CSSAM: Code Search via Attention Matching of Code Semantics and Structures

Bo Cai¹, Yaoxiang Yu¹, and Yi Hu¹
¹School of Cyber Science and Engineering, Wuhan University, Wuhan, China
cai@whu.edu.cn, yu.yaoxiang@whu.edu.cn, csehuyi@whu.edu.cn

Abstract—Code search greatly improves developers’ coding efficiency by retrieving reusable code segments with natural language queries. Despite the continuous efforts in improving both the effectiveness and efficiency of code search, two issues remained unsolved. First, programming languages have inherent strong structural linkages, and feature mining of code as text form would omit the structural information contained inside it. Second, there is a potential semantic relationship between code and query, it is challenging to align code and text across sequences so that vectors are spatially consistent during similarity matching.

To tackle both issues, in this paper, a code search model named CSSAM (Code Semantics and Structures Attention Matching) is proposed. By introducing semantic and structural matching mechanisms, CSSAM effectively extracts and fuses multi-dimensional code features. Specifically, the cross and co-attention layer was developed to facilitate high-latitude spatial alignment of code and query at the token level. By leveraging the residual interaction, a matching module is designed to preserve more code semantics and descriptive features, which enhances the relevance between the code and its corresponding query text. Besides, to improve the model’s comprehension of the code’s inherent structure, a code representation structure named CSRG (Code Semantic Representation Graph) is proposed for jointly representing abstract syntax tree nodes and the data flow of the codes. According to the experimental results on two publicly available datasets containing 475k and 330k code segments, CSSAM significantly outperforms the baselines in terms of achieving the highest SR@1/5/10, MRR, and NDCG@50 on both datasets, respectively. Moreover, the ablation study is conducted to quantitatively measure the impact of each key component of CSSAM on the efficiency and effectiveness of code search, which offers insights into the improvement of advanced code search solutions.

Index Terms—code search, graph representation, feature interaction, attention mechanism, graph attention network

I. INTRODUCTION

With the increasing popularity of open source communities, the amount of code hosted on platforms such as Github and StackOverflow is increasing daily, giving program developers more opportunities to reuse code. In the face of huge code resources, how to accurately find the corresponding code segment according to user intent has become a popular research problem. Code search has gone through two stages, the stages of code search are based on information retrieval (IR) techniques, such as Portfolio [1] proposed by McMillan et al. and CodeHow [2] proposed by Fei et al. These IR-based methods only treat code as a text fragment like natural language, and conduct natural language to code fragment retrieval work using search engine ideas. Nevertheless, it is evident that code snippets and natural language are heterogeneous and have significant differences. Deep learning-based code search methods have proven effective in addressing such issues since their introduction. In 2018, DeepCS [3] proposed by Gu et al. was the first method to use deep learning to solve the code search problem. It embeds natural language and code snippets into a high-dimensional vector space through a neural network, and then uses the cosine distance to judge the similarity between them. Compared with information retrieval based models, deep learning (DL) based models can capture higher dimensional information. Therefore, their findings are also significantly superior to those of IR models.

Like IR models, DL models also have their limitations. Existing research has been able to comprehend and analyze natural language using deep neural networks. However, these models cannot grasp programming languages (e.g., C, Java) effectively since their structural composition and syntactic links are distinct from those of natural languages. To solve this problem, some models make use of tokens, API sequences, and method names to extract semantic information from code segments, and this strategy proved effective. In the code2vec [4] model, an attempt is made to use the abstract syntax tree (AST) as a feature of the code for embedding. Then comes the proposed SBT [5] method for traversing the AST to enhance the model’s comprehension of the tree topology. All of these studies process the code’s AST as sequence features while learning its features using RNN or LSTM models.

Theoretically, graph neural networks (GNN) [6] should be able to embed such topologies as AST more effectively [7], [8]. Wan et al. proposed MMAN [9] to learn the code attribute graphs of AST, control flow graph (CFG), data flow graph (DFG), etc. as features with GNN, which achieved excellent results. However, each graph node represents a basic code block, which is too coarse-grained in comparison to other code representations such as AST or token sequences. As a result, the combination of graph and GNN cannot effectively learn the fine-grained semantic information present in the utterance. Therefore, graph structure data combined with GNN cannot effectively capture high-level semantics based on dependencies between statements.

The success of large-scale natural language models has resulted in the emergence of pre-training-based code search models. These models (e.g., CodeBert [10], CodeT5 [11]) are designed to embed the code and natural language descriptions into the same feature space, then the cosine or L2 similarity...
of these vectors is computed. This type of model cannot retain the deep code structure and semantic information, so the semantic interaction and text alignment operations cannot be adjusted intuitively. Consequently, they were unable to learn the dependencies between variables in the code and the transition process between program states. In addition, even though the fact that the aforementioned models can capture the semantic information of individual code fragments or query texts, it is challenging to investigate the semantic relationships within the code and between the code and the description text at a finer granularity. When training models with the same size code dataset, our fine-grained method gives better results than the pre-training models.

