type-2 is 1 since their lexical sequences are the same after the appropriate code normalization step. Type-3 can be further divided into strongly Type-3 and moderately Type-3 according to the degree of syntactical similarity: strongly Type-3 with [0.7, 1) similarity and moderately Type-3 with [0.5, 0.7) similarity. Following the same similarity measure, the clone pairs in Type-4 are annotated on a range of similarity scores from 0 to 0.5, and these pairs account for more than 98% of total clone types. In our experiment, we use the same training, validating and evaluating sets as the state-of-the-art method (Guo et al., 2020; Wang et al., 2020a) to train and evaluate the models. The benefit is twofold: (1) we use exactly the same training/validating/evaluating from Guo et al. (2020) for fair comparison; (2) the dataset contains 1,731,860 code pairs in total, which is sufficient for estimating the performance of large-scale neural network models.

[1] https://github.com/YuanJiangGit/Code-Representation-Graph-LSTM.git.

[2] https://sse.hit.edu.cn/train/login.aspx.

[3] https://github.com/YuanJiangGit/Code-Representation-Graph-LSTM/blob/master/resources/dataset/OJ-All/programs.pkl.

5.3. Experiment settings

In this work, we adopt Progex[4] and Joern[5] to generate the PDGs for Java and C programs respectively, and employ tree-sitter-java[6] and tree-sitter-c[7] to parse Java and C programs into ASTs respectively. For both tasks, the token embedding matrix M in the syntactic encoder was initialized by word2vec (Mikolov et al., 2013), a word embedding technique, trained on respective training sets to capture the semantic relations between tokens. For convenience of implementation, we use gensim package (version 3.0.1) for word2vec embedding with all default settings. The embedding size of each token vector is set to 128. The syntactic encoder and semantic encoder adopt the exactly same architectures and parameters to reduce the complexity of the model. That is, any changes to one encoder will affect the other one. The hidden layer dimension of the two encoders is set to 256. The task-specific classifier is a feed-forward and fully-connected network with a single, 100-dimensional hidden layer and the ReLU activation function. In addition, to train the model, we use the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.001 and a batch size of 64. Our approach is implemented with the Pytorch library in Python with an Intel(R) Core i7@3.2 GHz, 64 GB DDR4-RAM and 1 GeForce RTX 2080Ti GPU.

5.4. Evaluation metrics

Code clone detection and program classification tasks can be converted into binary and multi-class classification problems respectively, as illustrated in Section 2.3. Therefore, we use accuracy, precision, recall and F-measure to evaluate the effectiveness of our proposed method. These metrics are defined as follows: $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$, $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F-measure = 2 * \frac{precision*recall}{precision+recall}$, where TP is true positives, FP is false positives, FN is false negatives and TN is true negatives.

Note that, to maintain consistency in evaluation metrics with previous approaches (Zhang et al., 2019a; Fang et al., 2020), we report the test accuracy metric on program classification task, while, on code clone detection task, we report the precision, recall and F-measure value.

5.5. RQ1: How does our approach perform in program classification compared with the state-of-the-art approaches?

In this section, experiments are conducted to demonstrate the effectiveness of the proposed approach on the program classification task. To achieve this, we compare our approach with the following existing baseline methods:

- **TextCNN** and **LSTM** are the standard models for text classification in NLP. Since one simple way to model source code is to just view it as plain text (Hu et al., 2018a; Dam et al., 2016; Hellendoorn and Devanbu, 2017), we can use the same language models (e.g., TextCNN and LSTM) as used in NLP to extract code semantics directly from the "token sequences" of a program.
- **TBCNN** (Mou et al., 2016) and **ASTNN** (Zhang et al., 2019a) are popular tree-based methods. The former uses the custom convolutional neural network to learn program representations on the entire ASTs, while the latter first splits each large AST into a sequence of small tree structures, and then encodes these structures to vectors via the augmented recurrent neural network recursively. The two methods can capture both the lexical and syntactical semantic of statements.
- **(PDG or augmented AST** Allamanis et al., 2017)+**GGNN** are two graph-based methods, where the former uses PDGs to represent the semantic structure of source code while the latter introduces a new graph representation (i.e., augmented AST) to represent both the syntactic and semantic information of source code. Both methods employ a graph-based neural network (i.e., Gated Graph Neural Network (GGNN) Li et al., 2015) to learn code semantics over program structures.
- **Transformer based methods** are a set of transfer learning methods based on the transformer architecture, which firstly pre-trained on a large-scale general-purpose dataset by using a self-supervised task, and then fine-tuned on specialized datasets to achieve better generalization (Mastropaolo et al., 2021). In this paper, we compare our approach with two strong transformer baselines: (1) CodeBert (Feng et al., 2020) and (2) GraphCodeBert (Guo et al., 2020). Note that we use the encoder part of these methods as the feature extractor, followed by a classifier for the program classification task.

Table 1
Test accuracy between our method and other methods on the program classification task.

Groups	Model	OJ-Data-1	OJ-Data-2	OJ-All
Lexical-based	TextCNN	0.882	0.891	0.835
	LSTM	0.905	0.864	0.869
Tree-based	TBCNN	0.949	0.838	0.886
	ASTNN	0.980	0.928	0.953
Graph-based	PDG+GGNN	0.908	0.876	0.885
	Augmented AST+GGNN	0.925	0.908	0.919
Transformer-based	CodeBert	0.927	0.879	0.90
	GraphCodeBert	0.982	0.914	0.947
	Our method	**0.987**	**0.951**	**0.966**

To make a fair comparison, we use 128 as the embedding size, 256 for hidden size and 64 for batch size for all baselines except for transformer based methods. As for the transformer based methods, we use the default architecture and hidden size in order to reuse their pre-trained models. In addition, if some baselines (e.g., TBCNN, ASTNN and transformer based methods) are open-sourced, we employ the default settings for other hyper-parameters including the learning rate and optimizer in our experiments, otherwise, we use the same configuration as our method. More detailed discussions on the number of model parameters are presented in Section 6.2.

Since the OJ datasets used in the program classification task is quite balanced: among all programming programs, each of them corresponds to 500 programs. We can use the accuracy metric for different evaluation models on the testing dataset to make a fair comparison. The experimental results are shown in Table 1.

Table 1 shows that the transformer based models outperform the other DL-based methods by a clear margin. This is not surprising, since these models have a large number of parameters and thus show powerful learning capabilities in extracting a rich variety of information from code data. Our observation is also in line with Guo et al. (2020) and can be further confirmed by rigorous statistics about the model parameters as discussed in Section 6.2 (See Table 6 for details). In addition, as seen in Table 1, GraphCodeBert is the winner among the transformer based models since it provides several ways of communicating the code structure to the transformer instead of treating source

[4] https://github.com/ghaffarian/progex.git.
[5] https://github.com/octopus-platform/joern.git.
[6] https://github.com/tree-sitter/tree-sitter-java.
[7] https://github.com/tree-sitter/tree-sitter-c.

code as simple sequence data as in the CodeBert. Among DL based methods, AST-based methods (e.g., ASTNN) perform better than the token-based methods (e.g., TextCNN and LSTM) on almost all OJ datasets. This is because these token-based methods fail to consider the structural information of source code which carries additional semantics beyond the lexical terms. In addition, although TBCNN and ASTNN have been designed based on ASTs, these approaches have a large variance in performances on the program classification task. Our results show that ASTNN performs better than the TBCNN, which indicates the tree-based LSTM is more effective than the tree-based CNN in capturing code semantics (Yu et al., 2019). A possible explanation of this phenomenon is that most real-world programs' AST can be very large in syntax nodes and deep in tree layers due to the complexity of programs, which results in the failures for TBCNN to model the long-distance context information from the raw ASTs. In an attempt to tackle this problem of tree-based CNN in dealing with complex programs, ASTNN, a recent approach to AST embedding proposed by Zhang et al. (2019a), firstly splits a large AST into a sequence of small tree structures, and encodes these small trees into statement vectors that are then aggregated into the program-level feature.

