Utilizing Edge Attention in Graph-Based Code Search

Wei Zhao
School of Software Engineering
Tongji University
Shanghai, China
2031569@tongji.edu.cn

Yan Liu*
School of Software Engineering
Tongji University
Shanghai, China
yanliu.sse@tongji.edu.cn

***Abstract*—Code search refers to searching code snippets with specific functions in a large codebase according to natural language description. Classic code search approaches, using information retrieval technologies, fail to utilize code semantics and bring noisy and irrelevant keywords. During the last recent years, there has been an ample increase in the number of deep learning-based approaches, which embeds lexical semantics into unified vectors to achieve higher-level mapping between natural language queries and source code. However, these approaches are struggling with how to mine and utilize deep code semantics. In this work, we study how to leverage deeper source code semantics in graph-based source code search, given graph-based representation is a promising way of capturing program knowledge and has rich explainability. We propose a novel code search approach called EAGCS (Edge Attention-based Graph Code Search), which is composed of a novel code graph representation method called APDG (Advanced Program Dependence Graph) and a graph neural network called EAGGNN (Edge Attention-based GGNN) which can learn the latent code semantics of APDG. Experiment results demonstrate that our model outperforms the GGNN-based search model and DeepCS. Moreover, our comparison study shows that different edge enhancement strategies have different contributions to learning the code semantics.**

Keywords—semantic code search; graph neural network; graph representation learning

I. INTRODUCTION

Code search is one of the most common activities in software development, some studies [1][2] have shown that 19% of developers' time will be spent searching desirable code snippets. Especially in recent years, with the rise of agile development [3], the developer needs to iterate the projects rapidly. Accurate search results can be reused with only slight reconstruction and help quickly realize specific project functions to boost developers' productivity. However, the existing code search engines, such as those on GitHub [4] and Stack Overflow [5], treat the source code as plain text and search the code snippets mainly based on keyword matching, lacking the semantic understanding of natural language and source code. Therefore, it is difficult to use such IR (Information Retrieve) techniques to optimize code search.

More recently, deep learning-based code search can reduce the interference of noisy keywords and learn code features by vectoring the code, which can recognize semantically related words. For example, DeepCS [6] obtained the API sequence, method name and token information according to defined rules and embedded them into unified vectors to represent the code. Therefore, the basic code representation method has a far-reaching impact on the expression of code semantics, which will further affect search performance. As code is structural and has unique language-specific semantics, the graph is a natural and effective representation of code. Taking Java as an example, there are a series of implicit relationships between code elements [7], such as the order of method calls, class inheritance, etc., and these relationships can reveal the potential code semantics. Inspired by this, many researchers utilized variants of abstract syntax tree (AST) [8] and other code graphs to represent latent code semantics. For instance, DeepCS extracted API sequence from AST, and Xiang et al. [9] generated a code graph based on AST with different nodes (terminal/non-terminal nodes) and edges. However, code with different syntax structures may express the same functionality [10], as shown in Figure 1 and Figure 2, which indicates that simply using AST to represent code is not enough to accurately express the deep code semantics. So we need to break through the current limitations of sequential data and enhance code semantics via utilizing the rich structural information behind programs.

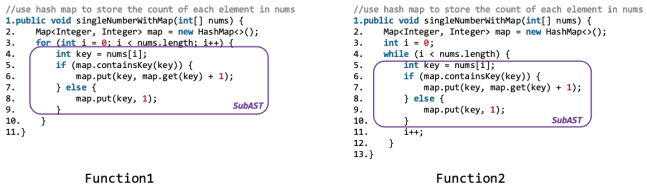

Figure 1. Two code snippets with the same functionality

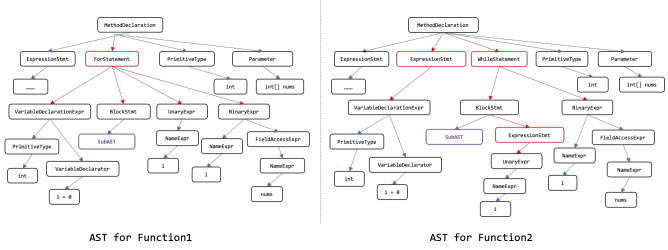

Figure 2. The two ASTs correspond to the code snippets in Figure 1, where the nodes and edges marked in red indicate the structural differences.