Based on the existing research, this paper proposes a new code search model CSSAM (Code Semantics and Structures Attention Matching), that fully leverages code semantics and structures, and is more effective than the current state-of-the-art model. The following are the contributions of this paper:

- A new deep learning-based code search model CSSAM is proposed\(^1\), which semantically extracts and matches codes and descriptions by introducing multiple levels such as semantic level and structural level matching modules.
- A new code representation structure, the Code Semantic Representation Graph (CSRG) is proposed. It is based on the topology of AST, which reduces the size of tree nodes while preserving the semantic nodes like variable or method name, as well as expands the data flow between leaf nodes through the set of edges in DFG, so that the representation contains more semantic information.
- A bilingual information matching module CrCo(Cross and Co-attention) is proposed and used. Specifically, we develop a matching strategy based on residual interaction between bilingual information, which improves the model’s description ability by cascading residual information and attention mechanisms between code and query texts, preserving more textual features of codes and descriptions. After that, we introduced a weight-sharing mechanism for matching descriptions in the same feature space to improve the correspondence between two different languages.
- The CSSAM proposed in this paper can be accurately trained and tested on two publicly available large datasets, and the experimental results are better than three state-of-the-art models.

The remainder of this paper is organized as follows. Section II introduces the related work of this paper. Section III presents our proposed model CSSAM. Section IV describes the experiment setup. Section V and Section VI show the experimental results and discussion respectively. Section VII concludes the work and presents future works.

\(^1\)We release the source code of our approach in https://github.com/yx-yu/CSSAM

II. RELATED WORK

In this section, we mainly introduce and discuss the related work of this paper from two aspects: code search and code representation based on deep learning.

A. Code Search

Many IR-based code search algorithms [12] have been presented over the years. The earliest research [13] utilized a parameterized matching method to find identical or nearly identical code and query statement pairs. Another research [14] leverages re-categorized code search features and reassigns feature weights according to different semantic categories to improve the retrieval effect. Hill et al. [15] added method name information to the retrieval model in an effort to improve the performance of code search results. Moreover, Lv et al. [2] combined API understanding to improve the code search system based on information retrieval, which has become a common research baseline. Other researchers [16]–[19] focused a great deal of effort on query extension in order to enable the model to match the semantics between user query intent and code snippet functionality.

Semantic similarity research for code search has gained popularity in recent years. With the development of deep learning, Jiang et al. [20] were the first researchers to study the semantic gap between source code and natural language, the matching relationship is generated by the depth model once the feature vector has been obtained. Researchs [21], [22] created a neural network-based IR model by fusing the sorting learning training objective with a deep learning model. Wang et al. [23] provided a summary for adjusting loss functions in neural retrieval models, and gave suggestions for how to make future models more accurate. Wang et al. [24] used reinforcement learning to enhance query comprehension and hence search efficacy.

Since these studies rely on deep learning’s ability to pull out semantic features as a way to improve performance, they don’t take into account the fact that code is not the same as traditional natural language and has structural features that are different from natural language. Therefore, we divided the feature extraction procedure into two parts, a semantic extraction level and a structural extraction level, and then merged the two distinct features. From this point of view, our research is focused on how structural features of code can be used to make code search models work more effectively.

B. Code Representation

In early research, code was treated as a natural language, and it was taken for granted to process code fragments using natural language models [25]. Gu et al.’s research and experiments [3] proved that method names and API sequences in code are also crucial for code feature extraction, and these methods were used in code search and code annotation generation. To determine whether the AST data structure represents the code more accurately, the distributed code2vec model for learning code using AST paths was proposed, which was used in the method name prediction of code fragments.

403

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
with very good results. Furthermore, subsequent research [26] has demonstrated that the AST contains a significant amount of code semantic information.

Since code2vec was proposed, people have started to study the topological representation of code. Code2vec uses distributed representation of words to learn the vector representation of tokens in code by traversing the AST [27], [28] paths. The proposed SBT by Hu et al. [5] gives a rule for going through the AST and changing the topological structure into a sequence structure. Wan et al.’s proposed MMAN [9] converts the topological structure of code into a sequence structure. MMAN treats the tree structure of the code (ASTs) representation [29], [30] as a special graph structure and uses graph neural networks to learn the embedding of codes.

Understanding the semantics of code through words alone often fails to understand the deeper semantic information expressed in code. As an intermediate representation of high-level language and machine instructions, AST contains the logical information of the code. Recent research like SBT, Tree-LSTM [31], etc., all serialize the AST into sequences,
but transforming the topology into a serial structure inevitably loses a lot of information. Typically, different code representations such as DFG result in more information about various structure spaces, we propose a code semantic representation graph based on the AST and DFG as a way to better connect different code-structured representations.