In addition to comparing the performance difference between tree-based CNN (i.e., TBCNN) and tree-based LSTM (i.e., ASTNN), we also compare tree-based methods with the graph-based methods (i.e., PDG+GGNN and Augmented AST (Allamanis et al., 2017) + GGNN). As can be seen from Table 1, the graph-based methods perform worse than the state-of-the-art Tree-based method ASTNN. This may be due to the following reasons: (1) for the PDG+GGNN, it uses an average embedding vector of all tokens in each PDG node as its initial embedding (Allamanis et al., 2017) in GGNN (Li et al., 2015), which causes the loss of local semantic information in the source code; (2) for the Augmented AST (Allamanis et al., 2017) + GGNN, it integrates a lot of additional edges among tree nodes to the AST, which may contain some redundant information and noisy data, and thus leads to lower performance. These observations are consistent with the previous studies (Jayasundara et al., 2019; Fang et al., 2020).

In addition, the results shown in Table 1 reveal that our approach outperforms all baseline methods, achieving a much higher accuracy on the three OJ datasets, which is the best result that can be achieved. This demonstrates the superiority of our approach in capturing both the structural and sequential semantic information using the proposed representation framework.

Conclusions: The experimental results demonstrate that the proposed approach outperforms the state-of-the-art code representation methods on the program classification task, which indicates its superiority in capturing more accurate and comprehensive semantic information from source code than the token-based, tree-based and graph-based and existing transformer-based methods.

5.6. RQ2: How does our approach perform in code clone detection compared with the state-of-the-art approaches?

To validate the effectiveness and generalizability of our method on the code clone detection task, we compare our proposed method with the following eight baseline methods conducted on the same datasets:

- **SourcererCC** (Sajnani et al., 2016) is a state-of-the-art token-based clone detector for large inter-project repositories, which can detect multiple types of clones (e.g., from Type-1 to Type-3 clones), albeit with a bag-of-tokens representation.

Table 2
Comparisons between our method and other methods on the clone detection task.

Methods	BigCloneBench			OJClone		
	Precision	Recall	F1	Precision	Recall	F1
SourcererCC	0.88	0.02	0.03	0.07	0.74	0.14
Deckard	0.93	0.02	0.03	0.99	0.05	0.10
DLC	0.95	0.01	0.01	0.71	0.00	0.00
CDLH	0.92	0.74	0.82	0.47	0.73	0.57
FCCD	–	–	–	0.97	0.95	0.96
ASTNN	0.92	0.94	0.93	0.99	0.93	0.96
CodeBert	0.947	0.934	0.941	0.999	0.961	0.979
GraphCodeBert	0.948	0.952	0.950	**1.0**	0.977	0.988
Our method	**0.985**	**0.972**	**0.979**	0.994	**0.995**	**0.994**

- **Deckard** (Jiang et al., 2007) is an AST-based approach, which represents the syntactic level of source code from the aspect of discrete AST embedding and measures code similarity between two code fragments using Euclidean distance.
- **DLC** (White et al., 2016) explores the use of deep learning techniques to address code clone detection, which uses a recursive autoencoder to extract unsupervised deep features.
- **CDLH** (Wei and Li, 2017) leverages AST-based neural network for code clone detection, which can detect a pair of code fragments with semantic similarity but differing in both lexical and syntactical levels.
- **FCCD** (Fang et al., 2020) is an approach recently proposed for ***F**unctional **C**ode **C**lone **D**etection*, which identifies the similar functionality of different code snippets at functionality granularity by analyzing the call graph, and simultaneously learns the syntactic and semantic features from ASTs and CFGs using fusion embedding technique.
- **ASTNN** (Zhang et al., 2019a), similar to that described in Section 5.5, learns the code semantics from a sequence of smaller AST-trees, i.e., the code representation learning using ASTNN on code clone detection task is the same as that on the program classification. The differences between the two tasks lie in the specific layers, aiming for achieving different tasks.
- **CodeBert** (Feng et al., 2020) and **GraphCodeBert** (Guo et al., 2020) are based on the use of pre-trained transformer based models, which leverage a fully attention-based approach to capture the contextual information of source code. More importantly, GraphCodeBert extends the CodeBert by incorporating the semantic structure of source code to learn code representation. From the output of the encoder of CodeBert or GraphCodeBert, we get the vector representations for a code pair, and then measure the similarity between them to determine whether the code pair belongs to a clone.

For each baseline method, we follow the same settings as described in Section 5.5 to make a fair comparison. Table 2 summarizes the experimental results evaluated via standard metrics, such as precision, recall, and F-measure.

It can be seen from Table 2, our approach consistently outperforms the state-of-the-art baselines on the very different datasets (i.e., from C based dataset (i.e., OJClone) to Java based dataset (i.e., BigCloneBench)) in all evaluation metrics. For example, our approach achieves a 2.9% improvement in F-measure and a 2.0% improvement in Recall compared to the state-of-the-art method GraphCodeBert on the BigCloneBench dataset. More importantly, our approach achieves an F-measure performance of 99.4%, which is the best that can be achieved on the OJClone dataset. This illustrates that our approach is proven to be very effective in extracting high-level representations from the raw source code in the code clone detection task.

Table 2 shows that the deep learning based methods (e.g., GraphCodeBert, DLC, CDLH, FCCD and ASTNN) outperform traditional ones (e.g., SourcererCC and Deckard) by a clear margin. We argue that this is due to the fact that traditional approaches utilize discrete representations (e.g., token-based method SourcererCC and AST-based method Deckard) to extract low-level grammatical (i.e., syntactic) information from source code, not capturing high-level syntactic and semantic information; the lack of deep understanding of programs makes it difficult to find all code clone pairs, or easy to report many false positives, and thus produces fairly low recall or low precision. In contrast, as we can see deep learning based methods demonstrate a much powerful performance compared to traditional work.

Among deep learning based methods, DLC performs much worse than the other five deep learning based methods. This is because DLC is an unsupervised learning approach based on recursive autoencoder networks to learn program representations, which cannot directly optimize the training process of the desired task (i.e., clone detection task) and thus unable to improve the performance of detection models. Apart from unsupervised detection methods (e.g., DLC), we notice that the performance difference between supervised methods (e.g., CDLH, FCCD and ASTNN) is considerable. In these techniques, the detection process usually has two steps: (1) feature extraction, where representative (i.e., lexical, syntactic or semantic) features are extracted to capture the essential characteristics of the program under analysis; and (2) categorization, where intelligent techniques (e.g., mapping function) are used to automatically group a pair of programs into different classes (i.e., clone pairs or none-clone pairs) based on computational distances of their feature representations. Obviously, a key to achieving high performance of clone detection is feature representation methods designed for capturing deep code semantics in Step 1. For example, the method CDLH (Wei and Li, 2017) proposed to leverage Tree-based LSTM to learn both the lexical and syntactical aspects of source code. Although this method shows promising results on the code clone detection task (e.g., CDLH achieves an F-measure of 82% on BigCloneBench), it still faces serious limitations, such as the lack of capability to capture the program's semantic information (i.e., control and data dependencies between statements) and no support for learning long-distance correlations in a much bigger tree due to the vanishing gradient problem in Tree-based LSTM. To alleviate these issues, FCCD focuses on better characterizing both the semantic and syntactic features of programs by combining structural embedding based on CFG and the syntactic embedding based on AST, while ASTNN strives to efficiently learn lexical and syntactical knowledge between statements by separating the large AST of the program into several small subtrees. These improvements of FCCD and ASTNN to CDLH are the main reasons that the performance increases significantly. Specifically, compared to the DLC, FCCD and ASTNN improve the clone detection performance in terms of F-measure by nearly 68.4% for the OJClone dataset.