To deal with the aforementioned challenges and utilize structural information, we propose a novel code search approach called EAGCS, which can significantly enhance the expression of structural and semantic information of source code. More specifically, we first construct a statement-level advanced

* Corresponding Author

DOI reference number: 10.18293/SEKE2022-078

program dependence graph (APDG) which transforms three common control statements and adds control and data edges to improve the awareness of neighbors. Program dependence graph (PDG) [11] is the graphical representation of a program where nodes represent program statements and edges represent latent dependence information, but it fails to reflect the order in which statements are executed and the implicit control logic [12]. In APDG, we enrich code semantics based on our defined data and control dependence rules, introducing the statement execution information and control logic missed in PDG. Besides, the APDG is constructed based on AST, which can keep the syntax information of source code, and statement-level graph nodes can preserve local semantics compared with excessive fine granularity nodes in AST (i.e., *NameExpr* and *ExpressionStmt* nodes in Figure 2). Concentrating on multiple edge types, we apply EAGGNN to learn the semantic features of ADPG. Furthermore, we calculate the cosine similarity of the code and description vector embedded by our model and search the top-k relevant code snippets according to the given user queries. Experiments have been conducted and the results demonstrate that our model outperforms the other start-of-the-art models.

The main contributions of this paper are as follows:

- We introduce a novel statement-level code representation method called APDG, which optimizes the traditional PDG and strengthens both data and control dependence information to enrich the edge semantics.
- We propose EAGCS, a graph-based code search approach that enhances the GGNN [13] via an edge attention mechanism to improve the expression of code semantics in APDG.
- We conduct experiments on our model and other start-of-the-art models and the results have shown that our model outperforms the others. Besides, we also explore the impact of different edges on the model performance.

II. Related Work

With the in-depth study of code search in recent years, a variety of research methods have been proposed. Traditional code search methods treated source code as plain text and obtained the most relevant code snippets through the information retrieve (IR) technology. For example, Lv et al. [14] designed CodeHow, which expanded the user query with the APIs and applied an extended boolean model to perform code search. While Portfolio [15] combined keyword matching and PageRank to retrieve a series of functions according to user input.

To solve the problem that code snippets without keywords related to the description cannot be searched in the above-mentioned models, Gu et al. [6] proposed the first deep learning-based code search tool named DeepCS. DeepCS embedded code snippets and natural language descriptions into high-dimensional vector space separately, which can recognize semantically related words. On this foundation, some other previous works used an attention mechanism or reconstructed the model structure to boost the search performance. Shuai et al. [16] utilized CARLCS-CNN based on Convolutional Neural Network (CNN) instead of LSTM used in DeepCS and built a semantical correlation between the code and description vectors via a co-attention mechanism. The work in [17] applied a self-attention network to learn the contextual representation and global semantic relations for code snippets and their corresponding queries.

Some other researchers are mainly dedicated to enhancing the representative ability of code semantics. Wan et al. [18] constructed the code features with the sequential tokens, ASTs (abstract syntax trees), and CFGs (control-flow graphs) to represent syntactic and semantic information of source code. Similarly, Zeng et al. [10] encoded source code into variable-based flow graphs and utilized an improved gated graph neural network (GGNN) to model more precise code semantics. Liu et al. [19] transformed code snippets and descriptions into ASTs and dependence parsing graphs separately to capture their joint semantic relationship.

In addition to semantic enhancement of source code, some work focused on query expansion and reinforcement. For instance, Sirres et al. [20] augmented user query with program elements, such as method and class names, from the extracted snippets. Xuan et al. [21] proposed DERECS to reinforce the code based on the method call and the structural characteristics of the code fragment, which reduced the difference between source code and query.

III. EAGCS

A. Overview

Figure 3 shows the overall structure of EAGCS, including 4 components: preprocessing, code graph generation, description embedding, and code graph embedding. Despite AST can reflect the syntax information of the source code, it is complex so we need to prune it to remove redundant nodes. The APDG we proposed not only retains the syntax information of AST in statement level which reduces the scale of code graph, but also adds data and control edges to enhance the code semantics. When the model searches code snippets, code semantics are explicitly expressed through multiple graph edges in APDG. Moreover, the edge attention-based GGNN (EAGGNN) can help to obtain a deeper understanding of APDG, it learns the node embeddings from multiple edges to focus on both data and control dependence. After embedding both descriptions and code snippets into unified vectors, the model can recommend code snippets with higher cosine similarity according to the given user query.