III. PROPOSED APPROACH

This section first introduces the whole workflow of code search, and then presents the details of our proposed model architecture for CSSAM.

Figure 1 shows the workflow of code search [3], [9], [32], and our model also follows this process. The code search can be divided into two parts: offline training and online search, where the offline training phase requires us to extract a large number of <code, query> pairs from the open source code repository as training data, and then train our model CSSAM with the training data. After the training is completed, we get a trained search model. Another phase is the online search phase, in which we only need to enter a natural language description, and the trained model will search for the closest code fragment.

The overall structure of the CSSAM model is shown in Figure 2. To allow the model to concentrate on extracting structural and semantic data from the code, respectively, our structure can be divided into two parts: semantic level matching and structural level matching. In the text semantic level matching module, we designed a matching module based on residual and interaction methods in order for the model to understand the association between query and code and align them in the same feature space. The purpose of this module is to generate the co-interaction matrix using the features in the two sequences to establish the fine-grained relationship between them. We created a new code representation structure of CSRG for structural feature extraction in the structure level matching module. We didn’t just add structural and semantic features together to figure out how similar something was. Instead, we made the fusion attention machine layer to make sure that each part had an equal impact.

A. Code Semantic Representation Graph

For snippets with a very large amount of code, there is the problem of long coding spans when data or variables are called repeatedly. For example, a function customizes a new variable in the first several lines but does not call it again until the code’s last line of the function; consequently, when tokens features are extracted by models such as GNN or RNN, the long-term relationship of that variable in the code text would disappear due to the increase in sequence length. Hence, the data dependency of each statement can be determined by analyzing the data stream that the statement uses.

To minimize information loss and maintain the original AST structure, this paper proposes CSRG, a graph structure based on AST that aggregates node information. CSRG is based on the AST but more compact, and keeps the data flow information in the CSRG structure. This representation model was designed to incorporate the acquisition of contextual information about code statements. By adding directed edges to variables in the context, the model could eliminate the problem of disappearing code semantics.

Fig. 3. Example of code semantic representation graph. In CSRG, there are two different kinds of edges: one comes from AST and the other from DFG.

Figure 3 shows the generation process of the code semantic representation graph. Third-party tools\(^2\) are used to generate the AST and DFG of the code segments.

We first generate the DFG abstract representation of the code using the tree-sitter tool, and then use depth-first search algorithm to locate the corresponding DFG node in the AST. We can notice that there are many nodes in the AST and DFG. Nodes represent variables such as \(a_{\{i\}}, b_{\{i\}}\) and \(x_{\{i\}}\), where \(i\) indicates its position in the sequence of tokens of the code. Nodes in the DFG can be located in the AST. To obtain CSRG, we first de-duplicate the nodes with the same attributes, ignoring the indicates. Then we get the AST and DFG with a reduced number of nodes. Secondly, find the corresponding DFG node in the AST, and add the node relationship from the DFG to the AST. The improved AST now contains two types of edges: one that is present in the AST itself and another that is added based on the DFG. We can assign different weights to these two types of edges. As a result of the rule for generating three-address code(3-AC) by the compiler, variables and attributes are typically located at the leaf node positions in the AST. After combining the data flow in this structure, directed edges are added between each leaf node to determine the relationship between branch statements in the AST, which further reduces the sequence span during feature extraction. Lastly, the strings in the nodes are changed based on the source code’s syntax parsing dictionary in order to get the full semantic flow graph of the code. And now we have obtained the code semantic representation graph, which has a smaller number of nodes compared to the AST, and also has a data flow feature. There is still a strong connection between the leaf nodes of diverse subtrees. The flow of the CSRG generation algorithm is depicted as Algorithm 1.

B. Tokens and Nodes Embedding

Deep learning model can not properly comprehend the <code, query > pair as original input. Code tokens and

\(^2\)https://tree-sitter.github.io/tree-sitter/
Algorithm 1: CSRG Generation Algorithm

**Input:** input code segment $C$

**Output:** generated CSRG structure information $G_{c}(V, E)$

1. Use compiler tool to generate AST structure information $G_{a}(V, E)$ and DFG structure information $G_{d}(V, E)$
2. Node de-duplication for $G_{a}(V, E)$ and $G_{d}(V, E)$
3. for each $v_{i} \in G_{a}$ do
4.   for each $v_{j} \in G_{d}$ do
5.     if $v_{i} == v_{j}$ then
6.       Add $E_{ij} \in G_{d}$ in $G_{a}$
7.     end if
8.   end for
9. end for
10. return $G_{a}$

query words need to be tokenized and fed into the embedding model for vector training respectively. To embed the code tokens and query vocabulary, we use the unsupervised learning fasttext [33] model and various feature enhancement methods, including n-gram and subword regularization [34], so original inputs can be applied to this component to transform them into vectors set $A = \{a_{1}, a_{2}, \ldots, a_{t}\}$ and $B = \{b_{1}, b_{2}, \ldots, b_{q}\}$ with the same dimension, where $a_{i}, b_{j} \in \mathbb{R}^{k}$ is the embedding vector with $k$ dimensions that corresponds to the i-th token or word in the $<code; query>$ pair.