We also notice that there are only slight differences between the results obtained by FCCD and ASTNN, even if the code semantics are considered in FCCD. This could be due to the fact that FCCD adopts a simple strategy to concatenate the learned semantic and syntactic feature vectors of one program into one feature vector. This concatenating strategy is so simple to force some information loss, e.g., it breaks up the connection between the syntax of a single statement and the associated semantic meaning, and it is not suitable to concatenate different feature vectors learned by different methods together in a single vector space. However, the proposed network in this paper first learns the low-level syntactic information from the local context (i.e., sub-tree) for each statement and then combines dependencies between statements to learn the high-level semantic information from the global context. The experimental results also show the superiority of the proposed network on benchmark datasets. More concretely, on the OJClone dataset, our method improves the performance by about 3% in F-measure compared to FCCD.

Compare to the above DL-based methods, the transformer based models generally exhibit an overall better performance on the code detection task. This is because they can capture global or long distance relationships in a sequence via the multi-layer, multi-head attention mechanism. However, this technique has its drawback: running transformer based models is computationally expensive since they contain a large number of trained parameters. More detailed discussions on the model's parameters are presented in Section 6.2.

Conclusions: The deep representation learned from the program provides useful semantic information for the code clone detection task. The experimental results also demonstrate that our proposed method can further capture the complicated semantic correlations between and within statements and achieve a significant performance boost on code clone detection benchmarks (i.e., BigCloneBench and OJClone).

5.7. RQ3: What's the effectiveness of the proposed semantic graph and Graph-LSTM in our method?

In the previous sections (Sections 5.5 and 5.6), we have demonstrated the effectiveness of our approach on two common program comprehension tasks (i.e., program classification and code clone detection). To better understand where the performance improvements of our proposed approach come from, in this section, we conduct further experiments to analyze the effectiveness of the main component (i.e., the semantic graph and Graph-LSTM) in our proposed model.

For this purpose, we evaluate the following three types of variants of our approach on the program classification task using the combined OJ dataset (i.e., OJ-All).

- **(PDG or Augmented AST)+(GCN, GAT or GGNN)**: To verify the effectiveness of the proposed method, we replace semantic graph and Graph-LSTM in our method with other semantic-based representation (i.e., PDG or augmented AST) and the commonly used graph neural network (i.e., GCN Kipf and Welling, 2016, GAT Veličković et al., 2017 or GGNN Li et al., 2015) respectively, and then compare their performance difference on the program classification task.
- **Semantic graph+(GCN, GAT or GGNN)**: To verify the effectiveness of our proposed Graph-LSTM, We replace it in our method with other graph neural networks (i.e., GCN, GAT and GGNN) and then conduct comparison experiments to confirm the usefulness of Graph-LSTM in extracting both global semantic and sequential features.
- **(PDG or Augmented AST)+Graph-LSTM**: To verify the effectiveness of our proposed semantic graph, we replace it in our method with the most commonly used semantic-based code representation (i.e., PDG) or other recently proposed code representation (i.e., Augmented AST), and then conduct comparison experiments to confirm the usefulness of semantic graph in integrating the syntactic and semantic information. Note that when Graph-LSTM is applied to PDG or Augmented AST, breaking cycles and topological-sort for these graphs need to be performed beforehand.

The evaluation is done under the same dataset (i.e., OJ-All) and experimental settings. Concretely, we train all GNNs and our model for 100 epochs with a learning rate of 0.001 and a batch size of 64. The embedding size is set to 128 and the hidden dimensions in each layer for all neural networks are set to 256 to

Table 3
Test accuracy between our method and two groups of variants on the OJ-All datasets.

Groups	Model	Accuracy
(PDG or Augmented AST) + (GCN, GAT or GGNN)	PDG + GCN	0.782
	PDG + GAT	0.864
	PDG + GGNN	0.885
	Augmented AST + GCN	0.849
	Augmented AST + GAT	0.870
	Augmented AST + GGNN	0.919
Semantic graph + GNN	Semantic graph + GCN	0.928
	Semantic graph + GAT	0.936
	Semantic graph + GGNN	0.941
PDG or Augmented AST + Graph-LSTM	PDG + Graph-LSTM	0.932
	Augmented AST + Graph-LSTM	0.952
Our method	Semantic graph +Graph-LSTM	**0.966**

make a fair comparison. In addition, to optimize the performance of GNNs, we vary the number of layers in {3, 6, 9, 12} for GCN, GAT and GGNN, and set the number of time steps for GGNN among {3, 6, 9, 12}. The best experimental results for each variant and our method are given in Table 3, where the bold values indicate those with the highest accuracy.

As can be seen from Table 3, the best performance with an accuracy of 96.6% is acquired when two components (semantic graph and Graph-LSTM) are included. That means our method combining semantic graph and Graph-LSTM is advantageous over the variants using only one. The satisfactory result demonstrates that the proposed method can be successfully employed for representing deep semantic features of programs, which is crucial for the good performance of many downstream software engineering tasks (e.g., program classification). In addition, the experiments uncovered the effectiveness of the individual components. Concretely, compared to the other representations (i.e., PDG and Augmented AST), the proposed semantic graph improves the accuracy of GNNs (e.g., GCN, GAT or GGNN) on the program classification task by a clear margin. This illustrates the semantic graph is capable of preserving both the syntax and semantic structures of a program and thus is more suitable as input for the GNNs to capture the structural semantic information. Similarly, the proposed Graph-LSTM is much more superior than other GNNs. For example, compared to GCN, GAT and GGNN, Graph-LSTM offers an average improvement of 8.8% and 7.2% on the PDG and augmented AST respectively, which shows that the sequential information of the source code is of great help to obtain better representations besides the structural information.

Overall, our proposed Graph-LSTM performs better than GNN-based baselines when given the same input representation of programs (e.g., PDG, Augmented AST or semantic graph), and our proposed semantic graph allows a model (e.g., GCN, GAT, GGNN or Graph-LSTM) to learn a deeper understanding of code semantics more effectively than other code representations (e.g., PDG or Augmented AST). To provide some intuition on how the semantic graph and Graph-LSTM contribute to different degrees of performance improvement, we present the test accuracy curves for different types of variants in 100 epochs (One epoch is an iteration over the OJ dataset) as shown in Fig. 9, where X-axis is for epoch iteration and y-axis is for accuracy computed on the test dataset.

As can be seen from Fig. 9, our proposed model can be trained faster and more effectively as expected, showing that it indeed has the ability to capture deep semantic and sequential information in programs and can abstract high-level features spontaneously, thus benefiting downstream tasks. Another important finding is that the variants of our proposed method with the

Table 4
Running time comparisons between our method and other baselines on the clone detection task using OJClone dataset.

