Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection

Ting Long [1*], **Yutong Xie** [2*], **Xianyu Chen** [1], **Weinan Zhang** [1†] **Qinxiang Cao** [1], **Yong Yu** [1†]

[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2]School of Information, University of Michigan
{longting,xianyujun,wnzhang,caoqinxiang}@sjtu.edu.cn, yutxie@umich.edu, yyu@apex.sjtu.edu.cn

Abstract

Program representation, which aims at automatically extracting features from source code and representing programs as vectors, is a fundamental problem in programming language processing (PLP). Recent works try to represent programs with neural networks based on the structures of source code. However, such methods often focus on the syntax and consider only one single perspective of programs, limiting the representation power of models. In this paper, we propose a multi-view graph (MVG) representation method. MVG pays more attention to the semantics of code and include both data flow and control flow simultaneously as multiple views. We combine these views together and process them with a graph neural network (GNN) to obtain a program representation that covers various aspects. Our proposed MVG approach is thoroughly evaluated in terms of algorithm detection, an important and challenging subfiled of PLP. Specifically, we use a public dataset `POJ-104` and also construct a new dataset `ALG-109` to test our method. In experiments, MVG outperforms previous methods significantly, demonstrating our model's strong capability for representing source code.

Introduction

With the advent of *big code* (Allamanis et al. 2018), programming language processing (PLP) gains plenty of attention in recent years. PLP aims at assisting computers automatically understanding and analyzing source code, which benefits downstream tasks in software engineering like code retrieval (Lv et al. 2015; Nie et al. 2016), code annotation (Yao, Peddamail, and Sun 2019), bug predicting and fixing (Xia et al. 2018; Wang, Singh, and Su 2017), program translation (Chen, Liu, and Song 2018; Gu et al. 2017). To take advantage of deep learning, the program representation problem, *i.e.*, how to convert source code into representational vectors, becomes a critical issue in PLP.

A great deal of literature devotes its efforts to the problem of program representation. Among these works, a majority of them represent source code only based on syntactic information like abstract syntax trees (ASTs) (Mou et al. 2016; Alon et al. 2019), while ignoring the semantics of programs. Thus, some researchers propose to include semantic information by adding semantic edges onto ASTs (Allamanis, Brockschmidt, and Khademi 2018; Zhou et al. 2019). However, the program representation still highly depends on the syntax and the semantics are relatively underweighted. Moreover, in previous methods, information from different aspects like syntax, data flow, and control flow are often mixed up together into one single view, making the key information hard to be separately extracted out.

Therefore in this paper, to address the problem mentioned above, we propose to use a multi-view graph (MVG) representation for source code. To understand programs more comprehensively, we consider multiple graph views from various aspects and levels. In particular, we emphasize more on the semantics and include the following views in MVG: the data-flow graph (DFG), control-flow graph (CFG), read-write graph (RWG), and a combined graph (CG). DFG and CFG are widely used in compiling and traditional program analysis. We construct RWG based on DFG and CFG to capture the relationship between operations and operands. We also include CG, which is a combination of the former-mentioned graphs, to have an integral representation of the program. We then apply a gated graph neural network (GGNN) (Li et al. 2016) to automatically extract key information from the four graph views.

We validate our proposed MVG method mainly on the algorithm detection task, which is an important subfield of PLP, aims at identifying the algorithms and data structures that appear in the source code. The first reason for which we choose this subfield is because of its wide application range: the detection results can be used as intermediate information for further program analysis; we might also apply algorithm detection in areas like programming education, *e.g.*, determining which algorithms are mastered by the students. In addition to the wide range of application, algorithm detection is also very challenging and can serve as a benchmark for PLP program representation. This is because: (1) A piece of code can contain multiple different algorithms and data structures; (2) One algorithm or data structure can have multiple possible implementations (*e.g.*, `Dynamic Programming`, `Segment Tree`); (3) Different algorithms can have very similar implementations (*e.g.*, `Dijkstra's Algorithm` and `Prim's Algorithm`). Under this algorithm detection task, we use two datasets to test our MVG model. The first

*These authors contributed equally.
†Corresponding author

one is a public dataset `POJ-104` (Mou et al. 2016). We also create a new dataset `ALG-109`, which is more challenging than the former one. On both two datasets, our MVG model outperforms previous methods significantly, demonstrating the outstanding representation power of MVG.

In summary, our contributions are as follows:

- We propose the MVG model that can understand source code from various aspects and levels. Specifically, MVG includes four views in total: the data-flow graph (DFG), control-flow graph (CFG), read-write graph (RWG), and a combined graph (CG);
- We create an algorithm classification dataset `ALG-109` to serve as a program representation benchmark;
- We validate MVG on the challenging algorithm detection task with a public dataset `POJ-104` and our constructed dataset `ALG-109`. In experiments, MVG achieves state-of-the-art performance, illustrating the effectiveness of our approach.

Related Work

Previous program representation methods could be divided into four categories: *data-based*, *sequence-based*, *tree-based*, and *graph-based*.

