Program Classification Using Gated Graph Attention Neural Network for Online Programming Services

Mingming Lu, Member, IEEE, Dingwu Tan, Naixue Xiong, Senior Member, IEEE, Zailiang Chen, and Haifeng Li, Member, IEEE

Abstract—The online programing services, such as Github, TopCoder, and EduCoder, have promoted a lot of social interactions among the service users. However, the existing social interactions is rather limited and inefficient due to the rapid increasing of source-code repositories, which is difficult to explore manually. The emergence of source-code mining provides a promising way to analyze those source codes, so that those source codes can be relatively easy to understand and share among those service users. Among all the source-code mining attempts, program classification lays a foundation for various tasks related to source-code understanding, because it is impossible for a machine to understand a computer program if it cannot classify the program correctly. Although numerous machine learning models, such as the Natural Language Processing (NLP) based models and the Abstract Syntax Tree (AST) based models, have been proposed to classify computer programs based on their corresponding source codes, the existing works cannot fully characterize the source codes from the perspective of both the syntax and semantic information. To address this problem, we proposed a Graph Neural Network (GNN) based model, which integrates data flow and function call information to the AST, and applies an improved GNN model to the integrated graph, so as to achieve the state-of-art program classification accuracy. The experiment results have shown that the proposed work can classify programs with accuracy over 97%.

Index Terms—Abstract syntax tree, attention mechanism, code understanding, function call, data flow, graph neural network, program classification

I. INTRODUCTION

Recently, numerous large-scale online programming services, focusing on programming sharing, exercising, and learning, have received increasing attention. These online services have promoted a lot of social interactions among the service users, such as programmers, educators, and IT companies. For example, Github [1] motivates programmers to share their codes and promote cooperation among programmers; TopCoder [2] provides a platform for software design & implementation, which inspires IT companies to recruit potential employees from the top coders; EduCoder [3] constructs a platform for starters to learn and exercise programming, which encourages the communication between programming educators and programming starters.

However, in the existing online programming services, users have to manually explore the resources that best match their interests. For example, in Github, most cooperated projects rely on a single programmer to complete the most significant part of the projects [4], due to the rapid increasing of source-code repositories, which make it hard to match the requirement of the projects and the capability of project contributors; similarly, in TopCoder, it is difficult to match the requests from the IT companies to the programming skills from the programmers; finally, in EduCoder, it is a challenge to suggest the best programming exercises to starters according to their programming levels.

The emergence of source code mining [5] provides a promising way to analyze the source codes submitted by service users. By analyzing those source codes through various machine learning and deep learning techniques, those programming online services can help understand and classify the programming projects, and make suggestions for social interactions among service users accordingly. For example, source code mining can classify the shared source codes in Github, TopCoder, and EduCoder, according to the semantic meaning of the codes. Based on the code/program classification, those online programming services can analyze the programming capability of the program authors and recommend the programs to the appropriate service users. Moreover, the program representation learned by program classification can pave the way to understand the source codes [6].

Inspired by the Natural Language Processing (NLP), at the early stage of source code mining, researchers tended to regard source codes as text sequences and apply NLP models to the source codes to perform various prediction tasks [7]–[11]. However, unlike nature languages, which have a relatively fixed vocabulary, the possible class/function/variable names appear in source codes can be unlimited. Therefore, two programs with the same syntax structure and the same semantic meaning, but with different variable names, may be regarded as different programs, as illustrated by the example codes shown in Figure 1.

To address the above problem, several works [12]–[17] proposed to adopt Abstract Syntax Tree (AST) to characterize the syntax structure information of the source code. The AST corresponding to the program in Figure 1(a) is shown in Figure 2 which is also the AST for the code in Figure 1(b).

Note that the function/variable names shown in Figure 3 is just for illustration, which in fact do not appear in the AST.

Although the AST-based techniques can utilize the syntax information from the source codes, they cannot reflect numerous semantic information embedded in the source codes. Therefore, recent works [18], [19] proposed to attach addi-
The same code with different variable names.

Fig. 1: An example code.

The graph model used in this paper is a directed graph $G = (V, E)$, among which $V$ represents the node set of size $|V|$ and depicts the directed edge set of size $|E|$. A node in $V$ is identified by a number and a directed edge in $E$ is expressed by $e_{ij}$, representing a directed edge from node $i$ to node $j$.