Methods	Training time	Prediction time
SourcererCC	–	30 s
Deckard	–	32 s
DLC	120 s	67 s
CDLH	8307 s	82 s
FCCD	8043 s	63 s
ASTNN	9902 s	72 s
CodeBert	10 807	64 s
GraphCodeBert	18 389 s	165 s
Our method	7420 s	89 s

semantic graph or Graph-LSTM significantly perform better than the variants without it, which highlight the importance of (1) a high-level representation of source code that is suitable for deep learning, (2) a robust neural network that is capable of effectively capturing both the structural and sequential information. Furthermore, we note that the proposed model with both semantic graph and Graph-LSTM has higher performance than the variant using only one. This is consistent with our previous experimental observation in Table 3.

Conclusions: We evaluate the effectiveness of the individual components proposed in this paper by analyzing the performance of possible combinations of them. Our experimental results show the proposed semantic graph and Graph-LSTM are greatly beneficial for learning more accurate code representation and bringing substantially faster convergence.

5.8. RQ4: How efficient is our approach during the training and testing phase?

In this section, we compare the time performance between our proposed method and the baselines in the clone detection task. We run each method to detect code clones on the OJClone dataset with the optimal parameters. Also we perform our experiment multiple (e.g., five) times and report the average over the multiple runs.

Table 4 reports the training and predicting time of each method. We can see that, on the OJClone dataset, our approach takes about 7420 s for training model. For prediction, our approach takes 89 s for all testing samples. That is, 0.247 s and 0.009 s for each sample in the training and testing dataset, respectively. Compared with state-of-the-art approaches except for DLC, the proposed method offers a better efficiency in training detection model.

Also note SourcererCC and Deckard do not require training the model and perform clone detection on the source code of the project directly. Therefore, they do not have corresponding training time, which are padded with "–" in Table 4. Additionally, we found that the supervised deep learning methods (e.g., CDLH, FCCD, ASTNN) take more actual training time to yield satisfactory results than the unsupervised detection method DLC. This is because for these methods, a deep encoder of the full AST of the program is generally developed to generate a fixed-length vector that represents the syntactic information of the program. However, due to the large shapes and sizes of the ASTs, the encoder process needs a lot of time to recursively traverse the parse trees to perform vector computations for children's and parent's nodes, which takes up the most part of total training time overhead. The unsupervised detection method (i.e., DLC) adopts a simple network to learn feature representation of programs automatically from source code, which consumes much less time (Fang et al.,

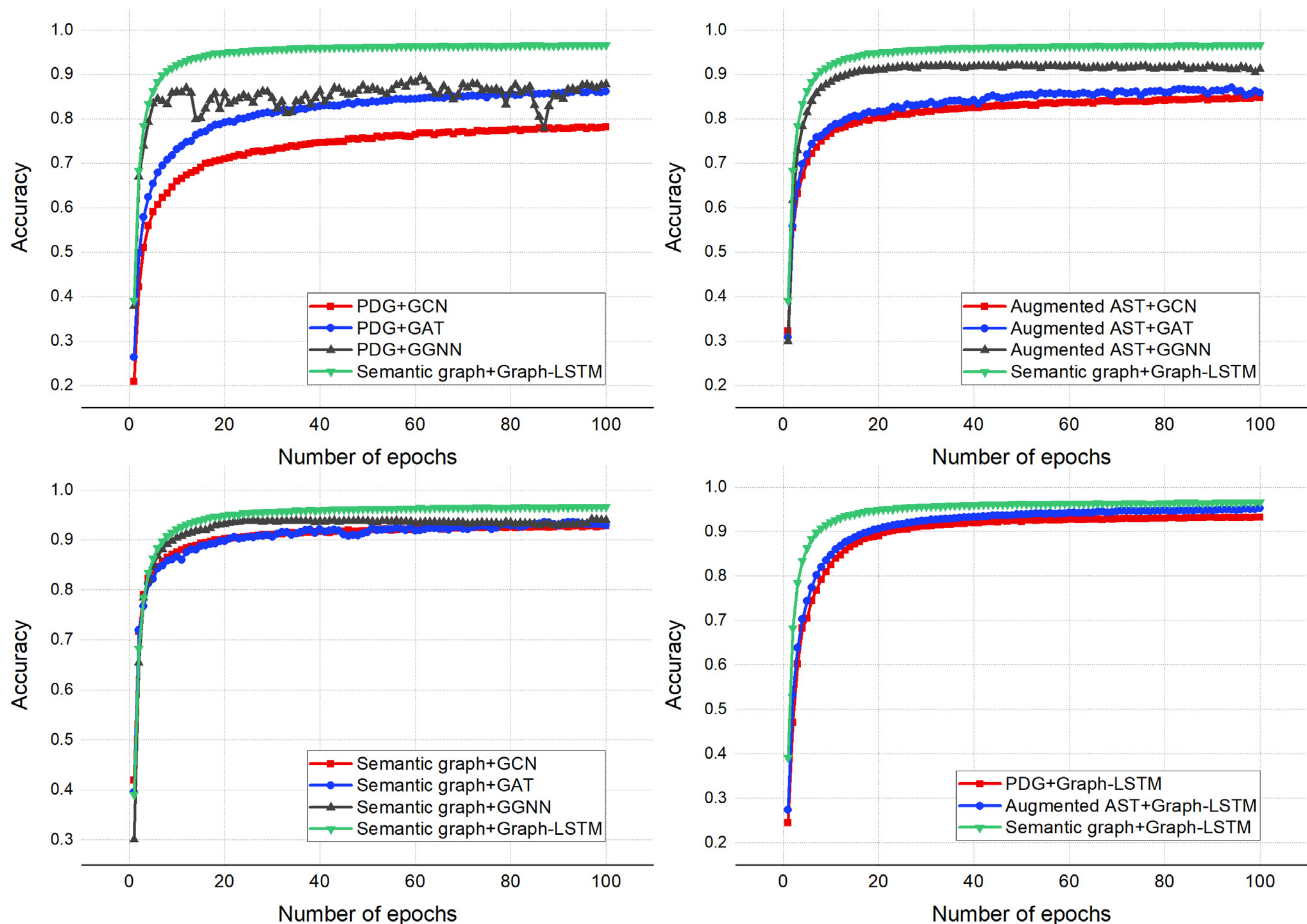


Fig. 9. Comparison of Accuracy for our variants using their best configurations. Semantic graph+Graph-LSTM achieves the best performance, i.e., the highest test Accuracy, compared to the other variants.

2020). Unsurprisingly, the transformer based models (e.g., CodeBert and GraphCodeBert) take the longest time to train since they use much more complex architectures and often consist of over 100 million parameters. Moreover, in order to leverage the graph structure of code to learn code representation, GraphCodeBert introduces a complex graph-guided masked attention to encode the dependency relation between variables and thus requires longer time to train in contrast to BERT. In addition, there are no noticeable differences in the prediction time for most existing DL-based methods. Furthermore, the prediction time could be amortized for all testing samples. Therefore, a few seconds varying for the entire testing dataset is almost negligible.

Conclusions: From the results, we observe that the proposed approach achieves promising and competitive training efficiency when compared with the state-of-the-art methods. The main reason is that our method does not perform time-consuming learning on full ASTs. However, even so, the proposed method outperforms the state-of-the-arts by a large margin as shown in Section 5.6.

6. Discussion

In this section, we first discuss the different behaviors and the major benefits of our proposed approach over the baseline methods. Then we understand the embeddings computed by the proposed method through visualization. Finally, we discuss the potential threats to the validity of our study at this point.