For the CSRG structure representation of the code, our model needs to learn the structural and semantic information between the local and global within each node, so as to extract the features embedded in the different structures in multiple structural representations. In order to make each token node able to obtain the local relationships between the internal nodes, we design the graph embedding layer by using the DeepWalk [35] method, and DeepWalk is divided into two parts: random walk and generation of representation vectors. Firstly, some vertex sequences are extracted from the graph using the Randomwalk algorithm; then, with the help of natural language processing ideas, the generated fixed-point sequences are regarded as sentences composed of words; all the sequences can be regarded as a large corpus, and finally, each vertex is represented as a vector of embedding dimension $d$ using the natural language processing tool word2vec [36], [37]. After the graph embedding procedure, the CSRG of the model input is transformed from $G_{c}(V, E)$ to the feature matrix $\Phi \in \mathbb{R}^{V \times d}$, $V$ represents the number of nodes in the CSRG.

C. Query Features Representation

Current feature extraction models have achieved very good results in the field of natural language processing for natural language descriptions. Since the length of code description statements is generally short, for the extraction of semantic characteristics from descriptions, our model employs Bi-LSTM [38] as a context embedding layer. We construct each fasttext-generated word embedding $b_{i} \in B$ into a matrix $X_{B} \in \mathbb{R}^{k \times q}$ by taking it as an input. Then, we embed them as follows using Bi-LSTM model, then sum the features output at the hidden layer for each time step of the forward and backward LSTMs. Finally, we apply a mean pooling layer to generate a high-level embedding $h_{\text{lstm}} \in \mathbb{R}^{1 \times T}$ for query context $A$, where $h_{i}$ is the output vector of the hidden layer in each time step, and $T$ is the embedding dimension of the LSTM cell.

$$h_{\text{lstm}} = \frac{1}{q} \sum_{t=1}^{q} h_{t}$$

D. Graph Representation

As a graph structure, using GNN to encode CSRG can dig deeper into its node features and patterns. GCN(Graph Convolutional Network) [39] can perform convolutional operations on graphs. However, GCN has some drawbacks: it relies on the Laplacian matrix and cannot be used directly on directed graphs; model training relies on the whole graph structure and cannot be used on dynamic graphs; different weights cannot be assigned to neighboring nodes during convolution processing. Therefore, Graph Attention Network (GAT) [40] is proposed to solve the problems of GCN, and we use GAT to extract CSRG features of the code.

Suppose graph $G_{c}$ contains $V$ nodes, after the graph embedding procedure, each node has a feature vector of $h_{i}$ and embedding size $d$, denoted as $h = \{h_{1}, h_{2}, \ldots, h_{V}\}$. If node $j$ is a neighbor of node $i$, the importance of node $j$ to node $i$ can be calculated using the Attention mechanism. The specific attention approach of GAT is as follows: the feature vectors $h_{i}, h_{j}$ of nodes $i, j$ are first dimensionally expanded by sharing parameters $W$. Then, node $i$ and $j$ are stitched together after the parameter transformation using the concat operation. Finally, the attention mechanism is used to map the spliced high-dimensional features to real numbers. The attention factor $e_{ij}$ calculation formula is as follows:

$$e_{ij} = (\tilde{a}^{T} ([Wh_{i}; Wh_{j}]))$$

where $\tilde{a}^{T}$ is a trainable parameter vector and $W \in \mathbb{R}^{T \times d}$ is a trainable parameter matrix, $[;]$ denotes the concat operation.

The activation function uses $\text{LeakyReLU}$ with the following equation:

$$\alpha_{ij} = \frac{\exp (\text{LeakyReLU} (e_{ij}))}{\sum_{k \in N_{i}} \exp (\text{LeakyReLU} (e_{ik}))}$$

where $N_{i}$ are all neighbor nodes of node $i$.

Based on the calculated attention coefficients, the feature vector of node $i$ after attention is as follows:

$$h'_{i} = \sigma \left( \sum_{j \in N_{i}} \alpha_{ij} Wh_{j} \right)$$

$\sigma(\cdot)$ represents a nonlinear activation function, and $h'_{i}$ is the new feature added by GAT for each vertex (incorporating neighborhood information). Finally, we can obtain the graph matrix $H \in \mathbb{R}^{V \times T}$ by GAT model. The output of our module should be a one-dimensional vector of length $T$, so we must pool the graph matrix. There are many graph pooling methods, including average pooling, max pooling, and so on.
techniques, including weighted average and GraphSAGE. In this case, we use the technique of adding up all the node features and averaging graph matrix \( H \) to obtain a vector 
\[
h^{CSRG} = \frac{1}{V} \sum_{i=1}^{V} h_i
\]  
(5)