### II. RELATED WORK

In the earlier stage of source code mining, most of the existing works focused on the NLP-based techniques, which process source codes as tokens [7], [8], [17] or APIs [9], [10] sequences. The related tasks are to predict sequences composition, translation between natural language and code, code completion, and etc. Hindle et al. [8] believed that source code has certain statistical properties, similar to natural languages. Thus, they used n-gram, a popular statistical model in NLP, to model source codes and illustrated that their model can enhance the Eclipse’s ability to complete Java code. Bhoopchand et al. [20] used the Pointer Network model for end-to-end training of source code to achieve IDE code completion task. Gu et al. [9] adopted Encoder-Decoder based RNN to obtain API usage sequences from API-related natural language queries.

To extract syntactic and semantic information from source code, Mou et al. [13]–[16] proposed a Tree-Based Convolutional Neural Network (TBCNN) to transform the AST into a distributed vector that preserves semantic information. Bui et al. [12] utilized double TBCNNs to achieve source code translation between different program languages.

The popularity of Graph Neural Networks (GNN) [19], [21], [22] inspired researchers to applying GNN to source code mining [18], [19]. The core idea is to represent the program as a simple graph and then feed the graph as an input into GNN to obtain the distributed node representation for each node in the graph. Finally, the distributed representation is used to detect error variables or predict variable names.

The most related work to our work is the work of Alamanis [18], which applied GGN to the program graph integrating AST and data-flow edges, in order to predict variable names. However, this work did not take into accounts the function-call relations embedded in the programs, the dynamic distributed vector representations for edges in the program graph, the influence of different adjacent nodes through attention mechanism, and the aggregation of global information through attention mechanism. To the best of our knowledge, our work is the first work that applies the GNN-based model to program classification task.

### III. PRELIMINARIES

This section mainly introduces some basic concepts used in this paper, including graph, graph neural network, gated graph neural network.

#### A. The Graph Model

The graph model used in this paper is a directed graph $G = (V, E)$, among which $V$ represents the node set of size $|V|$ and depicts the directed edge set of size $|E|$. A node in $V$ is identified by a number and a directed edge in $E$ is expressed by $e_{ij}$, representing a directed edge from node $i$ to node $j$.
Since node $y$ represents the source node and the target node, respectively.

Time step, so as to predict the node attributes or the attributes hidden states and its own state information at the previous dates its hidden state information by aggregating its neighbors’ state information and gathers them as $m_i(t)$, as shown in Formula (1), where $N_i$ represents node i neighbor set. Then, the aggregated neighbor information is combined with node i ‘s previous state information $h_i(t-1)$ to form the new state information $h_i(t)$ through a neural network, called $nn$, as shown in Formula (2). It is worthy to note that the parameters of the propagation matrix $A_{ij}$ in Formula (1) are learned and shared for the same type of edges.

$$m_i(t) = \sum_{j \in N_i} A_{ij} \cdot h_j(t-1) \quad (1)$$

$$h_i(t) = nn(m_i(t), h_i(t-1)) \quad (2)$$

### C. Gated Graph Neural Network

Gated Graph Neural Network (GGNN) is actually an extension of GNN by replacing function $f$ in Formula (2) with the GRU function shown in Formula (3). The GRU function enables each node to memorize history information so that the temporal sequential relations among $h_i(t)$, $h_i(t-1)$ and $m_i(t)$ can be taken into account.

$$h_i(t) = GRU(h_i(t-1), m_i(t)) \quad (3)$$

### IV. THE GGANN ARCHITECTURE

This section mainly presents the architecture of the proposed work, as shown in Figure 4 which consists of two main components: the construction of the program graph and the application of the proposed GGANN model to the program graph. The first component converts the source code of a program to a program graph, while the second one learns the distributed representation of the program graph and utilizes the learned representation to classify the program. In the following, we will present each component individually.

#### A. The Program Graph Construction

As shown in Figure 4, the program graph construction itself actually consists of two main steps: 1) the construction of the Abstract Syntax Tree (AST), the Function Call Graph (FCG), and the Data Flow Graph (DFG) individually by parsing the source code; 2) the construction of the program graph by integrating the AST, FCG, and DFG. In the following, we will present the four types of graphs individually.

1. **Abstract Syntax Tree (AST):** AST [25] is an intermediate representation of a program generated through the program syntax and semantic analysis. It uses context-free grammar parsing rules[4] to construct a tree structure that characterizes the grammar structure and execution order of a program. Thus, an AST can equivalently represent the syntax structure of a program. For each node in this tree, its syntax and semantic analysis. It uses context-free grammar representation of a program generated through the program parsing, while 1 means complete information propagation.

2. **Function Call Graph (FCG):** The FCG is a directed graph that represents the function call structure of a program. Each node in this graph represents a function call, and the edges represent the function call relationships between functions.

3. **Data Flow Graph (DFG):** The DFG is a directed graph that represents the data flow relationships in a program. It consists of data nodes and control nodes, and the edges represent the data flow relationships between these nodes.