6.1. Investigating whether breaking cycles forces some semantic information loss during learning

As we stated in Section 3.2, the sequential information of statements being processed is important to deep semantic understanding of the source code. In determining the topological order of statement processing, breaking cycle process is inevitable. To this end, we propose a way of removing cycles in the constructed program graph via the DFS-based method. Although we have given experimental evidence that our proposed model learned over the acyclic semantic graph in topological order can give better results than existing state-of-the-art graph-based methods, more investigations are needed to understand whether or not removing cycles from the constructed program graph forces some semantic information loss during learning. To achieve this, we conduct the following experiments using the combined OJ dataset (i.e., OJ-All).

First, we parse each program to construct an integrated program graph. Second, we remove the cycles in the integrated program graph to generate the corresponding semantic graph. Third, we apply the GNNs on the integrated program graph and semantic graph, separately to compute the graph embedding, and obtain a feature vector for each program, thus fulfilling the program classification task. The most commonly used graph neural networks GCN, GAT and GGNN are the chosen techniques for our study. The experimental results are shown in Table 5.

As seen from the results, there is no significant difference between the two groups of performances when applying GNNs

Table 5
Test accuracy when applying GNNs to the integrated program graph or semantic graph on the program classification task using the combined OJ dataset (i.e., OJ-All).

Model	Integrated program graph	Semantic graph
GCN	0.931	0.928
GAT	0.935	0.936
GGNN	0.927	0.941

Table 6
Comparison of model capacity for the program classification task.

Model	Layers	Parameters
TextCNN	2	0.39M
Bi-LSTM	2	2.79M
TBCNN	2	2.42M
ASTNN	–	2.13M
PDG+GGNN	3	2.94M
Augmented AST+GGNN	6	4.32M
CodeBert	12	124.72M
GraphCodeBert	12	125.31M
Our method	–	1.92M

to the integrated program graph or semantic graph. This is most likely due to the fact that many dependence edges among subtrees in the integrated program graph may be redundant or not be highly correlated with class labels, therefore removing them has less impact on the model performance.

6.2. Comparison of the model capacity between different methods

Previous research work has documented that model capacity is a potential threat to any comparison between different types of models (Guo et al., 2020). Therefore, to get a fair comparison, the selected baselines should span a similar parameter count range with the proposed model. However, evaluating the performance of these models with such experimental settings might severely underestimate their true ability because the impressive effectiveness of most state-of-the-art models (e.g., CodeBert and GraphCodeBert) can be partially attributed to their large number of parameters (Soldaini and Moschitti, 2020). To avoid underestimating the performance of baselines, we use the experimental setting as we detailed in Sections 5.5 and 5.6 to conduct the experiments in this study. As can be seen in Tables 1 and 2, our proposed method outperforms the baselines and get satisfactory improvement on two program understanding tasks. The high performance of our method may be due to the fact that it can capture the information related to not only syntax and dependency relations, but also the sequential semantics of statements being processed. However, we still have no insight on whether or not the relatively higher performance of our method is due to the fact that it has more trainable parameters than the state-of-the-art methods. Therefore, in this section, we seek to answer the following question: Is the proposed model benefiting from more parameters than other baselines. To investigate this, we give the statistics on the amount of parameters that have to be estimated for all baselines and our method as shown in Table 6.

As can be seen from the table, most baselines except for TextCNN have more trainable parameters than our method. This indicates that the improvement in performance might not only owe to a greater number of parameters but also benefit from suitable architectures of neural networks. Thus, it is fair to say that our method offers the competitive performance but uses less parameters compared to many larger DL-based models. In addition, some researchers have pointed out that training models with one hundred million (M) parameters is not feasible and practical for most users (Bhargava, 2020). Therefore, it is of interest to study models that not only achieve satisfactory performance but also rely on simpler model architecture.

Table 7
Performance comparisons between our method and other methods on the vulnerability detection task.

Method	Accuracy	Precision	Recall	F1
Flawfinder	0.698	0.228	0.296	0.257
RATS	0.672	0.128	0.147	0.137
Checkmarx	0.729	0.309	0.432	0.361
VUDDY	0.712	0.980	0.099	0.164
VulDeePecker	0.922	0.582	0.876	0.666
SySeVR	0.954	0.921	0.927	0.924
Devign	0.960	0.930	0.928	0.929
Our method	**0.972**	**0.952**	**0.947**	**0.950**

6.3. Investigating whether the proposed model is effective on a more difficult task

As described in previous sections, our evaluation on two widely used benchmark datasets (OJ-Data and BigCloneBench) shows the effectiveness of our method in achieving state-of-the-art performance in both program classification and clone detection tasks. However, the majority of the instances in both datasets are of shorter length and thus may be easier to understand by code representation models. In this section, we would like to investigate whether our approach can be effectively applied to a more difficult task. To this end, we choose the vulnerability detection task with a public dataset released by Li et al. (2021b) to further evaluate the generalization ability of our model. The overall procedure for applying our approach to vulnerability detection can be simply summarized in the following three steps. First, we preprocess program representations into semantic graphs. Second, we learn vulnerability features directly from the constructed semantic graph by means of the proposed Graph-LSTM. Third, we train a one-layer fully-connected feed-forward neural network to predict the vulnerability-proneness of code. More details of the above steps can be seen in the supplemental materials.[8] We choose several state-of-the-art baselines for comparison and use the default parameters in the literature by the authors for their assessment experiments. The experimental results on four evaluation metrics (i.e., Accuracy, Precision, Recall, F-measure) are shown in Table 7, where the best result is shown in bold face.

From the results, we can make the following key observations. (1) The open-source rule-based vulnerability detection tools (e.g., Flawfinder Wheeler, 2006, RATS Fortify, 2014 and Checkmark SecureSoftware, 2018) perform worse on average (i.e., lower recall and precision) than the DL-based baselines, which can be attributed to the limited expressiveness of human-written rules and thus fail to deal with complex or novel vulnerable cases; (2) The DL-based methods (e.g., VulDeePecker Li et al., 2016 and SySeVR Li et al., 2021b) shows significant performance improvement over the previous rule-based methods, which may be due to the powerful capability of deep learning on automatic feature characterization (Li et al., 2021a); (3) Among these baselines, Devign performs the best in all four metrics, which demonstrates the effectiveness of graph neural networks for better encoding code structural semantics than the regular recurrent neural networks on the task of vulnerability detection (Zhou et al., 2019); (4) Our method achieves a state-of-the-art performance with the overall F1 score of 0.95, which indicates our proposed semantic graph and Graph-LSTM are effective to characterize vulnerabilities with high diversity and complexity in real-world source code by learning comprehensive program semantics.

8 https://1drv.ms/b/s!As5Z1c5DW_I-nEp0FL8xwa2_CXiU.

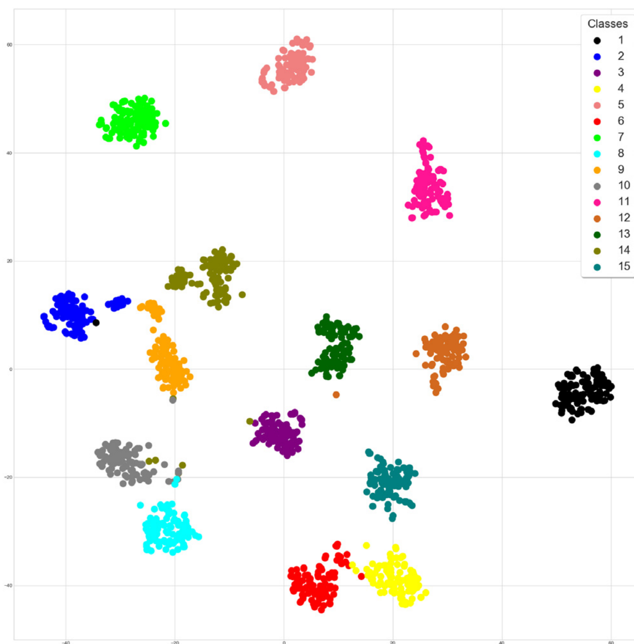


Fig. 10. A 2D visualization of code embeddings of testing programs produced by our method on the program classification task using t-SNE. The legend presents the functional categories of projected programs.