B. Preprocessing

1) Data Collection: To guarantee the performance of our model, we need a large-scale dataset that contains query descriptions and their corresponding code snippets. The dataset provided by DeepCS contains more than 18 million data items, which is used in most existing studies and is our ideal dataset. Unfortunately, the code graph generation approach we designed needs to be applied through the source code (raw data)， but the dataset provided by DeepCS has been already preprocessed and embedded, which fails to provide in-depth semantic information. Therefore we choose the dataset published by CodeSearchNet[1], which is a little less than that of DeepCS but

[1]https://github.com/github/CodeSearchNet

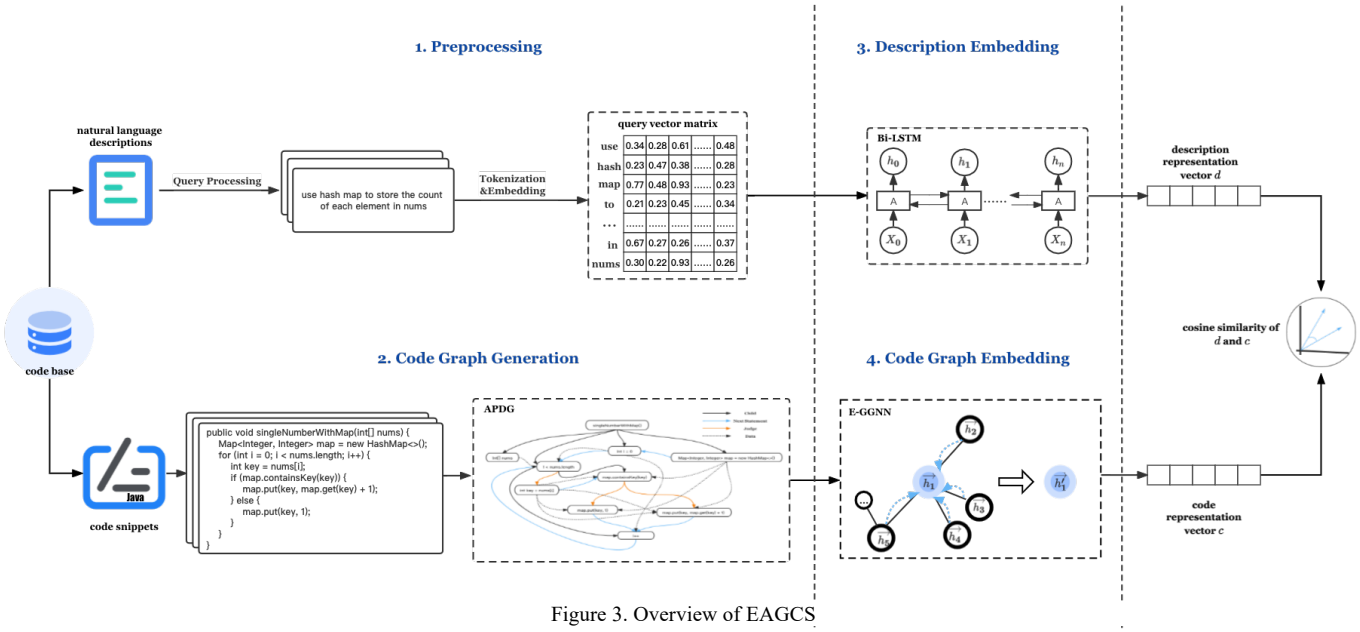

Figure 3. Overview of EAGCS

contains raw code snippets.

2) Data Processing: The CodeSearchNet dataset includes code snippets extracted from real projects. However, due to the complexity of the actual development process, participants, and project types, the raw data contains noisy and dirty data items. So we need to go through several processes to improve the quality of data fed to the model.

Query Processing: We take the comments corresponding to the code snippets as the user query and handle them according to the following flow.

- Segment the comment according to the "." character, and select the first sentence as a user query. (User comment may include multiple sentences, but the first sentence is usually complete enough to describe the code, and the following statements are only for supplementary explanation)
- Remove non-English symbols and stop words[1].

Code processing: we apply the *javaparser*[2] library to transform the code snippets and generate APDG based on our code graph generation approach.

- Convert the code snippets into class. (The original code fragment is at the function level)
- Transform the code snippet into APDG.
- Delete non-English characters and generate words according to lower CamelCase for each statement-level node.