4. **Ast**: An AST can equivalently represent the syntax structure of a program. For each node in this tree, its syntax and semantic analysis. It uses context-free grammar representation of a program generated through the program parsing, while 1 means complete information propagation.
program. Each node within the AST represents an element in
the program. To illustrate AST, we use the program segment
shown in Figure 1(a) to generate the corresponding AST as
shown in Figure 5, where all the elements in the program
shown in Figure 1(a) are represented as nodes in the AST.
To illustrate the correspondence between the program and the
AST, we attach the variable/function/operator/literal elements
in the program to the corresponding nodes in the AST. For
example, in Figure 5, the FunctionDecl node is attached with
‘[add]’, which denotes that the node represents the add func-
tion declaration. Therefore, the subtree with the FunctionDecl
node as the root node represents the body of the add function.
Similarly, the VarDecl node represents a variable declaration,
and the IntegerLiteral node indicates that an integer literal is
declared.

As the parse tree of a program, an AST mainly covers the
following basic structures: selection structure (such as IF and
SWITCH), loop structure (such as WHILE and FOR), sequen-
tial structure (such as expression and assignment). Therefore,
AST, as an intermediate representation of a program, effec-
tively preserves the syntax of the associated programming
language and the context of the program runtime.

2) Function Call Graph (FCG): FCG [26] models the
function semantic information in a program. Each node in
FCG represents a function, while each edge in FCG denotes
the function-call relation between two functions. To illustrate
FCG, from the code fragment in Figure 1(a) and Figure 2,
we generate the corresponding FCG, as shown in Figure 6,
where the name of a node corresponds to the name of the
function. Similarly, the existence of an edge is determined by
the function-call relation between two function nodes.

3) Data Flow Graph (DFG): To capture the data flow
and control flow information associated with the program
runtime, we use DFG [27] to characterize this unique semantic
information. A node in DFG can be an entity, such as a
variable, a operator, a structure identifier, and etc, while an
edge in DFG represents the data transfer between two entities.
DFG can be used to characterize the logic and functionality
of a program.

To illustrate DFG, based on the code shown in Figure 1
we generate the corresponding DFG, as shown in Figure 7
where a superscript associated with a variable node denotes the
status of the corresponding variable. For example, in Figure 7
variable $y$ has 3 status, namely, $y_1$, $y_2$, and $y_3$, respectively.
The reason to introduce the superscript to describe various
status lies in that the value stored in a variable may change
during the process of program execution. Note that there
exist numerous situations, where a variable may be modified
or modify a variable value depend on a particular variable
status. For example, variable $y$ in the program fragment shown
in Figure 1(a) has three status, which correspond to two
assignment statements, and one return operation.

To characterize the above relations, we adopt 5 types of
DFG edges, namely, “LastUse”, “Compute”, “Formal”, “Re-
turn”, and “Operand”, which are illustrated through 5 different
colors in Figure 7. A “LastUse” edge characterizes the relation
between two different status of the same variable. For example,
consider the program shown in Figure 1(a) where the dark
green dotted edges denote the “LastUse” edges. The edge from
$x^2$ to $x^3$ is an “LastUse” edge because the value of variable
$x$ in statement “$y = x + 3$” directly depends on the value of $x$
in statement “int $x = FOO(m)$”.

A “Compute” edge represents the relation, where a variable
is computed from the other variable or a function, as shown in
the purple dotted edges in Figure 7. For example, in statment
“int x1 = Foo(m^2)”, x1 is computed from function Foo and
variable m^2. A “Return” edge represents the relation from
the function return value to the function name, which can
be illustrated through the red dotted edge from y^3 to add
in Figure 7. An “Operand” edge denotes the relation from
the operator to the operand in an expression, which can be
illustrated through the light green dotted edge from + to y^2
in Figure 7.

A “Formal” edge describes the relation from the formal
argument of a caller function to the formal argument of a
called function, or the relation from the actual argument in a
called function to the formal argument in the caller function,
which can be illustrated through the orange dotted edges in
Figure 7. Note that the “Formal” edge can also describe the
argument transfer among functions. For example, in Figure 7,
node m^1 represents the formal parameter m when function
‘Foo’ is declared, and m^1 denotes the actual parameter passed
to ‘Foo’ when it is called by function ‘add’. If the called
function has multiple arguments, it is necessary to connect the
actual parameters and the formal parameters through multiple
“Formal” edges correspondingly.