E. Cross and Co-attention Block

Since the lexical and syntactic structures of code and natural language have semantic gaps, it is a great challenge to align code and text across sequences to keep the vectors spatially consistent. Our task is essentially to create a kind of matching from code snippet to corresponding description, that is, to find the matching relationship between code snippet and description. At the lexical level, we can consider code and query as two different languages. In order to better match two kinds of texts just from words and sentences, this paper proposes a matching module based on residuals and interactions, called the CrCo (Cross and Co-attention) Block. It uses a bidirectional version of residual connections to cross consecutive query and code features. The sequence encoder first computes the contextual features of the query text and code text sequences within the CrCo block. The block’s inputs and outputs are concatenated by using an enhanced residual concatenation, then fed to the co-attention layer, which uses matrix multiplication and attention mechanisms to model the alignment and interaction of the two sequences. A fusion layer unites the co-attention layer’s inputs and outputs. Finally, the output of the code and query are then sent to the pooling layer, converted to a vector of fixed length, and used as input for similarity matching, implemented as shown in Figure 4.

Specifically, for the embedding layer, we linearly transform the tokens vectors set \( A \) and the docstring vectors set \( B \) using a projection matrix \( W \in \mathbb{R}^{T \times K} \), and then we use the average of the embedding vectors as the final vector \( c_0, q_0 \in \mathbb{R}^{1 \times T} \).

![Fig. 4. Details of a CrCo block](image)

For the Cross layer, we take the processing of the code on the left side of the module as an example. The \( n \)'s block output is denoted as \( o_n \) and the operation \((W_{c1} \odot ( h^{CSRG}_1 + b_c)) + o_n\) is denoted as \( \odot \). \( W_c \in \mathbb{R}^{T \times T} \) and \( b_c \in \mathbb{R}^T \) are the learned weight matrix and bias vector, respectively. \( c_1 \) contains the base information for the original features. On the other hand, it facilitates the subsequent matching with the addition of residual information.

The Co-attention layer, based on the attention mechanism, takes features from the two vectors as input and computes the aligned representations as output. Input from the first sequence of length \( c_0 \) is denoted as \( c_1 = (c^1; c^2; \ldots c^T) \) and input from the second sequence of length \( q_1 \) is denoted as \( q_1 = (q^1; q^2; \ldots q^T) \). The similarity score \( e_{ij} \) between \( c^i \) and \( q^j \) is computed as the dot product of the projected vectors:
\[
e_{ij} = F(c^i)^T F(q^j)
\]  
(6)

\( F \) is an identity function or a single-layer feedforward network. The choice is treated as a hyperparameter. The similarity score \( e_{ij} \) is used to compute the output vectors \( a' \) and \( b' \):
\[
a'_i = \frac{\sum_{j=1}^{l_k} \exp(e_{ij}) b_j}{\sum_{k=1}^{l_k} \exp(e_{ik})}
\]
\[
b'_j = \frac{\sum_{i=1}^{l_k} \exp(e_{ij}) a_i}{\sum_{k=1}^{l_k} \exp(e_{kj})}
\]  
(7)

(8)

For the Fusion layer, we changed the calculation process’s input type to \( \odot \) as \( c_2 = c_1 \odot (W_f a' + b_f) + a' \), where \( W_f \in \mathbb{R}^{T \times T} \) and \( b_f \in \mathbb{R}^T \) are the learned weight matrix and bias vector. Finally, we assign the value of the vector \( c_2 \) to the output of the \( n \)-th block \( o_n \). After the calculation of this module, the code and the description get a vector output \( h^{tok}, h^{doc} \in \mathbb{R}^{1 \times T} \) respectively.

F. Attention Fusion

By introducing the above modules, for the code we generate two vectors \( h^{tok} \) and \( h^{CSRG} \), and for their corresponding descriptions we generate two vectors, \( h'^{lstm} \) and \( h^{doc} \), and we need to merge these corresponding vectors into one vector, and here we use the attention mechanism. Take the example of merging code related feature vectors, in the fusion layer, the importance of each feature (e.g. \( h^{tok} \) and \( h^{CSRG} \)) to the final matching result is different. We need to figure out the attention score for each vector and then use weighting to get the final vector.
\[
\alpha^i = \frac{\exp(g(f(h^i), u))}{\sum_j \exp(g(f(h^j), u))}, i \in \{tok, CSRG\}
\]  
(9)

where \( f(\cdot) \) is a feedforward neural network, \( g(\cdot) \) denotes the dot product operation, and \( u \) denotes another feature of the code.