Tokenization: We first count the frequency of words in query and node statements separately and build two dictionaries based on their top 10000 words. Furthermore, we represent each word with a unique numeric ID and transform the query statements and node statements into sequences of numerical tokens.

Embedding: Embedding is a technology that maps an object (i.e., a word) into a real vector, which can make objects with similar meanings have vectors with similar distances. The commonly used word embedding approaches are CBOW [22] and Skip-Gram [23]. In our work, we use the *nn.Embedding*[3] function of *pytorch*[4] to initialize the word embeddings and then retrieve them using indices.

C. Code Graph Generation Approach

Program dependence graph (PDG) is one of the most widely used directed code graphs [11], where the nodes represent program statements and the edges represent the interdependence between program statements, but it fails to reflect the order in which statements are executed [12]. In our work, we propose advanced PDG (APDG) which adds *NextStatement* edges in PDG and we also provide clear guidelines for the optimization of common control statements to explore more program semantics. More specifically, we consider 12 types of nodes: *Method Declaration*, *Parameter*, *Unary Expression*, *Variable Declaration Expression*, *Method Call Expression*, *Assign Expression*, *Construction Declaration*, *Try Statement*, *Class or Interface Declaration*, *Condition*, *Return Statement* and *Assert Statement* based on the AST and the Soot's [24] internal representation. Moreover, we have also defined the extraction rules of control dependence (*Child, NextStatement*, *Judge* Edges) and data dependence, which will be formally described in detail:

1) Control Dependence: control dependence defines the constraint relationship of statement execution, which can reflect the syntax, execution order, and control information.

Child: *Child* edges connect parent and child nodes in AST and point from parent node to child node which can reflect the control dependence of statements at the syntactic level.

NextStatement: *NextStatement* edges concatenate statements inside a block according to the context, indicating the order of statement execution. The dashed box represents the virtual structure for better illustration, which will not appear in real APDG.

[1]https://www.textfixer.com/tutorials/common-english-words.txt
[2]https://github.com/javaparser/javaparser
[3]https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
[4]https://pytorch.org/

Judge: We transform three common control statements in AST: *IfStatement*, *ForStatement, WhileStatement* and add *Judge* edges to uncover the control logic.

2) Data Dependence: data dependence defines the constraint relationship of variables between statements. To mine the data dependence relationship, we have to keep a record of all accesses of all variables. The data dependence rule of variable *v* from statements s_1 to s_2 can be described as:

- *v* is defined or assigned in statement s_1.
- *v* is used in statement s_2.
- The scope of s_2 is inside the scope of s_1.
- If s_1 and s_2 have the same scope level, a *NextStatement* path exists from s_1 to s_2.

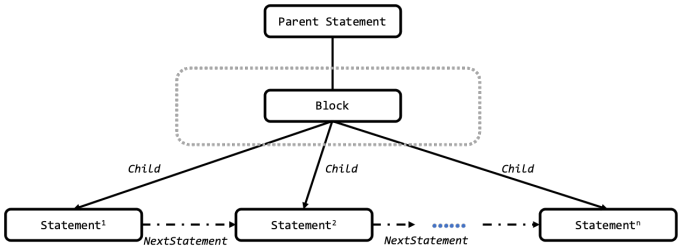

Figure 4. Illustration of Child and NextStatement edges.

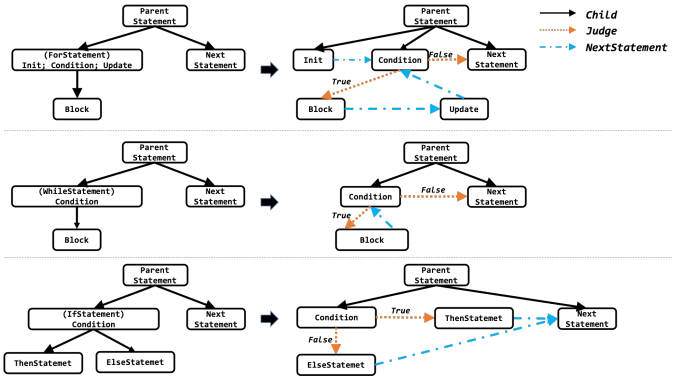

Figure 5. Illustration of control statements optimization and Judge edges.