4) The Integrated FDA Graph: Through comparing AST,
FCG, and DFG, it can be observed that each of them can
only represents partial information of a program. The AST
basically contains the grammatical structure information of
a program and a small amount of semantic information,
while an executable program has richer semantic information
at runtime. FCG and DFG describe the semantic runtime
information of a program from the perspective of code blocks
(such as, functions, methods, and etc.) and code identifiers
(such as, variables, operators, and etc.), respectively.

Therefore, it is desirable to integrate these three code
representations into a new program graph. Since the AST
contains both function nodes and variable/operator nodes, it
is convenient to add a FCG edge or a DFG edge to the AST
if there exists a function-call relation or data-flow relation
between two AST nodes. Through inserting the FCG or DFG
edges accordingly, the integrated program graph can not only
represent the syntax information but also the function-call
and data-flow semantic information. We call the integrated
program graph as the FDA graph, which embeds not only the
basic grammatical structure information, but also the runtime
information. Therefore, the FDA graph can effectively reduce
the information loss due to the conversion of a program to an
intermediate representation.

To illustrate the integration of AST, FCG, and DFG, we
added the FCG edges and the DFG edges to the AST in
Figure 6. The integrated FDA graph is shown in Figure 8,
where the yellow dotted edge represents the FCG edge and
the colors of the other dotted edge denotes the same types of
DFG edges shown in Figure 7.

The integrated FDA graph has 7 edge types in total, in-
cluding the original edge in the AST, the 5 types of edges
introduced by the DFG, and the function-call edge. The
enriched FDA graph can speed up the information propagation
on the proposed GGANN model, which will be presented as
follows.

B. The GGANN model

In GGNN, the node connection and information propagation
rule are defined by the connection matrix A, where the
propagation matrix are the same for the same type of edges and
also will not change over different period of time. However, the same information propagation path may have different semantic information for different contexts. For example, consider a conditional selection code block in a program, where there exists program fragment right after both the “if” and “else” branches. The execution of any of these two branches represent different contexts for the followed program fragment. Therefore, it is necessary to dynamically learn the parameters of the propagation matrix according to the context and state information. For this reason, we first introduce d-dimensional vector $h_{ij}^{(t)}$ to represent the hidden state of the directed edge $e_{ij}$, which reflects the dynamic variation of the edge state. $h_{ij}$ can be implemented by a multi-layer neural network $U_e$, the input of which includes both node $i$’s and node $j$’s hidden states, as well as the previous edge state information, as shown in Formula (4).

$$h_{ij}^{(t)} = U_e(h_{ij}^{(t-1)}, h_i^{(t-1)}, h_j^{(t-1)})$$

Based on this, we can replace the propagation matrix $A_{ij}$ in Formula (1) with a propagation matrix function, $A(h_{ij}^{(t)}) : R^d \rightarrow R^d \times R^d$, which learns the propagation matrix dynamically according to the dynamic information of the hidden state $h_{ij}^{(t)}$. $A(h_{ij}^{(t)})$ can also be realized by a multi-layer neural network, which, at time $t$, maps the hidden state $h_{ij}^{(t)}$ to the propagation matrix.

In addition, for each node, its neighbor nodes may have different importance in terms of the contribution to update the node hidden state. However, as shown in Formula (1), both GNN and GGNN simply sum the neighbors’ state information. For this reason, we first introduce d-dimensional vectors $\alpha_{ij}$, which compare the similarity between node $i$ and its neighbor node $j$, and the similarity is normalized through the activation function, softmax, as shown in Formula (5):

$$\alpha_{ij} = \text{softmax}(a(h_{ij}^{(t-1)}, h_j^{(t-1)}))$$

The parameters of the neural network $a$ are shared for the nodes with the same type and the same direction.

In a summary, the node information aggregation adopted in GNN and GGNN, as shown in Formula (1), can be replaced by the Formula (6) in GGANN.

$$m_i^{(t)} = \sum_{j \in N_i} \alpha_{ij}A(h_{ij}^{(t)})h_j^{(t-1)}$$

Finally, because the program classification task is to classify a program through the associated FDA graph, the vector representation of the FDA graph should be obtained after the calculation of each node’s embedding vector, i.e., its hidden state. Since different nodes in the FDA graph may contribute differently to classify the program, we adopt the soft attention mechanism to compute the weight of each node to the FDA graph. The soft attention mechanism involves the similarity comparison between each node and the entire graph.

However, the vector representation for the FDA graph is unknown before the calculation of a node’s contribution. To address this, we use the updated hidden state of each node after $T$ iterations of information propagation to approximate the vector representation of the entire FDA graph. The underlying reason lies in that each node contains the information of its T-hop neighbor nodes after $T$ iterations, which can well approximate the whole graph with sufficient large $T$.