Data-based methods assume programs are functions that map inputs to outputs. Therefore, the input and output data could be used to represent the program. Piech et al. (2015) embed inputs and outputs of programs into a vector space and used the embedded vectors to obtain program representations; Wang (2019) collects all the data during program execution and feeds the data to a long short-term memory (LSTM) unit (Hochreiter and Schmidhuber 1997) to obtain program representations. Though seemingly intuitive, data-based methods can often be limited by the availability of the input or output data, and it might take forever to enumerate all possible inputs.

Sequence-based methods assume that programming language is similar to natural language, and adjacent units in code (*e.g.*, tokens, instructions, or command lines) will have a strong correlation. Hence, these methods apply models in natural language processing (NLP) to source code. For examples, Harer et al. (2018), Ben-Nun, Jakobovits, and Hoefler (2018), and Zuo et al. (2018) apply the word2vec model (Le and Mikolov 2014) to learn the embeddings of program tokens. Feng et al. (2020),Wang et al. (2020) and Ciniselli et al. (2021) use a pre-trained BERT model to encode programs. Such sequence-based methods are easy to use and can benefit a lot from the NLP community. However, since source code is highly structured, simple sequential modeling can result in a great deal of information loss.

Tree-based methods are mostly based on the abstract syntax tree (AST), which is often used in compiling. Different from the sequential modeling of programming language, AST contains more structural information of source code. In the previous work, Mou et al. (2016) parse programs into ASTs and then obtain program representations by applying a tree-based convolutional neural network on the ASTs; Alon et al. (2019) obtain program representations by aggregating paths on the AST. Tree-based representations usually contain more structural information than sequences, but the semantic information of the program might be relatively ignored compared with the syntactic information.

Graph-based methods parse programs into graphs. Most approaches from this category construct program graphs by adding edges onto ASTs. For instance, Allamanis, Brockschmidt, and Khademi (2018) introduce edges like `LastRead` and `LastWrite` into AST. Then the program representations are obtained with a gated graph neural network (GGNN) (Li et al. 2016). Zhou et al. (2019) extend the edges types in the work of Allamanis, Brockschmidt, and Khademi (2018) and further improve the performance. Although graph-based methods can have better performance than previously mentioned categories (Allamanis, Brockschmidt, and Khademi 2018), we notice that information from different perspectives usually crowds in one single view, *i.e.*, most methods use one single graph to include different information in the source code, which can limit the representation power of the model. Moreover, such approaches tend to build their graphs based on the AST and will give much attention to the syntactic information, suppressing the semantics of programs. By contrast, in this paper, we propose the MVG method which will consider multiple views of programs simultaneously. We extract features from data flows, control flows, and read-write flows, focusing more on the semantic elements.

Methodology

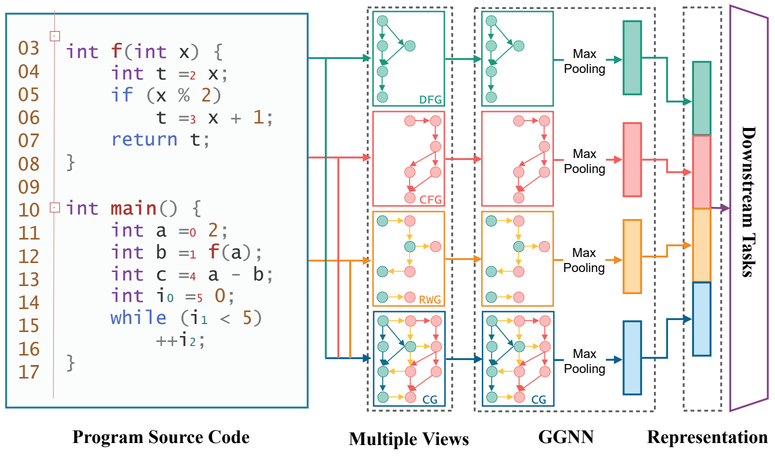
This section describes how the MVG method converts programs into representational vectors. In particular, as displayed in Figure 1, we first represent a piece of source code as graphs of multiple views. We process these graphs with a gated graph neural network (GGNN) to extract the key information that resides in graphs. The extracted information is then integrated to obtain a comprehensive representation.

Program Graphs of Multiple Views

To understand a program from different aspects and levels, we represent the program as graphs of multiple views. We consider four views in total: (1) data-flow graph (DFG); (2) control-flow graph (CFG); (3) read-write graph (RWG); and (4) combined graph (CG).

Data-flow graph (DFG) Data flow is widely used to depicts programs in traditional program analysis (Aho, Sethi, and Ullman 1986; Farrow, Kennedy, and Zucconi 1976; Fraser and Hanson 1995; Muchnick et al. 1997). We use DFG to capture the relationship between operands. In DFG, nodes are operands and edges indicate data flows. As it is presented in Figure 1(b), the DFG of the code in Figure 1(a) is the green. DFG includes two types of nodes, namely *non-temporary operands* and *temporary operands*. Non-temporary operands denote variables and constants that explicitly exist in the source code, and temporary operands stand for temporary variables that only exist in program execution. Two groups of edges are considered:

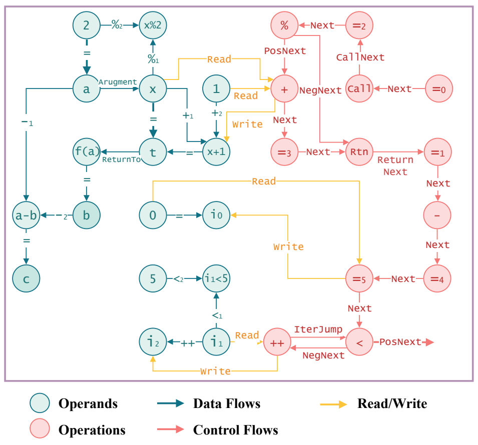
- *Operation edges* exist in non-function-switch code. They connect the nodes to be operated and the nodes that re-



```
int f(int x) {
    int t =2 x;
    if (x % 2)
        t =3 x + 1;
    return t;
}

int main() {
    int a =0 2;
    int b =1 f(a);
    int c =4 a - b;
    int i0 =5 0;
    while (i1 < 5)
        ++i2;
}
```

(a) The MVG pipeline.



(b) An example of the combined graph.

Figure 1: (a) The pipeline of MVG. Four graphs (*i.e.*, DFG, CFG, RWG, and CG) are constructed based on the given source code. These constructed graphs are then fed into a GGNN to obtain a final program presentation for downstream tasks. (b) An example of the combined graph (CG) corresponding to the program source code in (a).

ceive the operation results. Standard operations are included in this category, *e.g.*, $=$, $+$, $-$, $\star$, $/$, $>$, $<$, $==$. We distinguish different types of operations by using various types of edges.

- *Function edges* indicate data flows for function calls and returns, including two types of edges: `Argument` and `ReturnTo`. We use `Argument` edges in function calls to connect actual arguments and the corresponding formal arguments. We use `ReturnTo` edges to associate return values and the variables that receive the returns.

Control-flow graph (CFG) We utilize CFG to model the execution order of operations. As Figure 1(b) shows, the CFG of the code in Figure 1(a) is the red. Based on compilers principles (Aho, Sethi, and Ullman 1986; Allen 1970), we slightly adjust the design of CFG to better capture the key information of the program. Nodes in CFG are operations in the source code, including *standard operations*, *function calls* and *returns*. Edges indicate the execution order of operations. The following edge types are considered:

- *Condition edges* indicate conditional jumps in loops or branches (*e.g.*, `while`, `for`, `if`). We define `PosNext` and `NegNext` two subtypes to represent situations where the conditions are `True` or `False` respectively. These edges start from condition operations and end at the first operation in the `True` or `False` blocks.
- *Iteration edges* are denoted as `IterJump`. We use them in loops (*e.g.*, `while` and `for`) to indicate jumps at the end of each iteration, connecting the last and the first operations in the loop.
- *Function edges* are used in function calls and returns, including two subtypes `CallNext` and `ReturnNext`. `CallNext` edges start from function call operations and point to the first operations in the called functions. `ReturnNext` edges begin with the last operations in called functions and end at the operations right after the corresponding function calls.
- *Next edges* stand for the most common execution order except for the above cases. Denoted as `Next`, they connect operations and their successor in execution order.

Read-write graph (RWG) We design the RWG to capture the interaction between operands and operations. As Figure 1(b) shows, part of RWG for the code in Figure 1(a) is the yellow edges and the nodes connect to yellow edges. RWG is a bipartite graph with *operands* and *operations* as nodes. Two types of edges are introduced to connect operands and operations:

- *Read edges* start from operands and point to operations, meaning operations take operands to compute.
- *Write edges* start from operations and point to operands, meaning variables receive the operation results.

Combined graph (CG) In addition to DFG, CFG, and RWG, we further introduce a combined graph to capture the comprehensive overall information of a program. CG is an integral representation of the above three graphs and is obtained by first including all nodes and edges in DFG and CFG, and then adding `Read` and `Write` edges to connect variable and operation nodes as Figure 1(b) shows.

To summarize, formally, we can denote the graphs of each view as $\mathcal{G}_i = \{\mathcal{V}_i, \mathcal{E}_i\}$ where $i \in \{\text{DFG}, \text{CFG}, \text{RWG}, \text{CG}\}$, $\mathcal{V}_i$ is the node set and $\mathcal{E}_i$ is the edge set. We have $\mathcal{V}_{\text{RWG}} \subseteq \mathcal{V}_{\text{CG}} = \mathcal{V}_{\text{DFG}} \cup \mathcal{V}_{\text{CFG}}$, and $\mathcal{E}_{\text{CG}} = \mathcal{E}_{\text{DFG}} \cup \mathcal{E}_{\text{CFG}} \cup \mathcal{E}_{\text{RWG}}$.

Extracting Information with a GGNN

As mentioned above, a program can be represented as four views in the form of graphs. Here, we adopt a gated graph neural network (GGNN) (Li et al. 2016), a widely used graph neural network (GNN) model, to extract features from each graph view.

For a graph $\mathcal{G}_i$ of an arbitrary view, this GGNN first initializes nodes' hidden representations with one-hot encodings of node types (*i.e.*, operation or operand types). For any node $u \in \mathcal{V}_i$, we initialize its hidden state as below:

$$\mathbf{h}_u^0 = \mathbf{x}_u, \tag{1}$$

where $\mathbf{h}_u^0$ is the initial hidden state of u, and $\mathbf{x}_u$ is the one-hot encoding of u's node type.