Similarly, for fuse query related features, we calculate the attention score of each vector separately according to the above algorithm. Now that we have the vectors from the different modules and the attention scores for each vector, we can get the code for each vector and describe it.

\[
\text{CodeTokens}_{CrCo} = \alpha^{tok} h^{tok}
\]  
(10)

\[
\text{CSRG}_{GAT} = \alpha^{CSRG} h^{CSRG}
\]  
(11)

\[
\text{Docs}_{lstm} = \alpha^{lstm} h^{lstm}
\]  
(12)
\[ \text{Docs}_{\text{CrCo}} = \alpha \cdot \text{h}_{\text{doc}} \]  
\[ x_{\text{code}} = [\text{CodeTokens}_{\text{CrCo}}, \text{CSRG}_{\text{GAT}}] \]  
\[ x_{\text{docs}} = [\text{DocsTokens}_{\text{LSTM}}, \text{DocsTokens}_{\text{CrCo}}] \]

**G. Loss Function**

We have described above how to obtain the vectors corresponding to the code and the textual description. And we already know that semantically more similar statements than embedding them into the same high-dimensional vector space, the corresponding vector distances obtained are also more similar. In other words, given a code fragment \( x \) and a descriptive statement \( d \), if they are corresponding, then the vector similarity of our model embedding has the highest degree of similarity, and if they are not, then the vectors embedded in the model should have the smallest degree of similarity. In the training phase, we use the triple \( <x, d^+, d^-> \) to train the model, where \( d^+ \) denotes the description corresponding to the code segment \( x \), and \( d^- \) denotes the description not related to the code segment, and we want the model to maximize the similarity between \( <x, d^+> \) and minimize the similarity between \( <x, d^-> \). In summary, the model’s loss function is shown as follows:

\[ \mathcal{L}(\theta) = \sum_{<x, d^+, d^-> \in \mathcal{D}} \max(0, \beta - \text{sim}(x, d^+) + \text{sim}(x, d^-)) \]  

where \( \theta \) denotes the model parameters, \( \mathcal{D} \) denotes the training data set, \( \beta \) is a hyperparameter, \( \text{sim} \) denotes the similarity score between two vectors, \( x, d^+, d^- \) denotes the code fragment \( x \), the description segment \( d^+ \) corresponding to the code \( x \), and description statements \( d^- \) not corresponding to code fragments with the same dimension after model embedding.

**H. Online Code Search**

Given a set of code segments to be searched, for an input query \( q \), the model needs to sort the similarity of all code segments in the database and select the set \( x_1, x_2, ..., x_k \) of \( k \) code segments that are closest to the query \( q \). For the input query \( q \), the similarity between \( q \) and \( x \) is calculated by cosine similarity for each code segment \( x \) in the set of code snippets as follows:

\[ \text{sim}(x, q) = \cos(x_{\text{code}}, x_{\text{docs}}) = \frac{x_{\text{code}} \cdot x_{\text{docs}}}{\|x_{\text{code}}\| \cdot \|x_{\text{docs}}\|} \]  

where \( x_{\text{code}} \) and \( x_{\text{docs}} \) denote the vectors of code segments and query statements, respectively. The larger the value of similarity, the higher the relevance of the corresponding code segment and query statement.

**IV. EXPERIMENTS**

**A. General Settings**

To train our proposed model, we first randomize the training data and set the mini-batch size to 32. For each batch, the code is padded with a special token \( \text{PAD} \) to the maximum length. All tokens in our dataset are converted to lower case. We set the word embedding size to 300. For LSTM unit, we set the hidden size to be 256. For the number of

| Dataset | Hu’s dataset | Husain’s dataset |
|---------|-------------|-----------------|
| Train   | 475812      | 329967          |
| Test    | 10000       | 19015           |
| Avg. tokens in comment | 10.25 | 10.2 |
| Avg. tokens in code | 58.6 | 68.5 |
| Max tokens in comment | 32 | 440 |
| Max tokens in code | 841 | 369 |

1https://github.com/xing-hu/EMSE-DeepCom
4https://github.com/github/CodeSearchNet
D. Evaluation Metrics

1) MRR (Mean Reciprocal Rank): This is a commonly used metric to measure the effectiveness of search algorithms. For a query set \( Q \), the set of returned results is \( q \), the correct result appears at \( F_{\text{Rank}} \), and the score is the reciprocal of \( F_{\text{Rank}} \), then \( MRR \) is:

\[
MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{F_{\text{Rank}}_q}
\]

Higher MRR values indicate better performance of the code search model.

2) SuccessRate@k (Success Percentage at k): This metric measures the percentage of the first \( k \) results for which one or more correct results may exist, and is calculated as follows:

\[
\text{SuccessRate@k} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \delta(\text{F Rank}_q \leq k)
\]

where \( \delta \) is a function, the output is 1 when the input is true, otherwise the output should be 0. A good code search engine should place the correct results as close to the front of the result list as possible, so that users can easily find the results they need more quickly, and similarly, the higher the R@k value, the better the performance of the code search model.