Besides that, each node’s initial information usually uses the node’s input feature vector. Therefore, the weight of each node in the FDA graph can be replaced by $f(h_i^{(T)}, x_i)$, where $f$ represents a neural network to implement soft attention mechanism, and the output of $f$ is the similarity (a scalar) between the node and the entire FDA graph. Based on this, the weighted embedding vector of a node can be calculated by multiplying each dimension of the node embedding vector $g(h_i^{(T)})$ with the weight of the node.

Thus, the embedding vector of the entire FDA graph can be computed by summing the weighted embedding vector of each node. Once the embedding vector of the FDA graph is obtained, the category of the program associated with the FDA graph can be predicted by applying the softmax activation function to the embedding vector, as shown in Formula (7).

$$h_G = \text{softmax}(\sum_{i \in V} f(h_i^{(T)}, x_i) \odot g(h_i^{(T)}))$$

V. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of the proposed GGANN model and the FDA graph, we not only evaluate the model from the perspective of program classification accuracy, but also analyze what the proposed model learns from the perspective of the semantic information implied by the learned embedding vectors. To understand the learning effect of the model intuitively, some experimental results are presented visually.

A. The Dataset Description

The experimental data are collected from an Online Judge (OJ) programming exercise system, where administrators issue numerous programming tasks, to each of which students submit their coded solutions in the form of source-code files. To evaluate the correctness of the submitted codes, the OJ system will compile and execute the codes and compares the execution results with the standard output. Therefore, the data set is mainly composed of two parts: the programming tasks (represented by question number) and the corresponding source codes submitted by students.

To exclude the influence of programming languages, we consider only the source codes written in C++. Meanwhile, to ensure that there are sufficient source codes for each programming task, we filter the data so that only the programming tasks with more than 350 submitted codes will be considered for program classification. In total, 30 programming tasks meet this filtering requirement. Since the selected programming tasks are more common than the filtered tasks, we call them the common data subset.

Among these common tasks, there exist a few tasks, which are similar in terms of task description and coding implementation. These similar tasks can be utilized to evaluate
the robustness and effectiveness the proposed GGANN model and FDA graph. Thus, 10 programming tasks with similar descriptions are selected from the common data sets, and divided into two groups according to their similarity (5 tasks in each group).

We use clang, an open-sourced dependency library, to parse each C++ program into an AST, which is further processed to generate the FDA graph by adding data-flow call edges. The generate FDA data set is divided into the training set, the validation set and the test set with the ratio being 3:1:1. Table 1 lists the statistical information of the training, validation and test sets, where the common task and the similar task datasets are represented by Common and Similarity, respectively.

|          | Graphs | Nodes | Edges | Classes | Edge Types |
|----------|--------|-------|-------|---------|------------|
| Train    | Common | 35643 | 8456506 | 10895549 | 30 | 7 |
|          | Similarity | 7123 | 1408234 | 1884146 | 10 | 7 |
| Valid    | Common | 11297 | 2838685 | 3994853 | 30 | 7 |
|          | Similarity | 2411 | 461078 | 649538 | 10 | 7 |
| Test     | Common | 11366 | 3061059 | 4259252 | 30 | 7 |
|          | Similarity | 2583 | 458344 | 650955 | 10 | 7 |

Table I: Data set statistics

B. The Experiment Set-up

To train the model, the optimization algorithm adopted the ADAM optimizer’s Stochastic Gradient Descent (SGD) algorithm \(28\), the loss function is to minimize the cross entropy, the batch size is set as 10,000, and the number of epoch is set as 3000. The model parameters are initialized with the Glorot initialization method (set to 0.0001) \(29\). To avoid overfitting, we adopted the \(L_2\) regulation term with the initial value \(\lambda = 0.0005\), and applied dropout \(30\) to the input of each layer with probability \(\rho = 0.6\). We also used a linear learning-rate decay to adjust the learning rate from the initial learning rate \(\ell\) to the final learning rate \(\ell \ast F\), with the attenuation coefficient \(F\) in the range \([0, 1]\).

In our experiment, the GGANN network has 5 hidden layers representing 5 iterations of information propagation. The validation set is used to choose the model hyper parameters and determine the opportunity for early stopping, while the test set is used to calculate the classification accuracy. For example, the number of hidden layer neurons, i.e., the dimension of node features \(d\), used in the experiment, is a hyper parameter. \(d\) can be determined by the following two factors: (1) the convergence speed of the model, (2) the loss value. To determine the appropriate value of \(d\), several experiments have been conducted. The experimental results are shown in Figures 9(a) and 9(b). Figure 9(a) shows the variation of the training/testing speed in terms of the number of graphs processed per second along with the change of the \(d\) value, while Figure 9(b) shows the variation of the loss value along with the change of \(d\) value for both training and test sets. As can be seen from both Figures 9(a) and 9(b) when \(d\) is set to 270, the loss value of the model is relatively small, and the training speed is relatively fast. Therefore, in the subsequent experiments, we set the hidden layer vector dimension \(d = 270\).