The nodes then update their states by propagating messages in the graph as the following equations:

$$\mathbf{m}_{u,v}^t = f_e(\mathbf{h}_v^{t-1}), \quad e = (u, v) \in \mathcal{E}_i, \tag{2}$$

$$\bar{\mathbf{m}}_u^t = \text{Mean}(\{\mathbf{m}_{u,v}^t\}_{v \in \mathcal{N}(u)}), \quad u \in \mathcal{V}_i, \tag{3}$$

$$\mathbf{h}_u^t = \text{GRU}(\mathbf{h}_u^{t-1}, \bar{\mathbf{m}}_u^t,), \quad u \in \mathcal{V}_i, \tag{4}$$

where u, v are node indicators and e is an edge indicator, $\mathbf{m}_{u,v}^t$ stands for the message u receives from v at the t-th iteration, $f_e(\cdot)$ is a message passing function that depends on the edge type of e, $\mathbf{h}_v^{t-1}$ represents the hidden state of v from the last iteration, $\bar{\mathbf{m}}_u^t$ is the aggregated message received by u, Mean$(\cdot)$ denotes the average pooling function, $\mathcal{N}(u)$ is the set of u's neighbors, $\mathbf{h}_u^t$ is the updated hidden state, and GRU$(\cdot)$ is a gated recurrent unit (Cho et al. 2014).

After T iterations, the hidden states will contain enough information of the given graph. Therefore, we take the hidden states of nodes at the final iteration and integrate them using a max pooling to obtain a final vector representation of the graph view $\mathcal{G}_i$:

$$\mathbf{z}_i = \text{MaxPooling}(\{\mathbf{h}_u^T\}_{u \in \mathcal{V}_i}). \tag{5}$$

Program Representation

To form an overall representation of the program, we concatenate representations from all views:

$$\mathbf{z} = \mathbf{z}_{\text{DFG}} \oplus \mathbf{z}_{\text{CFG}} \oplus \mathbf{z}_{\text{RWG}} \oplus \mathbf{z}_{\text{CG}}, \tag{6}$$

where $\mathbf{z}_{\text{DFG}}, \mathbf{z}_{\text{CFG}}, \mathbf{z}_{\text{RWG}}, \mathbf{z}_{\text{CG}}$ are representations for DFG, CFG, RWG, and CG respectively computed as Equation 5, $\oplus$ denotes concatenation.

In summary, our proposed MVG method is outlined in Algorithm 1.

Algorithm 1: MVG Program Representation Method

Input: Source code of a program;
Output: The vector representation of the input program;

```
1: Construct DFG, CFG, and RWG;
2: Construct CG based on DFG, CFG, and RWG;
3: for $\mathcal{G}_i \in \{\mathcal{G}_{DFG}, \mathcal{G}_{CFG}, \mathcal{G}_{RWG}, \mathcal{G}_{CG}\}$ do
4:     $\forall u \in \mathcal{V}_i$, initialize its hidden representation with the
       one-hot encoding of the node type: $\mathbf{h}_u^0 = \mathbf{x}_u$;
5:     Iteratively update node hidden representation with a
       GGNN for $T$ steps (Eq. 2-4);
6:     Compute the graph representation $\mathbf{z}_i$ as Eq. 5;
7: end for
8: Obtain the program representation $\mathbf{z}$ as Eq. 6;
9: Feed $\mathbf{z}$ to downstream tasks, e.g., algorithm detection;
```

Experiments

In this section, we evaluate our proposed MVG method on two algorithm detection datasets `POJ-104` and `ALG-109`. The implementation for our proposed MVG model and the datasets are available at https://github.com/githubg0/mvg.

Baselines

We compare our MVG method with four representative program representation methods in the recent literature.

- **NCC** (Ben-Nun, Jakobovits, and Hoefler 2018) is a sequence-based method that compiles programs into intermediate representations (IRs) and obtains program representations with the skip-gram algorithm.
- **TBCNN** (Mou et al. 2016) is a tree-based method that extracts features from program ASTs.
- **LRPG**[1] (Allamanis, Brockschmidt, and Khademi 2018) is a graph-based method. It introduces semantic edges such as control flows and data dependencies into the AST and extracts program features from the resulted graph.
- **Devign** (Zhou et al. 2019) is an extension of LRPG and improves the performance by including more types of control-flow and data dependency edges.

`POJ-104`: Algorithmic Problem Classification

Dataset description `POJ-104` is a public dataset that contains source code solutions for algorithmic programming problems on the Peking University online judge[2] (Mou et al. 2016). This dataset contains 52,000 programs, and each program is labeled with an algorithmic program ID. In total 104 problems are included, corresponding to a multi-class single-label classification problem with 104 classes. Typically, a particular algorithmic problem will require the solution code to contain some certain algorithms or data structures to obtain the correct answer. Therefore, there is an implicit mapping between the Problem ID labels and algorithm types. The statistics for this dataset is listed in Table 1.

Implementation details We implement a rule-based parser to pre-process the source code of the input programs to obtain DFG, CFG, RWG, and we merge DFG, CFG and RWG to generate the CG. To predict the label of the input programs, we feed its program representation to a two-layer multilayer perceptron (MLP) wrapped

[1]LRPG: In the published paper, this model is called GGNN, which may be confused with gated graph neural networks. Here we refer to it as LRPG by taking the abbreviation of the paper title.