3) Normalized Discounted Cumulative Gain (NDCG): The normalized discounted cumulative gain is used as an evaluation metric for the sorting results to evaluate the accuracy of the sorting. Recommender systems usually return a list for a user, and assuming the list length is \( K \), the difference between this sorted list and the user’s real interaction list can be evaluated with \( NDCG@K \).

\[
NDCG@k = \frac{2^{r(i)} - 1}{\log_2(i + 1)}
\]

where \( r(i) \) is the score of the ith result. In the code search task, only correct or incorrect, the corresponding scores are 1 and 0. In our experiments, \( NDCG@50 \) is taken as the evaluation index considering the dataset size.

E. Implementation Details

1) Tokenization: As a natural language, it is enough to separate words according to their spacing, but for programming languages, there are elements such as hump nomenclature and a lot of symbolic language. For hump nomenclature, such as "getFileName", we can split it into three words: “get”, “file” and “name”. For the large number of symbols in the programming language, in some papers all the symbols are removed, leaving only the words, but in fact the symbols in the code also contain a lot of semantic information, so in this paper, we keep the symbols in the code syntax.

2) Bias of Code Semantic Representation Graph: To distinguish between the two types of edge information in the code semantic graph. After a variety of weight settings, we found that the weight setting of the edge in CSRG has little effect on the accuracy of the search results. We set the weight of the edge generated by AST to 0.4, the weight of the DFG edges is set to 0.6, which can achieve a relatively better result.

V. RESULTS

To evaluate our proposed approach, in this section, we conduct experiments to answer the following questions:

- RQ1. Does our proposed approach improve the performance of code search when compared with state-of-the-art approaches?
- RQ2. What is the effectiveness and the contribution of each strategy e.g., CrCo layer, CSRG of source code for the final retrieval performance, and what about their combinations?
- RQ3. What’s better about using CSRG than using AST or DFG?
- RQ4. What is the performance of our proposed model when varying the CrCo block number, code CSRG node number?
- RQ5. Is the search result of CSSAM better than state-of-the-art approaches?

In order to answer the first issue, we conducted comparative experiments to evaluate if our model outperforms the state-of-the-art models, including CodeHow, DeepCS, TabCS, and MPCAT. We ask RQ2 in order to evaluate the performance of each module. We ask RQ3 to analyze the performance of our proposed CSR. We ask RQ4 to analyze the sensitivity of our proposed model when varying the CrCo block number, code CSRG node number. We ask RQ5 to verify the searching results of our proposed CSSAM.

A. RQ1: Does our proposed approach improve the performance of code search when compared with state-of-the-art approaches?

We trained and tested CodeHow, DeepCS, TabCS, MPCAT, and our model CSSAM on the same dataset, and the experimental data are shown in Tables 3 and 4. From the data, our model outperforms three deep learning-based models (DeepCS, TabCS, and MPCAT) and one information retrieval-based model CodeHow.
For Hu et al.’s dataset, the results are in Table 3, our model achieves an MRR value of 0.483 and 0.322/0.636/0.705 in SR@1/5/10, CSSAM outperforms CodeHow, DeepCS, TabCS, and MPCAT 80.39%, 70.04%, 4.82%, 2.33% in MRR, respectively. For Husain’s dataset, the results in Table 4, our model achieves an MRR value of 0.394 and 0.259/0.491/0.575 in SR@1/5/10, CSSAM outperforms CodeHow, DeepCS, TabCS, and MPCAT in MRR and SR@k.

The experimental results show that our model performs better than the baseline model.

B. RQ2: What is the effectiveness and the contribution of each strategy e.g., CrCo layer, CSRG of source code for the final retrieval performance, and what about their combinations?

We did five sets of ablation experiments based on the model to verify the effect of each module on the experimental results. Table 2 shows the effect of each module on the experimental results. From the experimental data, we can see that both the semantic level matching module and the structural level matching module have positive effects on the experimental results, and the model performs better after fusing the matching modules of each level than using these modules alone, which also indicates that the complementary effects between the matching layers at different levels outweigh the conflicts between them. The experimental results also demonstrate the positive contribution of fused attention layers to the model effects by adding and removing attention layers.

Table III Experiments on Hu et al.’s Dataset.

| Model      | SR@1 | SR@5 | SR@10 | MRR  | NDCG@50 |
|------------|------|------|-------|------|---------|
| CodeHow    | 0.2544 | 0.3588 | 0.4094 | 0.3676 | 0.3441  |
| DeepCS     | 0.2574 | 0.3897 | 0.5169 | 0.2834 | 0.3690  |
| TabCS      | 0.2997 | 0.4946 | 0.6642 | 0.4721 | 0.4891  |
| MPCAT      | 0.3125 | 0.3744 | 0.6831 | 0.4099 | 0.5492  |
| CSSAM      | 0.3856 | 0.4912 | 0.5754 | 0.3941 | 0.4223  |

There is no doubt that structural information is very useful for code search. The code search MRR is improved after adding AST and DFG information. We all know that ast has more node information than DFG, which also leads to the training time of the AST model being longer than the DFG model. And using CSRG, although the training time is increased compared with baseline, it is less than that of AST. At the same time, MRR is also higher than that of AST and DFG models.

