2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00021

MulCS: Towards a Unified Deep Representation for Multilingual Code Search

Yingwei Ma, Yue Yu*, Shanshan Li*, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, Xiangke Liao
College of Computer Science
National University of Defense Technology
Changsha, China
{myw, yuyue, shanshanli, jiazhouyang, majun, runlin_xu, dongwei, xkliao}@nudt.edu.cn

***Abstract*—Code search aims to search for relevant code snippets through queries, which has become an essential requirement to assist programmers in software development. With the availability of large and rapidly growing source code repositories covering various languages, multilingual code search can leverage more training data to learn complementary information across languages. Contrastive learning can naturally understand the similarity between functionally equivalent code across different languages by narrowing the distance between objects with the same function while keeping dissimilar objects further apart. Some works exist addressing monolingual code search problems with contrastive learning, however, they mainly exploit every specific programming language's textual semantics or syntactic structures for code representation. Due to the high diversity of different languages in terms of syntax, format, and structure, these methods limit the performance of contrastive learning in multilingual training. To bridge this gap, we propose a unified semantic graph representation approach toward multilingual code search called MulCS. Specifically, we first design a general semantic graph construction strategy across different languages by Intermediate Representation (IR). Furthermore, we introduce the contrastive learning module integrated into a gated graph neural network (GGNN) to enhance *query-multilingual code* matching. The extensive experiments on three representative languages illustrate that our method outperforms state-of-the-art models by 10.7% to 77.5% in terms of MRR on average.**

***Index Terms*—Code search, multi-language, contrastive learning, intermediate representation**

I. Introduction

Code search aims to search for relevant code snippets by performing natural language queries on a large corpus, and many code search methods have been proposed in recent years [1]–[7]. With the availability of immense and rapidly growing source code repositories such as GitHub and Stack Overflow, learning-based code search models [4], [5], [8], [9] have achieved promising results by leveraging large amounts of training data. There are hundreds of programming languages in widespread use [10], each with its complexities and idiosyncrasies in practice [11] (*e.g.*, C is powerful for system kernel development, while JavaScript is preferred for web applications), and each with different data sizes. Therefore, developing multilingual code search models with shared parameters can leverage larger-scale multilingual code data and learn complementary information across languages compared to monolingual models. At the same time, the model can handle different programming languages without training multiple times, which can effectively reduce the cost of model deployment and maintenance.

Contrastive learning has emerged as a new paradigm that learns object representations through comparing pairs or collections of similar and dissimilar items [12], which enforces similar object representations to be closer while dissimilar object representations are further apart. This capability can naturally understand the similarity between functionally equivalent code in multilingual code search task. Inspired by this, we try to explore applying contrastive learning to multilingual code search. There exist some works addressing monolingual code search problem [13]–[15] with contrastive learning, however, they mainly focus on language-specific information, which leads to some difficulties in multilingual scenarios due to textual (*i.e.*, token) and syntactic (*e.g.*, AST) differences between multilingual code. In one hand, textual features are unreliable [16], [17] generated from the informal, noisy information latent in variable names, comments or coding styles, and also cannot represent the structural logic of source code. In the other hand, code implementations of the same functionality may be quite different, especially for cross-language (*e.g.*, both Java and Python have *if-related* flow control statements, but in Python, *else-if* is written as *elif*, and there is no *switch* statement in Python). It is much less likely to build a unified and effective represent model by only using lexical and syntactic analysis (*e.g.*, AST-based approaches [18], [19]), due to the high diversity of programming languages in terms of the grammar, format and structure.

The aforementioned issues inspire us to find a new way to bridge the code representation gap across multiple programming languages. Semantic information, representing the exact computational meaning to the execution platform, can eliminate differences across languages at a more fundamental level. Semantic information can be represented by data- and control-flow accurately extracted from the *intermediate representation* (IR). Thus, the IR of source code enables the unification of semantically equivalent language constructs, and it is promising to eliminate the enormous differences brought by code implementation, especially for cross-languages.

To derive a unified semantic graph representation based on

*Corresponding author

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00021

IR, however, several challenges need to be addressed. First, generating IR requires compilable code. It is hard to compile the code base automatically, since different code snipptes require various dependencies. Second, for source code with the same functionality written in different languages, there may exist non-negligible differences in the form of initial IRs directly transformed by different compilers (*e.g.*, The IR of Python code use the stack-based structure rather than the three-address structure). Third, how to utilize a unified representation to effectively capture the semantic relations of multilingual code snippets during model training is undoubtedly essential, while it has not been well studied by previous work. Therefore, some general rules need to be applied to make the semantic graph representation with the same functionality similar across different languages.