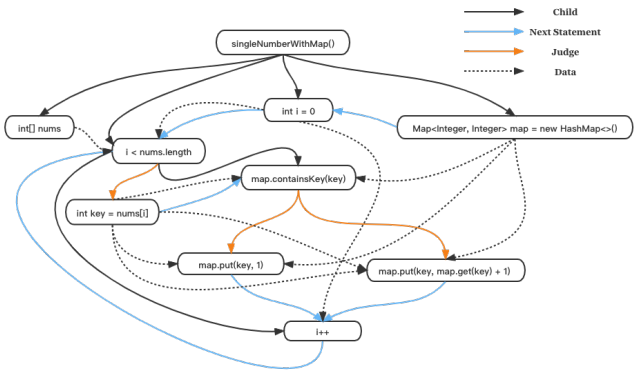

Figure 6. APDG corresponds to the code snippets in Figure 1.

As shown in Figure 6, we constructed APDG for the code snippets in Figure 1 based on the designed control dependence and data dependence rules. And latent code semantics can be expressed through node contents and multiple edges.

D. Description Embedding

By word embedding, we view a description $\boldsymbol{D}$ as a sequence of token vectors: $\boldsymbol{w_1}, \ldots, \boldsymbol{w_N}$, $\boldsymbol{D} = \{\boldsymbol{w_1}, \ldots, \boldsymbol{w_N}\}$. We use a bi-LSTM to extract semantic information from the input in both forward and reverse directions, and embed the description into a vector $\boldsymbol{d}$.

$$\overrightarrow{\boldsymbol{h_t}} = \overrightarrow{LSTM}\left(\boldsymbol{w_t}, \overrightarrow{\boldsymbol{h_{t-1}}}\right) \quad (1)$$

$$\overleftarrow{\boldsymbol{h_t}} = \overleftarrow{LSTM}\left(\boldsymbol{w_t}, \overleftarrow{\boldsymbol{h_{t+1}}}\right) \quad (2)$$

$$\boldsymbol{h_t} = \overrightarrow{\boldsymbol{h_t}} \oplus \overleftarrow{\boldsymbol{h_t}} \ \forall t = 1, \ldots, N \quad (3)$$

$$\boldsymbol{d} = \boldsymbol{maxpooling}(\boldsymbol{h_1}, \ldots, \boldsymbol{h_N}) \quad (4)$$

where $\boldsymbol{w_t} \in \mathbb{R}^d$, $\boldsymbol{h_t}$ is the hidden states at step *t*, $t = 1, \ldots, N$, *N* is the length of the sequence, $\oplus$ is the concatenation operation.

E. Code Graph Embedding

1) Node Embedding: We view a graph node $\boldsymbol{V}$ as several token vectors: $\boldsymbol{t_1}, \ldots, \boldsymbol{t_M}$. $\boldsymbol{V} = \{\boldsymbol{t_1}, \ldots, \boldsymbol{t_M}\}$. For there is no strict order between these tokens, we use a multilayer perceptron (MLP) to embed the node into a vector $\boldsymbol{n}$.

$$\boldsymbol{h_i} = \tanh(W^T \boldsymbol{t_i}) \ \forall i = 1, \ldots, M \quad (5)$$

$$\boldsymbol{n} = maxpooling([\boldsymbol{h_1}, \ldots, \boldsymbol{h_M}]) \quad (6)$$

where $\boldsymbol{t_i} \in \mathbb{R}^d$, $\boldsymbol{i} = 1, \ldots, M$, *M* is the number of the tokens, $\boldsymbol{W^T}$ is the learnable matrix in the MLP.

2) Edge Attention Based GGNN

Gated Graph neural network (GGNN) is a kind of graph neural network (GNN) that directly uses graph data as the structured input. For most GNNs, much information sharing might reduce the weight of the original information of the node itself, which can lead to overfitting. To overcome this issue, GGNN can selectively remember the hidden information of neighbor nodes and the hidden information in the process of node iteration by adding a GRU [13] component. As we have already enriched edge semantics in APDG, we propose EAGGNN which enhances the GGNN via an edge attention mechanism to deal with different types of edges to focus on both data and control dependence information for each iteration. Considering the program graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ is the node collection and $\mathcal{E}$ is the adjacency matrix. For each node $\boldsymbol{v} \in \mathcal{V}$, $\boldsymbol{h_v^{(0)}}$ is the initial hidden state of node $\boldsymbol{v}$ through node embedding, and $\boldsymbol{h_v^{(k)}}$ is the hidden state of node $\boldsymbol{v}$ in hop $\boldsymbol{k}$.