[2]Peking University online judge (POJ): http://poj.org/.

Table 1: Dataset statistics.

	POJ-104	ALG-109	ALG-10
Classification	Single-label	Multi-label	Multi-label
Label	Problem ID	Algorithms	Algorithms
#Classes	104	109	10
#Samples	52,000	11,913	7,974
Average #lines	36.26	94.27	94.37
Average #labels	1.00	1.94	1.70
Language	C	C/C++	C/C++

Table 2: Experiment results on POJ-104.

Method	NCC	TBCNN	LRPG	Devign	**MVG**
Accuracy(%)	94.83	94.00	90.31	92.82	**94.96**

by the Softmax function. The dimension is selected from $\{100, 120, 140, 160, 180, 200\}$, the iterations T for message propagation is selected from $\{1, 2, 4, 8\}$. We use the Adam optimizer (Kingma and Ba 2014) to train the model, the learning learning rate is selected from $\{1 \times 10^{-3}, 6 \times 10^{-4}, 3 \times 10^{-4}, 1 \times 10^{-4}\}$. For all the baselines, they share the same classifier with our method to make the result comparable, other hyperparameters are carefully tuned to the best performance.

Results and discussion Following previous work (Mou et al. 2016; Bui, Yu, and Jiang 2020), we evaluate the accuracy of model predictions on POJ-104. The higher accuracy denotes the better performance. The experiment results are shown in Tables 2.

From the results we can see that MVG achieves the highest accuracy 94.96%. However, other baselines can also achieve a very high accuracy, *e.g.*, 94.83% and 94.00%. We assume this is because algorithmic problem classification is too easy for the models. For example, algorithmic problems will often require some certain input and output formats, and this could leak information to the models, providing them a shortcut to find the right answers. Therefore, we do need a more challenging dataset to further distinguish program representation models.

ALG-109: Algorithm Classification

Dataset description As mentioned above, the algorithmic problem classification dataset POJ-104 is too easy to distinguish the representation power of compared models. Besides, there is no other public annotated algorithm detection dataset in the literature. Therefore, we construct a more realistic and more challenging algorithm classification dataset ALG-109 by ourselves to serve as a new benchmark. ALG-109 contains 11,913 pieces of source code collected from the the CSDN website[3]. Each program is labeled with the algorithms and data structures that appear in the source code. So different from POJ-104, the ALG-109 dataset corresponds to a much harder multi-class multi-label classification problem. The algorithm labels are annotated by previous programming contest participants who have adequate domain knowledge. Overall, 109 algorithms and data structures are considered. The most frequently appearing ten algorithms are listed in Table 3, and we denote this subset as ALG-10. The statistics of the constructed dataset are listed in Table 1.

[3]The CSDN website: https://www.csdn.net/.

Table 3: Most frequent ten algorithms in ALG-109, denoted as ALG-10.

	Algorithm	**#Samples**		**Algorithm**	**#Samples**
1	Recursion	4365	6	Enumeration	681
2	DepthFirstSearch	3117	7	GreedyAlgorithm	557
3	BreadthFirstSearch	1407	8	Recurrence	551
4	Queue	1083	9	DisjointSetUnion	548
5	SegmentTree	775	10	QuickSort	501

Table 4: Experiment results on ALG-109 and ALG-10.

	Method	**Micro-F1(%)**	**Exact Match(%)**	**Ham-Loss(%)**
ALG-109	NCC	48.96 ± 0.91	21.01 ± 1.24	1.61 ± 1.24
	TBCNN	35.03 ± 3.54	9.13 ± 1.34	1.44 ± 0.01
	LRPG	60.56 ± 0.87	30.14 ± 1.33	1.09 ± 0.02
	Devign	56.90 ± 1.57	27.67 ± 1.04	1.16 ± 0.02
	MVG	$\mathbf{65.26 \pm 0.85}$	$\mathbf{36.27 \pm 0.67}$	$\mathbf{1.03 \pm 0.02}$
ALG-10	NCC	72.18 ± 0.89	46.46 ± 1.34	9.29 ± 0.28
	TBCNN	67.53 ± 0.79	34.34 ± 0.96	9.88 ± 0.46
	LRPG	78.48 ± 1.51	55.21 ± 2.85	7.31 ± 0.59
	Devign	78.40 ± 0.98	55.85 ± 1.88	7.16 ± 0.23
	MVG	$\mathbf{80.15 \pm 0.86}$	$\mathbf{58.36 \pm 1.99}$	$\mathbf{6.67 \pm 0.29}$

Implementation details We implement a rule-based parser to pre-process the code to obtain DFG, CFG and RWG, and we merge DFG, CFG and RWG to obtain the CG. To predict the algorithms in the programs, we feed program representation to a two-layer MLP wrapped by a Sigmoid function to obtain the occurrence probability of each algorithm. If the occurrence probability of an algorithm is larger than 0.5, we consider it as one of algorithms which implement the corresponding program. The dimension is selected from $\{120, 144, 168, 192, 216\}$, the iterations T for message propagation is selected from $\{1, 2, 4, 8\}$. We use the Adam optimizer (Kingma and Ba 2014) to train the model, the learning learning rate is selected from $\{1 \times 10^{-3}, 6 \times 10^{-4}, 3 \times 10^{-4}, 1 \times 10^{-4}\}$. For the baselines, they share the same prediction layer with our method to make the result comparable, other hyperparameters are carefully tuned to the best performance.