In this paper, we propose a contrastive learning framework towards *Multilingual Code Search*, called MulCS, which trains neural networks from a large number of multilingual query-code snippets. Firstly, we obtain intermediate representation in multiple programming languages based on the corresponding IR tools. To improve the availability, we design a snippet wrapper by simulating the presence of missing surrounding code to make the incomplete source code without third-party libraries compilable. We further design a general semantic graph construction strategy to unify the intermediate representation across different languages. Our strategy presents a general solution for multi-language to mitigate the differences brought by syntactical structures and grammars of different IRs. Finally, we integrate the contrastive learning into the improved gated graph neural network to model our semantic graphs, which significantly enhances the mutual reinforcement among multilingual code corpus.

We evaluate the effectiveness of our proposed method on the large-scale dataset of three representative programming languages (*i.e.*, C, Java and Python). We argue that the semantic knowledge emphasized in this paper show a promising prospect for deep code representation, especially for further combining with large-scale pre-trained models (*e.g.*, expanding token or AST with IR merging into CodeBERT [20] and SynCoBERT [21]) during multilingual training [22]. In summary, our key contributions are:

- We propose a contrastive learning framework to perform code retrieval tasks in multiple programming languages scenarios. We integrate contrastive learning into the improved gated graph neural network, where the GGNN model can take advantage of graph-structured inputs, and the contrastive learning module can enhance differentiation among multilingual samples. All data and source code can be found in our repository. [1]
- We exploit semantic information to unify multilingual code representation. We design a general semantic graph construction approach based on the IRs transformed by different compilers, which can eliminate the enormous differences brought by the programming language. An effective snippet wrapper has been designed accordingly to make incomplete code fragments compilable.
- We conducted extensive experiments to evaluate our approach on a large-scale multilingual dataset, compared to the typical code search models. The results show that MulCS can achieve significant improvements from 10.7% to 77.5% of MRR on average over the corresponding baseline models. Furthermore, when our method is combined with CodeBERT and CodeT5 pre-trained models, the two pre-trained models improve by 21.1% and 18.3%.

[1]https://github.com/yingweima2022/MulCS

II. Unified Semantic Graph Construction

In this section, we first describe a motivating example to understand semantic graphs and then introduce the common and unique features of IRs in C, Java, and Python. Finally, we present the principles and rules of semantic graph construction.

A. A Motivating Example

About multi-language code search, we need a unified model to eliminate differences between languages. Intermediate representation (IR) is a natural abstraction of code functionality because it represents code semantics through control flow and data flow between operands, opcode and labels. Currently, most languages have their intermediate representation. For example, in Figure 1, we list C, Java and Python code snippets of *"count the number of positive integers"* along with their ASTs and IRs.

Tokens focus on the textual semantics of code, while the textual differences between languages are enormous, even though they are functionally identical. For example, as shown in Figure 1, C, Python and Java use different identifiers to represent arrays (C token *a*, Python token *arr* and Java token *number*), which may be treated as different features during training. Another reason is that the effectiveness of the token-based approach is highly dependent on the quality of the code, such as naming conventions. Oversimplified variable naming can hinder the effectiveness of training models.

ASTs focus not only on textual semantics but also syntactic structures of the code. ASTs and tokens have the same problem that different syntactic structures can achieve the same functionality within a language or across other languages. As shown in Figure 1, C, Python and Java use different ways to traverse arrays, which results in a completely different syntax structure in AST, even though they are semantically identical.

On the other hand, IRs show some similarities across different languages. IR consists of instructions, and one or more instructions compose labels, which can be seen as basic blocks. In these IRs, we label the corresponding opcode, operands, and labels with the same color. We find that all the IRs have similar data flow and control flow.