D. RQ4: What is the performance of our proposed model when varying the CrCo block number, code CSRG node number?

In order to explore the robustness of this model, we analyzed two hyperparameters that may influence the effectiveness of the model, which are the numbers of CSRG nodes. Figure 5 shows the effects of each parameter on the model evaluation metrics, and firstly, we can see that our model performs relatively stable when using different hyperparameters, and the robustness of our model can be proven.

In order to explore why CSRG is effective, we conducted performance tests according to different graph structures on Hu et al.’s dataset. The results are shown in Table 5. We can find that using graph structure information can significantly improve the code search results, but it also bring the burden of training. Compared with only using AST, our proposed CSRG significantly reduces the complexity of input data and improves the training speed. Compared with just using DFG, our CSRG significantly improves the accuracy of search results.

Table V Training Details on Hu et al.’s Dataset.

| Model          | Training time | MRR Score |
|----------------|---------------|-----------|
| Base-w/o.Graph | 16.2 hours    | 0.292     |
| Base-w.AST     | 27.1 hours    | 0.337     |
| Base-w.DFG     | 18.9 hours    | 0.304     |
| Base-w.CSRG    | 24.8 hours    | 0.348     |

There is no doubt that structural information is very useful for code search. The code search MRR is improved after adding AST and DFG information. We all know that ast has more node information than DFG, which also leads to the training time of the AST model being longer than the DFG model. And using CSRG, although the training time is increased compared with baseline, it is less than that of AST. At the same time, MRR is also higher than that of AST and DFG models.

Fig. 5. Sensitivity Analysis

We have tried the number of CrCo blocks from 0 to 5. It can be found that with the increase in the number of layers, all the indicators of the model show an upward trend, followed by a significant increase in the model’s consumption. We find that when the number of CrCo blocks is set to 5, the growth of the model indicators has slowed down. From a comprehensive consideration, we take 4 as the optimal parameter.

As the number of CSRG nodes increases, the indicators are in a stable growth trend, but when the number of nodes
exceeds 80, the indicators appear to decline, and finally, when we tried the case of 300 nodes, the indicators are still not as good as the optimal point. For this phenomenon, the reason of occurrence is related to the dataset, when the number of nodes is close to the average of this dataset, the model can be made optimal, too few nodes cannot provide enough useful information, more nodes means more padding, which is equivalent to diluting the information and cannot achieve the optimal performance of the model.

E. RQ5: Is the search result of CSSAM better than state-of-the-art approaches?

Figure 6 shows the first retrieved results of CSSAM and other models, for queries “save string into the file” and “converts the given hex string into a plain string”. We can notice that CSSAM returns the correct code snippet, but DeepCS and TabCS returned the correct results. They just matched the correlation between the code and the description at the token level. The fusion of DFG and AST enriches the semantic and associative information present in the nodes inside the code snippet, and the CrCo block we proposed further aligns the textual features of the code and query semantically, thus allowing our model to learn additional contextual features based on the sequence structure. On the other hand, CSSAM not only matches at the tokens level, but also takes into account the deep semantic information of the code, so the search results are more accurate.

VI. DISCUSSION

A. Strength of CSSAM

We have determined three benefits of CSSAM, which could account for its efficiency in code search: (a) a more thorough illustration of the source code. CSSAM makes use of CSRG, which includes additional data in addition to the encoding of code tokens. (b) CrCo block, a fine-grained matching module whose efficacy has been demonstrated through experiments. Additionally, (c) understanding the intermediate semantic space of description as well as the unified framework source code of heterogeneous representation. Using a unified architecture CSSAM, an end-to-end neural network model, learns the source code to represent the code and description in the intermediate semantic space.

B. Threats to Validity

Our proposed CSSAM may suffer from two threats to validity and limitations. One threat to validity is the evaluation metrics. In practice, there are multiple related code snippets for a query. However, for automated evaluation, each query has a limited number of correct code snippets. During the evaluation, the other results, which are also related, can not be recognized unless humans are involved.

Another threat to validity lies in the extensibility of our proposed approach. Our model has only been trained and tested on Java data sets. For other programming languages, the model results are unknown. At the same time, because CSSAM needs to use CSRG, but this graph structure needs to
[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013.

[38] J. Wang, L. Zhu, T. Dai, and Y. Wang, “Deep memory network with bi-lstm for personalized context-aware citation recommendation,” Neurocomputing, vol. 410, pp. 103–113, 2020.

[39] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017.

[40] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018.

[41] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[42] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, “Two-stage attention-based model for code search with textual and structural features,” in 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021. IEEE, 2021, pp. 342–353. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00039

[43] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation with hybrid lexical and syntactical information,” Empir. Softw. Eng., vol. 25, no. 3, pp. 2179–2217, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09730-9

[44] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” CoRR, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436