$$\boldsymbol{a_{v|D}^{(k)}} = A_{v|D}^{\tau}\left[\boldsymbol{h_1^{(k-1)\tau}} \ldots \boldsymbol{h_{|\mathcal{V}|}^{(k-1)\tau}}\right] + \boldsymbol{b} \quad (7)$$

$$\boldsymbol{a_{v|C}^{(k)}} = A_{v|C}^{\tau}\left[\boldsymbol{h_1^{(k-1)\tau}} \ldots \boldsymbol{h_{|\mathcal{V}|}^{(k-1)\tau}}\right] + \boldsymbol{b} \quad (8)$$

$$\boldsymbol{a_v^{(k)}} = \boldsymbol{a_{v|D}^{(k)}} \oplus \boldsymbol{a_{v|C}^{(k)}} \quad (9)$$

$$\boldsymbol{z_v^{(k)}} = \sigma\left(W^z \boldsymbol{a_v^{(k)}} + U^z \boldsymbol{h_v^{(k-1)}}\right) \quad (10)$$

$$\boldsymbol{r_v^{(k)}} = \sigma\left(W^r \boldsymbol{a_v^{(k)}} + U^r \boldsymbol{h_v^{(k-1)}}\right) \quad (11)$$

$$\widetilde{\boldsymbol{h_v^{(k)}}} = \tanh\left(W \boldsymbol{a_v^{(k)}} + U\left(\boldsymbol{r_v^{(k)}} \odot \boldsymbol{h_v^{(k-1)}}\right)\right) \quad (12)$$

$$\boldsymbol{h_v^{(k)}} = \left(1 - \boldsymbol{z_v^{(k)}} \odot \boldsymbol{h_v^{(k-1)}} + \boldsymbol{z_v^{(k)}} \odot \widetilde{\boldsymbol{h_v^{(k)}}}\right) \quad (13)$$

$$\boldsymbol{c} = maxpooling\left(\boldsymbol{h_1^K} \ldots \boldsymbol{h_{|\mathcal{V}|}^K}\right) \quad (14)$$

where $A_{v|D}$ and $A_{v|C}$ are the columns corresponding to node $\boldsymbol{v}$ in data adjacency matrix and control adjacency matrix

separately, $z_v^{(k)}$ is the update gate, $r_v^{(k)}$ is the reset gate, K is the number of hops in EAGGNN and $\boldsymbol{c}$ is the final code representation.

F. Model Training

Considering a code-description pair $P(c, d^+)$, where $c \in C$, $d^+ \in D$, C denotes the set of code snippets, D denotes the set of descriptions, c denotes the single code snippet, d^+ denotes the corresponding query description of c and d^- denotes another randomly selected query description from D, $d^- \in D$, $d^- \neq d^+$. Then we rebuild the pair P as $P' = (c, d^+, d^-)$ and train the model by minimizing the loss function $L(\theta)$ that is formulated as:

$$L(\theta) = \sum_{(c,d^+,d^-)\in P'} \max(0, \beta - \cos(c, d^+) + \cos(c, d^-)) \quad (15)$$

where θ denotes the model parameters, d^+denotes the positive description, d^-denotes the negative description, *cos* is cosine similarity function and β denotes the constant margin.

IV. Experiments

A. Dataset

We choose the dataset of CodeSearchNet as the training set which contains 454,451 samples, and then filter these samples according to the following rules:

- Remove duplicate queries. (The dataset contains override or overload functions that have the same comments, and we only choose the sample that appears first)
- Remove code snippets that cannot be compiled properly
- P with description d that contains less than 3 words or more than 20 words will be filtered out. (The excessively long query length does not meet the actual user requirements)
- P with code c that is less than 3 lines and more than 20 lines will be filtered out. (Too short code snippets are meaningless, while too long code is difficult to understand)

As result, we obtained 126,363 pieces of data and converted all the processed code snippets into ASTs and APDGs. In addition, statistics on the maximal/average/minimal number of nodes and edges of these ASTs and APDGs were conducted and the results were shown in Figure 7, which indicated that APDG effectively reduced the complexity of code graph and was conducive to model training.

For the test set, we selected the 10,000 code-query pairs provided by Gu et al. [6]. Through the same filtering flow, we gained 4,548 pieces of data for evaluation and utilized an automatic evaluation approach which used the 4,548 queries as model inputs while the corresponding code snippets were treated as ground truth. In the automatic evaluation, we randomly selected 99 other code snippets and combined them with the ground truth as the search base for a query input. This automatic evaluation approach can avoid the bias caused by manual ranking. Besides, selecting training and test sets from different projects can examine the generalization ability of the model.