For data flow, the same operands have similar operations. For instance, each of these IRs has an instruction that makes the variable self-increment by one (the yellow dashed box in the last label in Figure 1). For control flow, labels (basic blocks) have similar jump addresses. For example, in Figure 1, the green label (basic block surrounded by a solid green box)

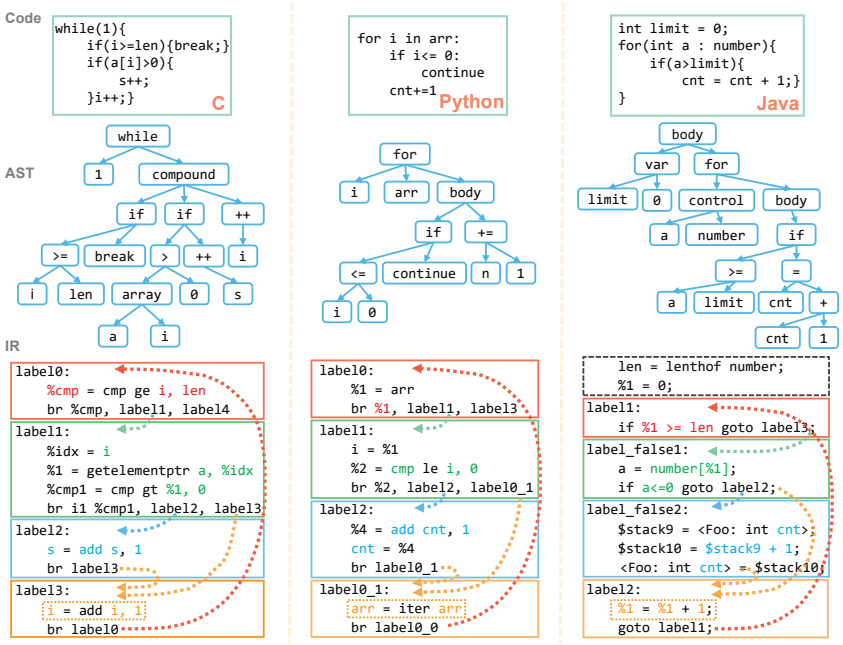

Fig. 1: Code snippets, ASTs and IRs of C, Java and Python

always has an instruction that jumps to the blue label and the yellow label. Therefore, we believe that IR will be a better candidate for semantic sources to unify different languages.

B. IRs Among Different Languages

Although current IR provides the abstraction for code representation regardless of specific grammar, there remain differences between IRs among different languages. In this section, we shed light on the common and unique features of IRs of C, Java and Python. We study these features and find opportunities for extracting data and control flow among different IRs. Accordingly, we set up a set of unified rules for semantic graph construction. As a result, code snippets with similar semantics (similar data and control flow) can be expressed as similar graphs.

We choose C, Java and Python which are three typical programming languages and represent structural, object-oriented and just-in-time languages. Moreover, their IRs have been widely used for over 20 years. IRs discussed in this section are from *LLVM* [2] (C), *Soot* [3] (Java) and *Dis* [4] (Python).

1) Common Features: According to the function of instruction, most instructions can be classified into data- and control-related instructions. Data-related instructions usually involve data operations. For example, all IRs include binary instruction (*add*, *sub*) and assignment instruction, etc. These data-related instructions can construct the data flow inside the IR.

Control-related instructions usually control the execution order of instructions. Specifically, control-related instructions decide which label the instruction will jump to. These control-related instructions can construct the control flow inside IR.

Some other instructions are weak semantic-related. For example, the monitor instructions (exception handling instruction, etc.) are used to monitor the data status. We ignore these kinds of instructions, including *alloca*, *atomicrmw*, etc.

2) Unique Features: There remain unique features among C, Java and Python IRs. First, the grammar of IRs might be different. For example, IR of Python is stack-based bytecode, the flow-related instruction in bytecode is stack-related operation. In these instructions, operands, labels are hidden, so it is inconvenient to observe the data and control flow. On the other side, C and Java use a three-address structure, in which we can easily obtain data and control flow from opcode, operands, and labels. Second, due to the features of languages, the naming of opcode in different IRs may be inconsistent, but the functionality is similar. For example, some data-related instructions, such as adding two numbers, are represented as *BINARY_ADD* in python, *add* in c, and + in java.

In conclusion, while these features have little influence on data and control flow, some transformations (*e.g.*, syntactic unification) are necessary to eliminate these differences.

C. Semantic Graph Construction

Semantic graph construction is inspired by DeGraphCS [23], we set up a set of unified rules for different languages in Figure 2. In a semantic graph, its nodes represent the operands, opcode and labels in IR. Its edges represent the data dependencies and control dependencies, where data dependencies are from data-related instructions, and control dependencies are from control-related instructions.