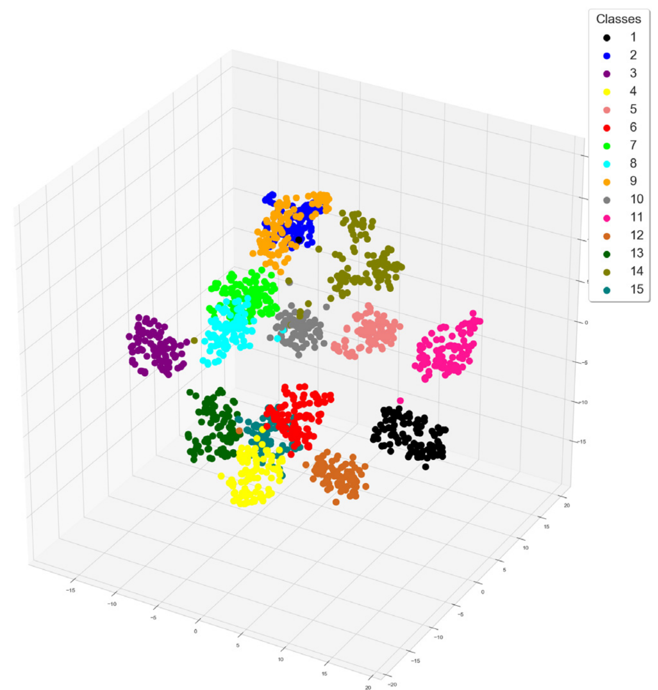


Fig. 11. A 3D visualization of code embeddings of testing programs produced by our method on the program classification task using t-SNE. The legend presents the functional categories of projected programs.

6.4. Understanding the embeddings

We visualize the embeddings of testing programs produced by our method to understand why the proposed model is effective to make accurate predictions. In particular, we choose the first 15 class labels, each of which corresponds to a programming problem and consists of around 100 source programs from the testing dataset of OJ benchmark. Next, we compute the embeddings of these programs using the proposed network trained on the corresponding training dataset. Then, we project the high-dimensional embeddings onto 2- and 3-dimensional spaces using t-SNE (Van Der Maaten, 2014). Finally, we plot the 2- and 3-D projected points in Figs. 10 and 11 respectively, where source programs with the same functionality (i.e., aiming to solve the same programming problem) are presented in the same color. We can find out that (1) for programs with the same functionality, their embeddings are close to each other; and (2) for programs with different functionalities, their embeddings are far from each other, i.e., the boundaries between programming problems are clear. Therefore, this visualization illustrates that our proposed framework learns a robust code representation network that can generate low-dimensional representations of previously unseen programs and preserve the semantic information of the target functionalities in their embeddings.

6.5. Threats to validity

6.5.1. Threats to internal validity

Threats to internal validity are related to our implementation or experimental bias. To help avoid experimental bias in this regard, many performance metrics were provided, including accuracy, precision, recall and F-measure. In addition, to ensure that our implementations of baseline models reflect the original performance, we tested our implementation using the same dataset and evaluation metrics. We report the results of our experiments with these implementations if they produce almost similar or consistent results, otherwise we choose to adopt previously published baseline results compared to ours. Also, to avoid implementation bias of our method, we have double checked our implementation and experiments to reduce this risk to the extent possible.

6.5.2. Threats to external validity

External threats are related to the generalizability of our findings. Firstly, to expediently evaluate the results, we employ two widely-used tasks, i.e., program classification and code clone detection tasks, to demonstrate the effectiveness of the proposed approach compared to the state-of-the-art methods, such as ASTNN and GraphCodeBert. However, we still have no insight on whether the proposed method is valid targeting other software engineering tasks, e.g., code summarization, code completion, code search, code generation and bug detection. Secondly, in terms of datasets, we evaluate our model on OJ and BigCloneBench datasets, which consist of C++ and Java programs respectively. Thus, we have no insight on whether the proposed method performs well on other programming languages as well (e.g., C#).

7. Related work

7.1. Source code representation

Source code representation is critical for machine to automatically understand the program. Therefore, developing efficient and effective code representation models to capture the deep semantics of source code is an important line of research in software engineering. This section highlights previous studies into lexical-based, syntactic-based and semantic-based source code representation methods.

7.1.1. Lexical-based source code representation

In the lexical-based representation, a program is transformed into a sequence of "tokens" via performing lexical analysis, and then the sequence is fed into the code representation model

to get the corresponding feature vector. Traditional Information Retrieval (IR) and Machine learning (ML) based methods have been actively adopted to the lexical-based code representation and have shown good performance in some downstream tasks (e.g., code clone detection). For example, SourcererCC (Sajnani et al., 2016) exploited an optimized inverted-index for ordering token blocks, to quickly scan for duplicated token subsequences that indicate potential code clones with a given code block. Allamanis and Sutton (2013) built an n-gram probabilistic language model of source code from a software corpus with more than one billion tokens. Raychev et al. (2014) employed the language model (e.g., n-gram or RNN) to learn patterns of API method calls from a large number of historical code snippets and use them to synthesize code completions for the holes. Although n-grams have proven promising for various program analysis tasks, their effectiveness has been limited due to simply learning models and algorithms that may not have enough capability to learn representations that express some latent syntactic information and semantic feature in a training corpus (Morin and Bengio, 2005; White et al., 2015).

Recently, some studies try making full use of information embedded in token sequences by deep learning approaches. More concretely, Bhoopchand et al. (2016) proposed a neural language model with a sparse pointer network to capture long dependencies that exist in the token sequences, which was used successfully in code suggestion. Allamanis et al. (2016) employed an attention-based convolutional neural network for source code summarization, where the attention mechanism is used to extract time-invariant and long-range topical features of token sequences. Hu et al. (2018b) improved code summarization by adding an additional API sequences encoder to leverage the learned API knowledge from the related task. For code search, Gu et al. (2018) proposed a novel deep learning network called CODEnn to jointly embed code fragments and natural language queries into the same high-dimensional vector space. Additionally, Li et al. (2021b, 2018, 2021a) proposed a new code representation framework for vulnerability detection that represents code snippets consisting of a sequence of tokens as vectors which accommodate syntactic and semantic information pertinent to vulnerabilities.

Due to the good results of transformer based models in various NLP tasks, Feng et al. (2020), developed the first large NL-PL pre-trained Bert model (named CodeBert) for multiple programming languages, and conducts extensive experiments on the tasks of code search and code-to-text generation, showing impressive improvements over the prior DL-based methods. Similar to the above study, Mastropaolo et al. (2021) empirically investigated whether the famous Text-To-Text Transfer Transformer (T5) architecture can yield better performance for Text-To-Text software engineering tasks than previous methods when pre-trained and fine-tuned. Although the transformer based models can achieve satisfactory performance, they suffer from the high cost of training due to the large number of parameters in their model architecture, and thus may not be feasible for most users (Bhargava, 2020). However, the proposed model has comparable performance but with far fewer parameters and a simpler architecture than transformer based methods.