C. The Experiment Results

This subsection mainly analyzes the program classification performance associated with the proposed GGANN model and the FDA graph, as well as what the embedding vectors learned, through experimental evaluation. We first analyze the program classification performance.

1) Program Classifying: The experiments for program classification are used to verify whether the proposed work can successfully learn grammatical structure and semantic information from source codes, i.e., whether the submitted codes implemented by different students for the same programming task can be classified into the same category, while the codes for different programming tasks with similar grammatical structure can be effectively distinguished.

In the experiments, we consider binary classification, which determines whether a given program belongs to the associated programming task or not. For each program, the corresponding programming task number is used as its label. The baseline is the Tree Based Convolution Neural Network (TBCNN), which, as far as we know, is the state-of-art work for program classification. In addition, since the proposed GGANN model is an extension of the GGNN model, we also adopt GGNN as the baseline model. It is worthy to note that TBCNN cannot apply to the FDA graph, because it is designed for AST only.

Table II shows the experiment results in terms of the average, minimum, and maximum program classification accuracy for the GGANN, GGNN and TBCNN models on the similar datasets. The experimental results show that the classification accuracy of the TBCNN model is significantly lower than those of the GGANN and GGNN models, and the GGANN model is also significantly improved compared with the GGNN model. From the experiment results, the following observation can be concluded. On one hand, TBCNN relies too much on AST, which makes it difficult for TBCNN to distinguish similar tasks with close static structures but different runtime semantic information. On the other hand, the edge embedding representation and the attention mechanism adopted by the GGANN model effectively improve the GGNN model.

Table II: The program classification accuracy of different models on the similar programming tasks.

|          | Accuracy | Average | Minimum | Maximum |
|----------|----------|---------|---------|---------|
| Similarity(GGANN) | 93.9% | 88.7% | 97.7% |
| Similarity(GGNN) | 90.3% | 84.2% | 94.8% |
| Similarity(TBCNN) | 85.7% | 83.2% | 87.9% |

To further clarify the origin of the performance improvement, i.e., whether the improvement comes from the model advantage of GGANN over TBCNN and GGNN or the information advantages of FDA over AST, we further compare the performance of the three models to AST and the performance of GGANN and GGNN to FDA for the common task data set. The experimental results are shown in Table III. Compared with the results in Table II it can be concluded that, for all
the three models, the classification accuracy on the common tasks is better than those on the similar tasks.

Furthermore, it can be inferred that, in contrast to GGNN and GGANN, the performance improvement of TBCNN for similar tasks over common tasks is about 9%, which is significantly higher than those of GGNN and GGANN. This indirectly illustrates that the graph network models (GGNN and GGANN) are more robust, because the performance of the graph models reduce much less than TBCNN when it deals with similar tasks.

Moreover, from Table III by observing the accuracy of the three models on AST, it can be inferred that the graph network model alone can significantly improve the classification accuracy. This can illustrate the contribution of the graph models.

TABLE III: The accuracy of the three models associate with AST and FDA for the common data set

|         | TBCNN | GGNN | GGANN |
|---------|-------|------|-------|
| Average | 0.941 | 0.964| 0.971 |
| AST     |       |      |       |
| Minimum | 0.926 | 0.935| 0.946 |
| Maximum | 0.952 | 0.976| 0.981 |
| Accuracy|       |      |       |
| FDA     |       |      |       |
| Minimum | -     | 0.943| 0.951 |
| Maximum | -     | 0.978| 0.983 |

However, the experimental results shown in Table III also imply that the performance improvement of the graph models on the FDA graph over AST is not significant. In order to further analyze this phenomenon, we compare the classification accuracy of the three models on each programming task of the common and similar task sets, respectively, as shown in Figure 10.

Figure 10(a) and Figure 10(b) visualize the classification accuracy of the three models associated with each similar programming task on the validation set and the test set, respectively. From the experimental results, the following can be observed: (1) the classification accuracy of the GGANN model is slightly higher than that of the GGNN model in almost every programming task; (2) the accuracy variance of GGANN in different tasks is relatively small; (3) both classification accuracies of the two graph models are higher than that of the TBCNN model.