[2]https://github.com/llvm/llvm-project
[3]https://github.com/soot-oss/soot
[4]https://pypi.org/project/dis/

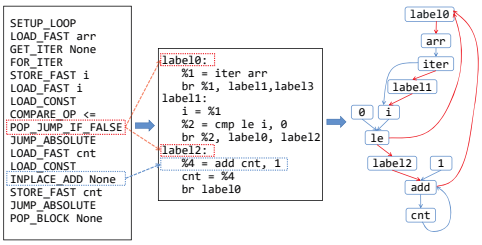

Fig. 2: Example of semantic graph construction

The principles used to construct semantic graphs are as follows. First, we need to unify IR's grammar since their structures might be different. Although most IRs use a three-address structure, some IRs like Python use a stack-based structure which is hard to extract data and control flow directly. Thus, we need to unify IR's grammar according to Algorithm 1 at first. Second, for each instruction, we build nodes and edges in the semantic graph according to the rules in Table I.

Algorithm 1 transform bytecode into three-address structure

Input: python bytecode
Output: three-address data structure

for inst in instruction of bytecode **do**
  if inst is data related instruction **then**
    if inst is stack operation on data **then**
      push/pop variable.
    else
      pop variable.
      combine variable with opcode.
      generate a temporary variable.
      assign combined variables to temporary variable.
      push temporary variable into stack.
    end if
  else if inst is control related instruction **then**
    create two labels which similar the label in LLVM-IR.
    put label before instructions which control related instruction point to.
    transform into 'br' which is similar to LLVM-IR.
  else
    abandon related instructions.
  end if
end for

1) Grammar Unification: We transform all the bytecode IRs into a three-address structure based on algorithm 1. Generally, we first determine whether the instruction is data- or control-related. If the instruction is stack operation, we pop variable from stack or push variable into the stack. If the instruction is operation on data, we should transform the operation and related data into a three-address-formed statement. For example, the *INPLACE_ADD* operation in Figure 2 adds the variable *cnt* with a constant and keeps the sum into the stack. Thus we introduce a temporary variable to keep the sum, and transform the instruction into *%4 = add cnt, constant*. If the instruction is control-related, we first find the instruction which the current instruction has a dependency and add a control label at the front of the instructions. As shown in Figure 2, the instruction *POP_JUMP_IF_FALSE* determines the next instruction that needs to be executed, we find the jumped instruction is *LOAD_FAST arr* or *LOAD_FAST cnt*, thus we add a label *label0* and a label *label2* at the front of the

TABLE I: Rules for constructing nodes and edges of semantic graph

Opcode	Edge type	Return	Oprands	Nodes&Edges
br	control	▫	%cmp,%label_true,%label_false	%cmp→%label_true,%cmp→%label_false
switch	control	▫	%cond,%label_1,...,%label_n	%cond→%label_1,...,%cond→%label_n
call/invoke	data	▫	%name,%param_1,...,%param_n	%param_1→name,...,%param_n→name,name→return
ret	data	▫	%1	%1→ret
cmp	data	▫	%1,%2	%1→cmp, %2→cmp, cmp→return
neg	data	▫	%1	%1→neg, neg→return
add	data	▫	%1,%2	%1→add, %2→add, add→return
sub	data	▫	%1,%2	%1→sub, %2→sub, sub→return
mul	data	▫	%1,%2	%1→mul, %2→mul, mul→return
div	data	▫	%1,%2	%1→div, %2→div, div→return
rem	data	▫	%1,%2	%1→rem, %2→rem, rem→return
load	data	▫	%1	%1→return
store	data	▫	%1,%2	%1→%2

[1] The explanation of the column name is as follows. Opcode is the name of this instruction. Edge Type determines the instruction is about data or control dependency. Return means whether return value exists. Operand shows the operandsused by the instruction. Nodes&Edges illustrate the rules to build nodes&edges in the semantic graph where "→" represents the edges while operands and return values arenodes.

two instructions. After obtaining the three-address code for all languages, we note that the naming of opcode in different IRs might be inconsistent but similar in function. In order to better unify the multi-language representation, we make the opcode names with the same function close to each other between all languages by some simple mappings. For example, an opcode for a function call is represented as *CALL_FUNCITION* in Python, *call* in C, and *invoke* in Java. After mapping, we use the *call* to represent this opcode.

2) Data and Control Flow Extraction: After the grammar is unified, we have transformed IRs of all languages into three-address IRs, based on which we use all operands, opcode, and labels as nodes of the semantic graph. For example in Figure 2, nodes are values *arr, i, 0, 1, cnt*, opcode *iter, le, add* and labels *label0, label1, label2*. It should be noted that temporary variables in IR will be removed and will not appear after graph construction.