7.1.2. Syntax-based source code representation

Besides the lexical information, syntax-structured information has attracted increasing interest in the field of source code representation. For example, Deckard (Jiang et al., 2007) leveraged discrete numerical vectors to characterize the syntactic features of the program's ASTs, and presented an efficient algorithm to cluster these vectors to find potential code clones. Due to the limitation of the discrete feature representation, the method Deckard can capture little semantic correlation between AST's nodes. To address this, deep learning techniques are increasingly being used to learn code embeddings, continuous distributed vectors for representing the ASTs of programs. The representative works here are Mou's TBCNN (Mou et al., 2016) and Wei's Tree-LSTM (Wei and Li, 2017) where the former leverages a novel tree-based CNN while the latter uses the improved LSTM to capture programs' abstract syntax information. To address the vanishing gradient problem of Tree-LSTM, Zhang et al. (2019a) proposed a novel neural source code representation model ASTNN, which splits each large AST into a sequence of smaller sub-trees, and then learns syntactic knowledge from each sub-tree separately. The above syntax-based models and these variations have been successfully applied in many real-world applications, such as code clone detection (Wei and Li, 2017; Zhang et al., 2019a; Yu et al., 2019), code completion (Liu et al., 2016b; Li et al., 2017), code generation (Rabinovich et al., 2017; Parisotto et al., 2016), code summarization (Hu et al., 2018a; Alon et al., 2018; Wan et al., 2018) and software defect prediction (Dam et al., 2018).

7.1.3. Semantic-based source code representation

Despite the compelling success achieved by these models based on lexical or syntactic representation of source code, they still suffer from problems, such as paying little attention to deep semantic information of source code. Therefore, more and more research focus has been directed towards designing new models or algorithms aiming to capture more real and complex semantics between program elements. Zhao and Huang (2018) proposed a novel network named DeepSim to encode data- and control-flow into a semantic matrix for detecting functionally similar code. Fang et al. (2020) proposed a novel joint code representation model that can simultaneously learn syntactic and semantic features of source code by applying fusion embedding techniques. Ben-Nun et al. (2018) defined a new kind of code representation (i.e., contextual flow) based on both code control flow and data flow to represent the dynamic behavior of a program, and then proposed an unsupervised approach inst2vec to learn the representation for each statement. Allamanis et al. (2017) proposed to convert ASTs to graphs by adding extra edges among tree nodes to represent a variety of code dependencies, and then leveraged GGNNs to capture code semantics in the constructed graph. Tufano et al. (2018) leveraged neural networks to automatically learn code similarities from four different code representations of source code (such as identifiers, AST, CFG and bytecodes) for the code clone detection task. The final similarity between a pair of code fragments is defined as the weighted average of these four similarities. Although in Tufano et al. (2018), extracting features from a variety of source code representations could prove useful for code clone detection, the simplest fusion strategy to combine the learned features at the program level may cause information loss of the correspondence between different types of abstraction representation at the statement level and result in failure to fully and precisely understand program semantics. Different from the study by Tufano et al. (2018), we make the learned local features of statements well aligned with the corresponding global structural and sequential features, since the latter is calculated based on the extracted features of the former. Recently, Guo et al. (2020) developed a new transformer based model named GraphCodeBert to learn the structural information of source code, which is a pre-trained model and gets satisfactory improvement on many SE tasks. However, like any transformer based model (e.g., CodeBert), it has a clear limitation in that it has a large number of parameters and increases the computational cost of training. In addition, a handful of recent studies (Li et al., 2019; Zhou et al., 2019; Wang et al., 2020b) designed new graph-based code representation specializing towards vulnerability (or bug) detection to capture the deep semantics of source code.

For semantic-based code representation, most previous work first represents programs as graphs, which enable code semantics modeling for the downstream tasks using graph neural networks. Although these methods have achieved promising results, they still have challenges, e.g., relying heavily upon the capability of graph neural network techniques and thus resulting in the loss of some useful sequential semantic information. Unlike previous studies, we propose a novel code representation framework to capture both low-level syntactic and high-level semantic knowledge as well as the sequential naturalness of statements.

7.2. Deep learning in software engineering

Recently, deep learning has attracted a great deal of attention in software engineering. Apart from its clear application in program comprehension, this technique can be useful in other research directions, e.g., duplicate bug report detection (Deshmukh et al., 2017), bug localization (Lam et al., 2017; Huo et al., 2016; Huo and Li, 2017), API usage sequences generation (Gu et al., 2016), and commit message generation (Jiang et al., 2017). The aforesaid related work mainly us es deep learning-based neural network techniques to help understand software-related natural language texts for various development and maintenance activities in software engineering, while we focus on the deep semantic representation of source code.

8. Conclusion and future work

In this paper, we highlight the issues and challenges associated with current methods of capturing both the low-level syntactic and high-level semantics of source code. To meet the challenge, in this paper, we first propose a novel code representation (i.e., semantic graph) to represent both the syntactic information and dependency relations of statements and then extend the Tree-LSTM model named Graph-LSTM to learn statement representations over the constructed semantic graph of source code. Next, considering that the sequential semantics is also critical for code understanding, we design a novel way to incorporate sequential information of source code into the code representation learning process. Finally, we conduct extensive performance evaluations to demonstrate the effectiveness of the proposed approach compared to previous work, and discuss it more in-depth about model size.

Our future work involves: (1) applying the proposed approach to other programming languages (e.g., C#); (2) applying the proposed approach to other program comprehension tasks (e.g., code summarization and code completion); (3) expanding the current dataset by mining large open source software repositories (e.g., github).

CRediT authorship contribution statement

Yuan Jiang: Conceptualization, Data curation, Methodology, Software, Writing – original draft, Writing – review & editing, Formal analysis. **Xiaohong Su:** Supervision, Project administration, Resources, Funding acquisition, Writing – original draft, Writing – review & editing. **Christoph Treude:** Investigation, Software, Writing – original draft, Visualization. **Tiantian Wang:** Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Nos.61672191) and "the 13th Five-Year" National Science and Technology Major Project of China (Grant No. 2017YFC0702204).

Appendix A. Automated vulnerability detection with hierarchical semantic-aware code representation

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111355.

References

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.
Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. PMLR, pp. 2091–2100.
Allamanis, M., Sutton, C., 2013. Mining source code repositories at massive scale using language modeling. In: 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, pp. 207–216.
Alon, U., Brody, S., Levy, O., Yahav, E., 2018. Code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400.
Ambient Software Evoluton Group, 2013. Ijadataset 2.0. http://secold.org/projects/seclone.
Ben-Nun, T., Jakobovits, A.S., Hoefler, T., 2018. Neural code comprehension: A learnable representation of code semantics. In: Advances in Neural Information Processing Systems. pp. 3585–3597.
Bengio, Y., et al., 2009. Learning deep architectures for AI. Found. Trends® Mach. Learn. 2 (1), 1–127.
Bhargava, P., 2020. Adaptive transformers for learning multimodal representations. arXiv preprint arXiv:2005.07486.
Bhoopchand, A., Rocktäschel, T., Barr, E., Riedel, S., 2016. Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307.
Bieber, D., Sutton, C., Larochelle, H., Tarlow, D., 2020. Learning to execute programs with instruction pointer attention graph neural networks. Adv. Neural Inf. Process. Syst. 33, 8626–8637.
Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.
Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T., Kim, C.-J., 2018. A deep tree-based model for software defect prediction. arXiv preprint arXiv:1802.00921.
Dam, H.K., Tran, T., Pham, T., 2016. A deep language model for software code. arXiv preprint arXiv:1608.02715.
Deshmukh, J., Podder, S., Sengupta, S., Dubash, N., et al., 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 115–124.
Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q., 2020. Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 516–527.
Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.
Fortify, H., 2014. Rough audit tool for security. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
Grover, A., Leskovec, J., 2016. node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864.
Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, pp. 933–944.
Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 631–642.
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
Harris, Z.S., 1954. Distributional structure. Word 10 (2–3), 146–162.