Figure 10(c) and 10(d) visualize the classification accuracy from the application of the three models on the two program intermediate representations (FDA or AST) associated with each common programming task on the validation set and test set, respectively. Besides the application of GGANN on FDA (the proposed work) and the application of TBCNN on AST (the state-of-art method), we also consider the application of GGNN on AST and FDA. The application of GGNN on AST can be used to compare the state-of-art method, so as to identify the advantage of GGNN over TBCNN. The application of GGNN on FDA can be used to compare with the application of GGNN on AST and the proposed work, in order to identify the advantage of FDA over AST and the advantage of GGANN over GGNN, respectively.

From the experimental results shown in Figure 10(c) and 10(d), it can be observed that, although the application of the two graph models on FDA perform better than the application of TBCNN on AST, the contribution of the FDA is not significant, because the application of the GGNN model on FDA is just slightly better than that on AST in terms of classification accuracy.

To evaluate whether the data-flow and function-call edges added to the AST is indeed useful, we remove each of the seven edge types from the FDA graph and apply the GGANN model to the seven modified FDA graphs, respectively, so as to observe the contribution of each edge type to the program classification accuracy.

The experiment results are visualized in Figure 11 where \textit{\textit{overline{edgeType}}} denotes the modified FDA graph with a given edge type removed. It can be inferred from the experiment results that no edge type contributes significantly more than other edge types except for a few program tasks. Thus, it can be concluded that there exist information redundancy among the seven edge types in most cases, so that the removal of any edge type will not have significant impact on the program classification accuracy.

However, for a few program tasks, certain edge types con-
Fig. 10: The classification accuracy of similar/common programming tasks for the combination of models and program intermediate representations.

In addition to classification accuracy, we also evaluate the impact of the proposed GGANN model and FDA through the loss values. Figures 12 show the experimental results, which compare the convergence trend of the loss values associated with the applications of GGANN/GGNN/TBCNN on FDA/AST, respectively, along with the number of iteration epochs.

From the experiment results, it can be observed that not only the final loss values of the two graph models are smaller, but also they converge faster than TBCNN, even if only the AST edges are used. The reason that graph network model converges faster might lie in that the graph models have stronger constraints on the AST nodes than TBCNN. More specifically, in TBCNN, the convolution operation enforces a one-way information propagation from child nodes to parent nodes, while in the graph models, for each node, each iteration involves two-way information propagations among all neighbor nodes, which will gradually spread to the whole network. This strong constraint on node relations also enables the stronger learning capability of the graph models than that of TBCNN, as illustrated in Figure 12 where the loss values of GGNN and GGANN almost converge at the beginning. Finally, from Figure 12 it can also be concluded that GGANN is better than GGNN in terms of convergence speed of loss values.
TABLE IV: The program tasks that can be impacted by certain edge types

|             | FDA | AST | LastUse | Compute | Return | Formal | Call |
|-------------|-----|-----|---------|---------|--------|--------|------|
| Task13      | 0.981 | 0.977 | 0.939 | 0.061 | 0.056 | 0.939 | 0.943 | 0.979 |
| Task14      | 0.984 | 0.990 | 0.979 | 0.979 | 0.979 | 0.979 | 0.979 | 0.979 |

Fig. 11: The contribution comparison among different edge types

Fig. 12: The variation of loss values for the four models along with the iterations.

2) The Semantic Analysis of the Learned Representation:
Based on the characteristics of GGANN model, the feature representation of each node is computed by learning its local structure, i.e., its neighbor nodes and the edges among them. Thus, the nodes with similar local structure should have similar representation. In order to evaluate whether the GGANN model can learn this structure similarity, we use K-mean, an unsupervised clustering method, to cluster the learned node feature vectors, and visualize the clustering results in two-dimensional space through t-SNE. The experimental results are shown in Figure 13, where it can be observed that the GGANN model does learn the semantic characteristics of nodes. For example, ForStmt, ContinueStmt, GotoStmt, BreakStmt, and etc are clustered into cluster 3, because most of them are related to control flow. Similarly, UnaryOperator, BinaryOperator, StringLiteral, IntegerLiteral, and etc, are clustered to cluster 1, because these operators are basically related to data reference/manipulation, while the statement _= and -=, as well as CompoundAssignOperator, are clustered to cluster 2, because most of them are related to composite operations.

Fig. 13: A t-SNE plot of the learned node representations, where different node colors denote different clusters.