Results and discussion We evaluate the performance of models on the testing data with three different metrics: the micro-F1 score, the exact match accuracy, and the Hamming loss. The higher micro-F1 score and exact match accuracy indicates a superior performance, while a lower Hamming loss stands for the better. The experiment results on ALG-109 and ALG-10 are shown in Table 4.

From Table 4, we observe that: (1) Our proposed MVG

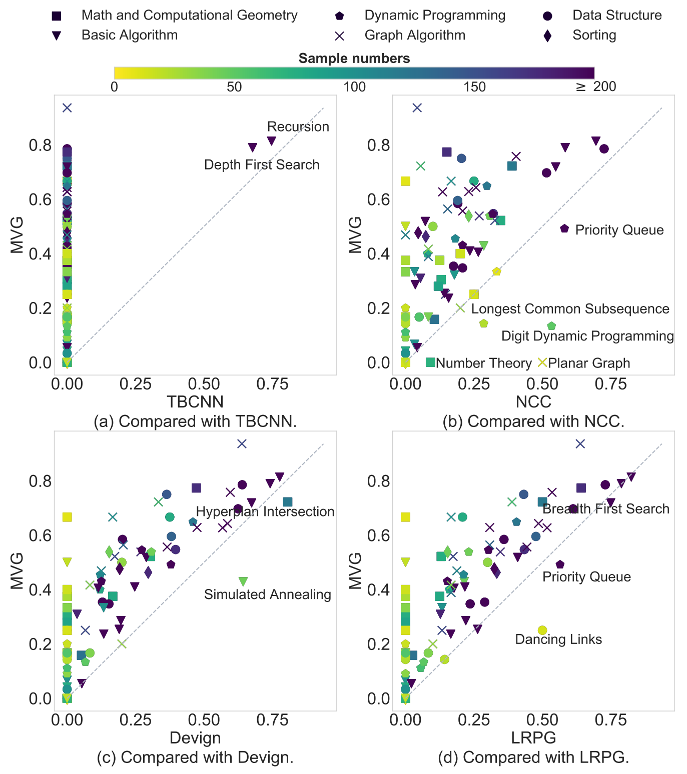


Figure 2: Comparing MVG and baselines on algorithm labels. In each subplot, each point represents one particular algorithm label. The y-coordinates of points are accuracy obtained by MVG, while the x-coordinates of points are accuracy obtained by the baselines. The number of samples and algorithm types of each label are distinguished by the color and point style respectively. A point lying above $y = x$ means MVG is performing better than the baseline for this algorithm label.

method surpasses all the baselines significantly on both ALG-109 and ALG-10, illustrating MVG's superior performance on algorithm classification. (2) Graph-based methods (*i.e.*, LRPG, Devign, and MVG) all perform remarkably better than the sequence-based method (*i.e.*, NCC) and the tree-based method (*i.e.*, TBCNN), showing the great potential of representing programs as graphs. (3) All models' performances drop when moving from ALG-10 to ALG-109, because labels in ALG-10 will be bound more training data. However, we can see the gap of MVG is smaller that others, which means MVG is relatively less sensitive to the insufficiency of data. (4) Comparing with the experiment results from POJ-104, we find the methods are more distinguishable on ALG-109, and there is still large room for models to further improve their performances on this dataset. Therefore, our constructed ALG-109 dataset might serve better as an algorithm detection or PLP program representation benchmark.

To further investigate how these models perform dissimilarly on each specific algorithm label, we compare MVG with the baselines and visualize the results in Figure 2. From

Table 5: Ablation study on ALG-109.

Variant	Micro-F1(%)	Exact Match(%)	Ham-Loss(%)
MVG	**65.26 ± 0.85**	**36.27 ± 0.67**	**1.03 ± 0.02**
+AST	65.19 ± 0.94	36.10 ± 1.25	1.04 ± 0.02
– DFG	62.34 ± 1.11	32.67 ± 0.85	1.09 ± 0.02
– CFG	64.18 ± 0.86	34.72 ± 1.08	1.06 ± 0.02
– RWG	64.01 ± 1.06	35.00 ± 0.91	1.06 ± 0.03
– CG	64.38 ± 0.78	34.86 ± 0.93	1.06 ± 0.02
OnlyCG	62.02 ± 0.74	32.06 ± 0.99	1.09 ± 0.02