Hellendoorn, V.J., Devanbu, P., 2017. Are deep neural networks the best choice for modeling source code?. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 763–773.

Hendrix, D., Cross, J.H., Maghsoodloo, S., 2002. The effectiveness of control structure diagrams in source code comprehension activities. IEEE Trans. Softw. Eng. 28 (5), 463–477.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018a. Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, pp. 200–20010.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018b. Summarizing source code with transferred api knowledge.(2018). In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelli-Gence (IJCAI 2018), Stockholm, Sweden, 2018 July 13, Vol. 19, pp. 2269–2275.

Huo, X., Li, M., 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: IJCAI. pp. 1909–1915.

Huo, X., Li, M., Zhou, Z.-H., et al., 2016. Learning unified features from natural and programming languages for locating buggy source code. In: IJCAI, Vol. 16. pp. 1606–1612.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2073–2083.

Jayasundara, V., Bui, N.D.Q., Jiang, L., Lo, D., 2019. Treecaps: Tree-structured capsule networks for program source code processing. arXiv preprint arXiv:1910.12306.

Jiang, S., Armaly, A., McMillan, C., 2017. Automatically generating commit messages from diffs using neural machine translation. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 135–146.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, pp. 96–105.

Karmakar, A., 2019. Establishing benchmarks for learning program representations. In: SATToSE.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kipf, T.N., Welling, M., 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., Socher, R., 2016. Ask me anything: Dynamic memory networks for natural language processing. In: International Conference on Machine Learning. PMLR, pp. 1378–1387.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2017. Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, pp. 218–229.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86 (11), 2278–2324.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Li, J., Wang, Y., Lyu, M.R., King, I., 2017. Code completion with neural attention and pointer networks. arXiv preprint arXiv:1711.09573.

Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Program. Lang. 3 (OOPSLA), 1–30.

Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2021a. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans. Dependable Secure Comput..

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201–213.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput..

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Liu, P., Qiu, X., Huang, X., 2016a. Recurrent neural network for text classification with multi-task learning. arXiv preprint arXiv:1605.05101.

Liu, C., Wang, X., Shin, R., Gonzalez, J.E., Song, D., 2016b. Neural code completion.

Mastropaolo, A., Scalabrino, S., Cooper, N., Palacio, D.N., Poshyvanyk, D., Oliveto, R., Bavota, G., 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 336–347.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Morin, F., Bengio, Y., 2005. Hierarchical probabilistic neural network language model. In: Aistats, Vol. 5. Citeseer, pp. 246–252.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S., 2017. Graph2vec: Learning distributed representations of graphs. arXiv preprint arXiv:1707.05005.

Ou, M., Cui, P., Pei, J., Zhang, Z., Zhu, W., 2016. Asymmetric transitivity preserving graph embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1105–1114.

Pantel, P., 2005. Inducing ontological co-occurrence vectors. In: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05), pp. 125–132.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., Kohli, P., 2016. Neuro-symbolic program synthesis. arXiv preprint arXiv:1611.01855.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535.

Raychev, V., Vechev, M., Yahav, E., 2014. Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 419–428.

Russell, R.L., Kim, L., Hamilton, L.H., Lazovich, T., Harer, J.A., Ozdemir, O., Ellingwood, P.M., Mcconley, M.W., 2018. Automated vulnerability detection in source code using deep representation learning, arXiv:arXiv:1807.04320v1.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V., 2016. SourcererCC: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering, pp. 1157–1168.

SecureSoftware, 2018. Checkmarx. https://www.checkmarx.com/.

Sheneamer, A., Kalita, J., 2016. A survey of software clone detection techniques. Int. J. Comput. Appl. 137 (10), 1–21.

Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., Matsumura, T., 2019. Automatic source code summarization with extended tree-lstm. In: 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8.

Sneed, H.M., Dombovari, T., 1999. Comprehending a complex, distributed, object-oriented software system: A report from the field. In: Proceedings Seventh International Workshop on Program Comprehension. IEEE, pp. 218–225.

Soldaini, L., Moschitti, A., 2020. The cascade transformer: an application for efficient answer sentence selection. arXiv preprint arXiv:2005.02534.

Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M., 2014. Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 476–480.

Tai, K.S., Socher, R., Manning, C.D., 2015. Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075.

Tip, F., 1994. A Survey of Program Slicing Techniques. Centrum voor Wiskunde en Informatica Amsterdam.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. Deep learning similarities from different representations of source code. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). IEEE, pp. 542–553.

Van Der Maaten, L., 2014. Accelerating t-SNE using tree-based algorithms. J. Mach. Learn. Res. 15 (1), 3221–3245.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks. arXiv preprint arXiv:1710.10903.

Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A., 1994. Accurate static estimators for program optimization. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 85–96.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 397–407.

Wang, D., Cui, P., Zhu, W., 2016. Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1225–1234.

Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020a. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 261–271.

Wang, Y., Wang, K., Gao, F., Wang, L., 2020b. Learning semantic program embeddings with graph interval neural network. Proc. ACM Program. Lang. 4 (OOPSLA), 1–27.

Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.. In: IJCAI. pp. 3034–3040.

Wheeler, D., 2006. Flawfinder home page. Web Page: http://www.dwheeler.com/flawfinder.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 87–98.

White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., 2015. Toward deep learning software repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 334–345.

Wotawa, F., Krenn, W., 2007. Knowledge extraction from c-code. In: 2007 Fifth Workshop on Intelligent Solutions in Embedded Systems. IEEE, pp. 49–60.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. Proceedings - IEEE Symposium on Security and Privacy 590–604. http://dx.doi.org/10.1109/SP.2014.44.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, pp. 70–80.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019a. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 783–794.

Zhang, S., Yao, L., Sun, A., Tay, Y., 2019b. Deep learning based recommender system: A survey and new perspectives. ACM Comput. Surv. 52 (1), 1–38.

Zhao, G., Huang, J., 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, pp. 141–151.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:1909.03496.

Yuan Jiang, is a Ph.D. candidate at the School of Computer Science and Technology, Harbin Institute of Technology. His main research interests are in the areas of mining software repository, software vulnerabilities detection, source code representation.

E-mail: jiangyuan@hit.edu.cn

Xiaohong Su, is a professor at the School of Computer Science and Technology, Harbin Institute of Technology. Her research interests include Intelligent software engineering, software vulnerability identification, code representation learning, bug triaging and localization, clone detection, and code search.

E-mail: sxh@hit.edu.cn

Christoph Treude, is a Senior Lecturer in the School of Computing and Information Systems at the University of Melbourne, Australia. He received his Ph.D. in computer science from the University of Victoria, Canada in 2012. The goal of his research is to advance collaborative software engineering through empirical studies and the innovation of tools and processes that explicitly take the wide variety of artifacts available in a software repository into account. He currently serves on the editorial board of the Empirical Software Engineering journal and was general co-chair for the IEEE International Conference on Software Maintenance and Evolution 2020.

E-mail: christoph.treude@unimelb.edu.au

Tiantian Wang, born in 1980. Received the Doctor's degree from Harbin Institute of Technology, Harbin, Heilongjiang, China, in 2009. Since 2013, she has been an Associate Professor in computer science department of Harbin Institute of Technology. Her current research interests are software engineering, program analysis and computer aided education.

E-mail: wangtiantian@hit.edu.cn