Intuitively, data and control dependency can be obtained in each instruction. For instance, *add* instruction has two arguments *%1* and *%2*, and there are data dependencies like: *%1→add*, *%2→add* and *add→return*. Similarly, we build rules to extract dependencies for other instructions in Table I. According to these rules, we can extract the data- and control-dependency of each instruction and build nodes and edges in the semantic graph. For each IR, we can aggregate these dependencies to obtain the data and control flow. For instance in Figure 2, blue arrows are about data dependencies and red arrows are control dependencies. In *label0*, we can find that value *arr* has data dependency with opcode *iter*. Similarly, in *label1*, *le* has control dependency with *label0* and *label2*.

III. Multi-language Code Search Model

A. Overview

Since intermediate representation (IR) are more consistent across multi-language code representation, we exploit the IR to search code snippets in different programming languages. As shown in Figure 3, the workflow consists of IR generation and unification, semantic graph construction and multi-language code contrastive learning module. IR generation and unification module extract the IR of source code by the corresponding compilers according to different programming languages. The semantic graph construction module extracts the semantic

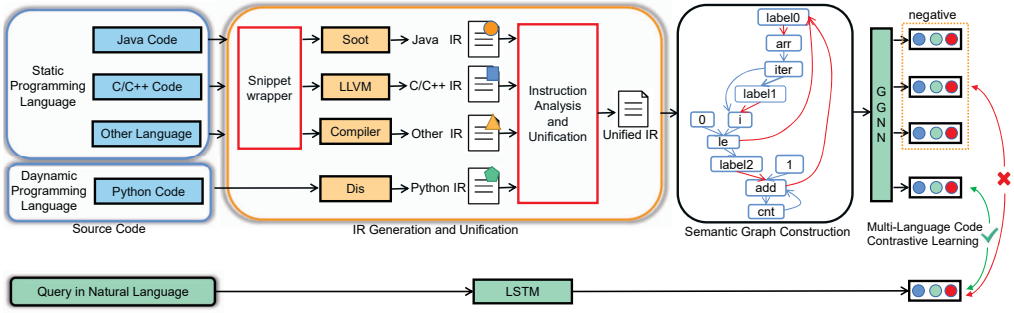

Fig. 3: Workflow of multi-language code search

information from IRs across different languages. The semantic graph and query are then fed into our multi-language code search model to obtain the vector representation. MulCS learns the relationship between the two representation through a multi-language code contrastive learning module. It forces the vector representation of the query and code snippet with the same functionality to be similar and the pair with different functionality to be different.

B. Code Representation

To build the unified semantic graph, we first need to pre-process code snippets to make them compilable and generate the corresponding IRs. Then we follow the principles in § II-C to transform IRs into unified semantic graphs. Finally, we model the semantic graph with GGNN to obtain code representation. The details of implementation are described as follows.

1) IR Generation: We can directly obtain the IR (i.e., bytecode) of Python code snippets by using the *Dis* module since Python is a dynamic language. However, we cannot directly extract IR from incomplete C or Java code snippet because they are static languages and lack the necessary surrounding code or third-party libraries to compile.

The existing work JCoffee [24] can convert the incomplete code snippets to their compilable counterparts by simulating the presence of missing surrounding code and unavailable third-party libraries. However, JCoffee is unable to be directly used in multilingual IR generation for two reasons. First, JCoffee only supports Java, but we have C code in our corpus. Second, JCoffee would modify the code snippet, which is intrusive and might influence our experimental results.

To better make C and Java code snippets compilable and generate corresponding IRs, we design a snippet wrapper that leverages LLVM and Javac feedback to add missing code. The snippet wrapper can be applied to both C and Java code. Moreover, our snippet wrapper is non-intrusive, i.e., it does not modify any code in the original snippet but simulates missing surroundings in the global. Our snippet wrapper mainly handles compilation errors according to the rules in Table II. For instance, when LLVM reports error messages like *"unknown type name"*, our snippet wrapper can recognize these errors caused by an undefined struct

TABLE II: Rules used by our snippet wrapper

Compilation errors	Language	Rules to complement code snippet
Unknown type name	C (LLVM)	Adding the struct definition to the global.
Unknown Variable	C (LLVM)	Adding the variable declaration to the global.
Incompatible type	C (LLVM)	Correct the variable type declaration.
Unknown Variable	Java (Javac)	Adding the variable declaration to class global variables.
Incompatible type	Java (Javac)	Correct class attribute type.

body. For an undefined struct, our snippet wrapper will add the corresponding struct definition information to the global. Similarly, for undefined variables, our snippet wrapper will add the corresponding type declaration information to the global. It should be noted that *"Incompatible type"* errors only happen when the snippet wrapper adds the variable declaration part. However, our declaration might be incorrect and cause conflicts during compilation. In this case, our snippet wrapper will further correct the variable type declaration.