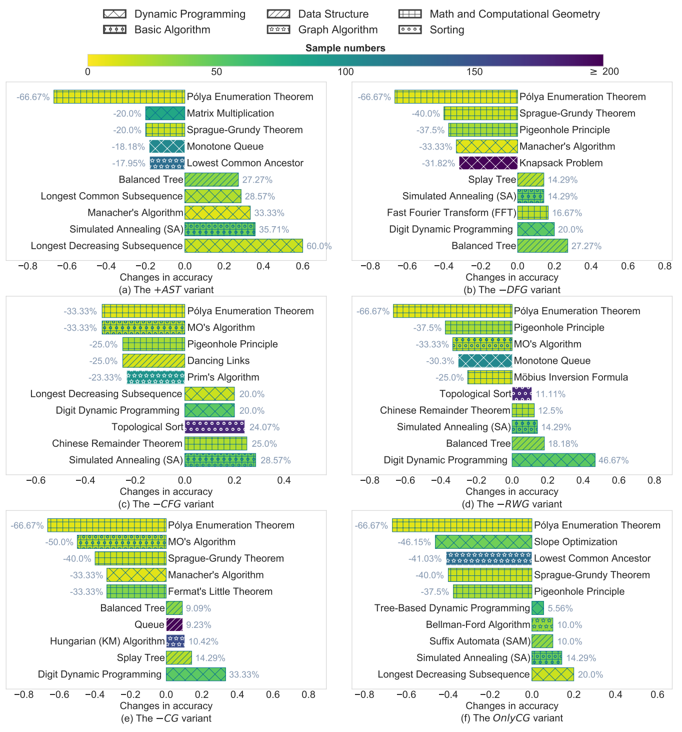


Figure 3: Algorithm classification accuracy changes of model variants. For each variant, the top fives labels with largest performance drops as well as increases are shown.

the visualization we can see that, for almost all algorithm labels, MVG will perform superior than the baselines, especially for the labels with insufficient data.

Ablation study To obtain a deep understanding on MVG's outstanding performance, we conduct some further ablation studies to learn each view's impact on the MVG model. Here, we consider six variants:

- **+AST** adds an abstract syntax tree (AST) view to MVG, which will include more syntactic information;
- **-DFG** removes the DFG view from MVG. The data-flow information in CG is also removed accordingly.
- **-CFG** removes the CFG view from MVG. The control-flow information in CG is also removed accordingly.
- **-RWG** removes the RWG view from MVG. The read-write information in CG is also removed accordingly.

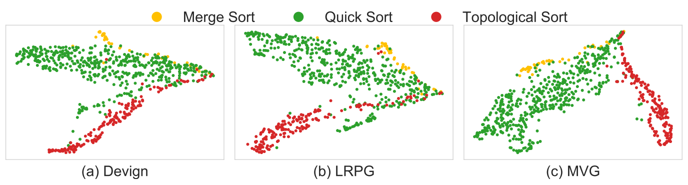
- **-CG** removes the combined CG view from MVG, so the other three views (*i.e.*, DFG, CFG, RWG) will no longer interact with each other.
- **OnlyCG** contains only the combined CG view, so the data-flow, control-flow, and read-write information will be mixed up together into one single view.

The ablation study results are listed in Table 5. From the results we find that: (1) Removing any view from MVG (*i.e.*, -DFG, -CFG, -RWG, and -CG) will cause a drop in performance, showing the indispensable role of every view in program representation. (2) Adding the AST view (*i.e.*, +AST) harms the performance slightly, which means the AST view is unnecessary in algorithm detection task and we should not emphasize too much on the syntax in program representation. (3) Removing CG undermines the accuracy, meaning interactively combining the other three views together helps MVG to better understand the programs. On the other hand, the performance of the OnlyCG variant is also inferior than MVG. Therefore, we can conclude that both the independent views (*i.e.*, DFG, CFG, RWG) and the integral view (*i.e.*, CG) are necessary for our program representation. (4) Comparing all the variants, we find by deleting the DFG view, the performance drops the most, showing that DFG is most critical in our program representation model.

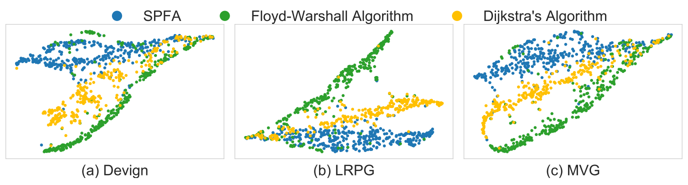
We also examine how the performance on different algorithms changes when using different model variants. The results are displayed in Figure 3. Here, for each model variant, we show the top fives labels with largest performance drops and increases. From the results we observe that: (1) Overall, by changing the design of MVG into other variants, the performance drops more while increases less; (2) Compared with the variants, our MVG seems has better representation for programs that contain *Math and Computational Geometry* algorithms, *e.g.*.., `Pólya Enumeration Theorem`, since when replacing MVG with other variants, the detection performance on these algorithms drop significantly.

Case study To intuitively figure out whether our MVG model can render better program representations, we use UMAP (McInnes, Healy, and Melville 2018) to visualize the representation vectors encoded by the three best-performing models (*i.e.*, MVG, Devign, and LRPG) in Figure 4.