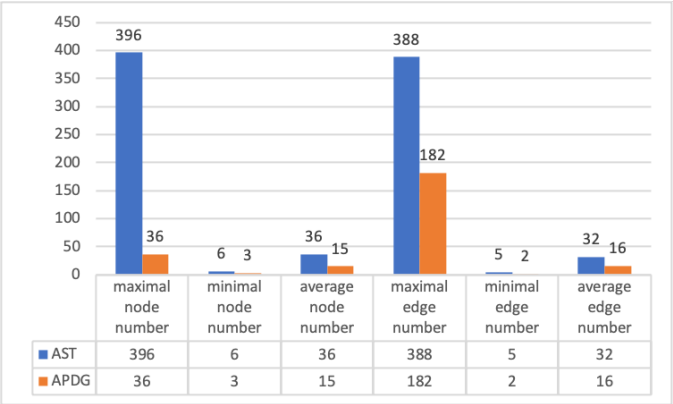

Figure 7. Statistical results of APDGs and ASTs on CodeSearchNet.

B. Evaluation Metrics

SuccessRate@k: The SuccessRate@k measures the percentage of queries for which the corresponding ground-truth code snippet could exist in the top k ranked results and it can be formulated as:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|} \delta(FRank_q \leq k) \quad (16)$$

where $\boldsymbol{Q}$ denotes the set of queries, $\delta(\cdot)$ denotes the function that returns 1 if the input is true and 0 otherwise, Frank denotes the rank position of the ground truth in the result list.

MRR: MRR is the average of the reciprocal ranks of results of a query set $\boldsymbol{Q}$, which can be defined as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|} \frac{1}{Frank_q} \quad (17)$$

where the reciprocal ranks is the inverse of Frank. As we expect to find the ground truth in the top 10 results, we set $1/Frank_q$ to 0 if $FRank_q$ is larger than 10.

C. Baseline Models

DeepCS is the state-of-the-art neural network to retrieve relevant code snippets given a query description. It extracted the method name, API sequence, and tokens of a method to represent the code semantics, and then embedded code and description to get the unified vectors to calculate the similarity between them.

GGNN represents the gated neural network without edge attention based on our APDG. It utilized the GGNN on data and control dependence separately and then combined the two hidden states via an *avgpooling* function to represent the code.

D. Implementation Details

We embedded the top 10000 tokens for the code statements and descriptions separately with a 128-dimensional size and used Adam [25] as the optimizer. The model was trained for 250 epochs while the batch size was set as 100. The iteration times of EAGGNN were set as 4 and the dropout was set as 0.6 in the word embedding layer for the learning process.

E. Results

Table I summarizes the experiment results of our model and the baseline models on the test set. The R@1, R@5, and R@10 denote the results of SuccessRate@k, where k is 1, 5, 10. The results have shown that our model outperforms the state-of-the-art models. The R@1, R@5, R@10, and MRR of EAGCS are respectively 0.15, 0.16, 0.18, and 0.15 higher than GGNN, which indicates the edge attention mechanism can effectively aggregate the edge information.

TABLE I. COMPARISON OF THE MODEL PERFORMANCE BETWEEN OUR MODEL AND THE BASELINE MODEL

Model	R@1	R@5	R@10	MRR
DeepCS	0.2199	0.3628	0.4574	0.2846
GGNN	0.4268	0.5500	0.5910	0.4826
EAGCS	**0.5785**	**0.7071**	**0.7682**	**0.6357**

TABLE II. EFFECT OF EACH EDGE TYPE

Model	D	C	R@1	R@5	R@10	MRR
EAGCS	✓		0.3173	0.4807	0.5706	0.3894
		✓	0.2718	0.4576	0.5521	0.3507
	✓	✓	**0.5785**	**0.7071**	**0.7682**	**0.6357**

Table II presents the influence of different types of edge on search performance. The header D and C indicate whether data edge and control edge exist in APDG, where the checkmark represents that the corresponding edge is added. Incorporating both data and control dependence can express the code semantics to the greatest extent and can get the best model performance. Results also show that data dependence has a slightly greater weight than control dependence for that all metrics are higher. Data dependence reflect the flow of variables between basic blocks under the control structure, which is a further and deeper analysis of program features.