3) Attention Analysis: GGANN defines a graph-level aggregation function for graph-level output, as shown in Formula (7). Each node has a weight (i.e., attention) calculated through attention mechanism, which reflects the similarity between the node and the whole FDA graph. In order to verify whether the node attention learned by GGANN indeed reflects the node importance, we present the attention of all nodes in an FDA graph in the form of thermodynamic chart, as shown in Figure 14, which has 104 rectangular bars, each of which corresponds to a node of the FDA graph. The lighter the color, the higher the node weight. In Figure 14, the nodes with higher weights are BinaryOperator, CallExpr, ImplicitCastExpr, IfStmt, DeclStmt, and CompoundStmt, with weights being 0.71, 0.74, 0.75, 0.75, 0.72 and 0.74, respectively. These nodes are basically located at the higher levels of AST, which are generally more abstract than the nodes at lower levels and contain more information about the whole program. The experiment results reflect that the GGANN model indeed learned the node importance and paid more attention to the important nodes.
VI. CONCLUSION AND FUTURE WORK

To address the problem of the limited social interactions among the users of the online programming service, we propose to use source-code mining technique to promote social interactions. As the first attempt to achieve the goal, in this work, we propose a graph-based program classification model, which can classify a give program with high accuracy by understanding the structural and semantic information of the program. More specifically, in this work, the FDA graph is proposed to convert a program’s source codes to an intermediate representation to be fed into the graph network models, and a graph network model GGANN is proposed for the program classification application. As far as we know, this is the first work that applies the graph network models to the program classification application. The experimental evaluation illustrates that the proposed work successfully learns the grammatical structure and semantic information of various programming tasks, including the tasks with similar descriptions, and can generalize well to the test set, comparing with the state-of-art program classification work, TBCNN. In the future, we will further mine the correlation implicated in the source code through the integration of the compiling techniques, accelerate the training speed of the GGANN model through the sampling and pooling techniques, and extend the GGANN model to other programming languages and other tasks.

TABLE V: The partial results of Kmeans clustering ($K = 5$).

| Cluster Symbols | 1 | 2 | 3 | 4 | 5 |
|-----------------|---|---|---|---|---|
| varStmt, StringLiteral, UnaryOperator, BinaryOperator, IntegerLiteral | 0.5 | 0.6 | 0.8 | 0.7 | 0.5 |
| DoStmt, CompoundAssignOperator, >>|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|
[21] M. Defferrard, X. Bresson, and P. Vandergheynst, “Convolutional neural networks on graphs with fast localized spectral filtering,” in Advances in Neural Information Processing Systems, 2016, pp. 3844–3852.

[22] T. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in International Conference on Learning Representations, 2017.

[23] M. Gori, G. Monfardini, and F. Scarselli, “A new model for learning in graph domains,” in Proceedings of 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 2, 2005, pp. 729–734.

[24] S. Franco, G. Marco, T. A. Chung, H. Markus, and M. Gabriele, “The graph neural network model,” IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61–80, 2009.

[25] F. Miller, A. Vandome, and J. Mcbrewster, Abstract syntax tree. Alphascript Publishing, 2010.

[26] X. Liu and Y. Tang, “Similarity analysis of malware’s function-call graphs,” Computer Engineering & Science, 2014.

[27] B. M. Olenick, C. A. Szyperski, D. G. Hunt, W. A. Manis, and T. Zmrhal, “Accessing and manipulating data in a data flow graph,” 2010.

[28] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” Computer Science, 2014.

[29] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in Proceedings of the the Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, 2014.

Dr. Xiong published over 300 international journal papers and over 100 international conference papers. Some of his works were published in IEEE JSAC, IEEE or ACM transactions, ACM Sigcomm workshop, IEEE INFOCOM, ICDCS, and IPDPS. He has been a General Chair, Program Chair, Publicity Chair, PC member and OC member of over 100 international conferences, and as a reviewer of about 100 international journals, including IEEE JSAC, IEEE SMC (Park: A/B/C), IEEE Transactions on Communications, IEEE Transactions on Mobile Computing, IEEE Trans. on Parallel and Distributed Systems. He is serving as an Editor-in-Chief, Associate editor or Editor member for over 10 international journals (including Associate Editor for IEEE Tran. on Systems, Man & Cybernetics: Systems, Associate Editor for Information Science, Editor-in-Chief for Journal of Internet Technology (JIT), and Editor-in-Chief for Journal of Parallel & Cloud Computing (PCC)), and a guest editor for over 10 international journals, including Sensor Journal, WINET and MONET. He has received the Best Paper Award in the 10th IEEE International Conference on High Performance Computing and Communications (HPCC-08) and the Best student Paper Award in the 28th North American Fuzzy Information Processing Society Annual Conference (NAFIPS2009).

Dr. Xiong is the Chair of Trusted Cloud Computing Task Force, IEEE Computational Intelligence Society (CIS), http://www.cs.gsu.edu/~cscnxx/index-TF.html, and the Industry System Applications Technical Committee, http://ieee-cis.org/technical/isatc/; He is a Senior member of IEEE Computer Society, E-mail: xiongnairxue@gmail.com.