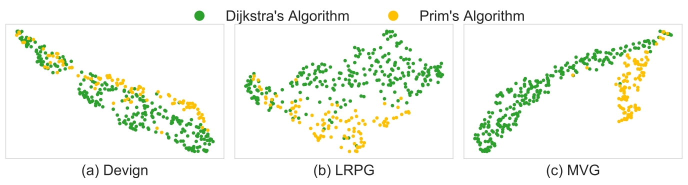
Three groups of algorithms are compared: (1) We first compare three sorting algorithms (*i.e.*, `Merge Sort`, `Quick Sort`, and `Topological Sort`). From the visualization we can see that, both Devign and LRPG can not distinguish `Quick Sort` and `Topological Sort` well, while MVG represents these two algorithms more differently. (2) For the shortest-path algorithms, the representation power of Devign, LRPG, and MVG seem almost the same. (3) We also compare `Dijkstra's Algorithm` and `Prim's Algorithm`, since they are designed for different intentions while having very similar implementations (*i.e.*, both the two algorithms utilize the `Breadth-First Search`). From the visualization, we can see MVG gives a much more clear decision boundary of the two algorithms, meaning our method has higher representation power than the other two baselines. (4) Associating the results presented in Table 4 and Figure 2, we find MVG is more capable of representing source code especially under the context of algorithm detection.



(a) Sorting algorithms.



(b) Shortest path algorithms



(c) `Dijkstra's Algorithm` and `Prim's Algorithm`.

Figure 4: Visualization of program representations.

Conclusion

This paper presents a multi-view graph (MVG) program representation method for PLP. To understand source code more comprehensively and semantically, we propose to include four graph views of different levels and various aspects: (1) the data-flow graph (DFG); (2) the control-flow graph (CFG); (3) the read-write graph (RWG); and (4) an integral combined graph (CG). Our proposed MVG method is evaluated mainly on the algorithm detection task, which is an important and challenging subfield of PLP. To fill the vacancy of a high-qualified algorithm classification dataset, we construct the `ALG-109` dataset which contains 109 algorithm or data structure classes in total. In experiments, our MVG achieves the state-of-the-art performance, demonstrating its outstanding capability of representing programs.

For the future work, it would be interesting to investigate how our MVG approach can be combined with other orthogonal techniques like pre-training. Moreover, we might also try to apply our MVG model and the annotated dataset `ALG-109` for the purpose of programming education.

Acknowledgement

We would like to thank Enze Sun and Hanye Zhao from Shanghai Jiao Tong University. The work is partially supported by Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102) and National Natural Science Foundation of China (62177033).

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. Compilers, principles, techniques. *Addison wesley*, 7(8): 9.

Allamanis, M.; Barr, E. T.; Devanbu, P.; and Sutton, C. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4): 81.

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs.

Allen, F. E. 1970. Control flow analysis. In *ACM Sigplan Notices*, volume 5, 1–19. ACM.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL): 40.

Ben-Nun, T.; Jakobovits, A. S.; and Hoefler, T. 2018. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, 3585–3597.

Bui, N. D.; Yu, Y.; and Jiang, L. 2020. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. *arXiv preprint arXiv:2009.09777*.

Chen, X.; Liu, C.; and Song, D. 2018. Tree-to-tree neural networks for program translation. In *NIPS'18 Proceedings of the 32nd International Conference on Neural Information Processing Systems*, volume 31, 2552–2562.

Cho, K.; Van Merriënboer, B.; Bahdanau, D.; and Bengio, Y. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Ciniselli, M.; Cooper, N.; Pascarella, L.; Poshyvanyk, D.; Di Penta, M.; and Bavota, G. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. *arXiv preprint arXiv:2103.07115*.

Farrow, R.; Kennedy, K.; and Zucconi, L. 1976. Graph grammars and global program data flow analysis. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 42–56. IEEE.

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. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 1536–1547.

Fraser, C. W.; and Hanson, D. R. 1995. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc.

Gu, X.; Zhang, H.; Zhang, D.; and Kim, S. 2017. DeepAM: migrate APIs with multi-modal sequence to sequence learning. In *IJCAI'17 Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 3675–3681.

Harer, J. A.; Kim, L. Y.; Russell, R. L.; Ozdemir, O.; Kosta, L. R.; Rangamani, A.; Hamilton, L. H.; Centeno, G. I.; Key, J. R.; Ellingwood, P. M.; et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

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

Le, Q.; and Mikolov, T. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*, 1188–1196.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. S. 2016. Gated Graph Sequence Neural Networks. In *ICLR*.

Lv, F.; Zhang, H.; Lou, J.-g.; Wang, S.; Zhang, D.; and Zhao, J. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 260–270. IEEE.

McInnes, L.; Healy, J.; and Melville, J. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

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

Muchnick, S.; et al. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

Nie, L.; Jiang, H.; Ren, Z.; Sun, Z.; and Li, X. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5): 771–783.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2015. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*.

Wang, K. 2019. Learning Scalable and Precise Representation of Program Semantics. *arXiv preprint arXiv:1905.05251*.

Wang, K.; Singh, R.; and Su, Z. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*.

Wang, R.; Zhang, H.; Lu, G.; Lyu, L.; and Lyu, C. 2020. Fret: Functional reinforced transformer with BERT for code summarization. *IEEE Access*, 8: 135591–135604.

Xia, X.; Bao, L.; Lo, D.; Xing, Z.; Hassan, A. E.; and Li, S. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering*, 584–584.

Yao, Z.; Peddamail, J. R.; and Sun, H. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *The World Wide Web Conference*, 2203–2214. ACM.

Zhou, Y.; Liu, S.; Siow, J.; Du, X.; and Liu, Y. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, 10197–10207.

Zuo, F.; Li, X.; Zhang, Z.; Young, P.; Luo, L.; and Zeng, Q. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*.