F. Threats to Validity

Our work may suffer from four validity. The first one is the model re-implementation. Replicating the baseline models may introduce some errors. To mitigate this threat, we used the authors' open-source projects on GitHub and processed our dataset into the same format required by the projects. The second one is the selected dataset. Because the dataset provided by DeepCS is vectorized, we can't obtain the original code snippets for our model to generate code graphs, so we utilized the CodeSearchNet dataset for model training, which was smaller than that of DeepCS but contained raw code snippets. Furthermore, the training and test set shared the same preprocessing flow. The third one is the model evaluation. We took the automatic evaluation approach to avoid manual risks. Given an input query, we set the same search base for all baseline models. The experiment results may be influenced by the scale or the origin of the search base, which is our future research content. The last one is the model comparision. In our experiment, We applied the same dataset, ran all the models in the same hardware environment, and adopted the same data preprocessing process to reduce this threat. While DeepCS does not perform on the graph structure, more related baselines may be needed to justify the advantages introduced by our proposed model in the future.

V. CONCLUSION

How to accurately understand and express code semantics has become a key challenge in the process of code search. In this paper, we propose a novel graph-based code search method called EAGCS, which mines latent code semantics by enhancing edge information in APDG and embeds the APDG into graph-level vector via edge attention-based GGNN to boost the semantic expression. In the future, we will strive to optimize the code graph structure and model architecture to improve search performance. We also plan to investigate the influence of the number of nodes in APDG and the length of query statements on the search results. Furthermore, how to excavate potential user search intention and reinforce user query is another rich field worthy to be penetratingly explored.

REFERENCES

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589-1598.

[2] K. Kevic and T. Fritz, "Automatic search term identification for change tasks," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 468-471.

[3] S. Nerur and V. G. Balijepally. "Theoretical reflections on agile development methodologies". Communications of the ACM, 2007, pp. 79-83.

[4] 2022. Github. Retrieved February 14, 2022 from https://github.com.

[5] 2022. Stack Overflow. Retrieved February 14, 2022 from https://stackoverflow.com.

[6] X. Gu, H. Zhang, and S. Kim, "Deep Code Search," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 933-944, doi: 10.1145/3180155.3180167.

[7] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, 2018, pp. 31.

[8] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1-5.

[9] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, "Deep graph matching and searching for semantic code retrieval," in *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2021, pp. 1-21.

[10] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, B. Xiao, W. Dong, and X. Liao, "deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search,"*arXiv preprint arXiv:2103.13020*, 2021.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *ACM Transactions on Programming Languages and Systems (TOPLAS)*. ACM, 1987, pp. 319-349.

[12] S. S. Patil, "Automated Vulnerability Detection in Java Source Code using J-CPG and Graph Neural Network," *University of Twente*, 2021.

[13] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated Graph Sequence Neural Networks," in *Proceedings of ICLR'16*, 2016.

[14] F. Lv, H. Zhang, J. -g. Lou, S. Wang, D. Zhang, and J. Zhao, "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)*, 2015, pp. 260-270, doi: 10.1109/ASE.2015.42.

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd*

International Conference on Software Engineering. ACM, 2011, pp. 111-120.

[16] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*. ACM, 2020, pp. 196-207.

[17] S. Fang, Y. S. Tan, T. Zhang, and Y. Liu, "Self-Attention Networks for Code Search," in *Information and Software Technology*, 2021, 134: 106542.

[18] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13-25.

[19] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, and Y. Liu, "GraphSearchNet: Enhancing GNNs via Capturing Global Dependency for Semantic Code Search," *arXiv preprint arXiv:2111.02671*, 2021.

[20] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon, "Augmenting and structuring user queries to support efficient free-form code search," in *Empirical Software Engineering*, 2018, 23(5), pp. 2622-2654.

[21] L. Xuan , Q. Wang, and Z. Jin, "Description Reinforcement Based Code Search," in *Journal of Software*, 2017.

[22] T. Kenter, A. Borisov, and M. De Rijke. "Siamese cbow: Optimizing word embeddings for sentence representations," *arXiv preprint arXiv:1606.04640*, 2016.

[23] A. Lazaridou, N. T. Pham, and M. Baroni, "Combining language and vision with a multimodal skip-gram model," *arXiv preprint arXiv:1501.02598*, 2015.

[24] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective" in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011, 15(35).

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