We perform experiments on Java and C datasets. The evaluation metric is the ratio of successfully compiled code snippets among the uncompilable code snippets. Experimental results show that our snippet wrapper improves JCoffee [24] by 9.52% on the Java dataset, with a success rate of 69%. The advantage can be attributed to the non-intrusive characteristic of our snippet wrapper. In concrete, JCoffee deals with *"unknown variable"* and *"incompatible type"* situations by inserting the variable declaration and type casting parts, which might modify the code snippet and introduce some syntax errors because JCoffee can not precisely locate where the surrounding code needs to be inserted. Our snippet wrapper can effectively solve the problem by exploiting the global variable mechanism. The type of the variables can be inferred by the incompatible error messages reported by the compiler. JCoffee can not be directly applied on the C dataset, while our snippet wrapper achieves a success rate of 49%. The remaining 51% of unsolvable situations are mainly caused by the fact that the error messages reported by the LLVM compiler are not always indicated and thus not able to be utilized for our wrapper.

2) Building Semantic Graph: To construct a unified semantic graph, we also consider the following problems. First, how to integrate dependencies in object-oriented languages.

Second, how to decrease the noises in semantic graphs and improve the training efficiency.

Integrating Dependencies in Object-oriented Languages Our semantic graph also supports dependencies remain in object-oriented languages. We represent the relationships in the object-oriented language in two methods. First, we treat statements in the form of *"a.b"*, in which *"a"* is an instance of a particular object, and *"b"* is the attribute of *"a"*. To represent *"b"* belongs to *"a"*, we link *"a"* to *"b"*. Second, we treat statement in the form of *"a.method()"*, in which *"a"* is an instance of a particular object, *"method"* is an interface of *"A"*, and *"A"* is the Class of the object *"a"*. We link *"A"* to *"method"* since the *"method"* is more relevant to Class. Significantly, in an object-oriented language, if a method invocation does not have a return value, the method invocation always changes the status of the object instance. For example, *"a.init()"* is an initial interface to init object *"a"*. Thus, we link the method to object instance to imply the change of object instance.

Decreasing Noises. We adopt some rules to decrease the noises of our semantic graph. In principle, we need to remove the redundant information brought by IR without changing the semantics. We follow the rules of [23] to optimize our semantic graph. First, we remove opcode nodes that are too trivial to reflect the semantics. For example, the *conversion* opcode aims to convert the data type and is meaningless for semantics. Second, we remove temporary variables nodes since these variables are introduced for compilation and do not appear in the source code. After that, we further delete exception handling-related nodes in the initial graph since they are always semantically irrelevant.

3) The Graph Neural Network Exploited in our Method: We exploit an improved gated graph neural network (GGNN) [25] with an attention mechanism to learn the vector representation of source code to model the directed graph with multiple types of edges. In our graph $G = (V, E)$, V represents a set of nodes (v, l_v), and E represents a set of edges $(v_i, v_j, l_{(v_i,v_j)})$. l_v represents the label of node v which is composed of the variables in IR instructions. $l_{(v_i,v_j)}$ represents the type of the edge from v_i to v_j including data dependency and control dependency.

GGNN learns the vector representation of G by message passing mechanism as follows. First, each node $v \in V$ is initialized with a one-hot embedding vector (h_v^0) based on l_v. Then, the embeddings of all nodes are trained by multiple iterations. In iteration t, each node v_i gets message $m_t^{v_j \to v_i}$ from neighbour v_j as: $m_t^{v_j \to v_i} = W_{l_{(v_i,v_j)}} h_{v_j}^{t-1}$. Here, $W_{l_{(v_i,v_j)}}$ represents the weight matrix of the edge type to map neighbour v_j to a shared space. Then, all messages of the neighbours are aggregated to v_i as follows:

$$m_t^i = f_{aggregate}(\{m_t^{v_j \to v_i} \mid v_j \to Neibour(v_i)\}) \quad (1)$$

Then, GGNN uses GRU (Gated Recurrent Unit) [26] to update the embedding of each node v_i. GRU uses the aggregated message and past state $h_{v_i}^{t-1}$ to update the current state as $h_{v_i}^t = GRU(m_t^i, h_{v_i}^{t-1})$. Finally, we exploit the attention mechanism to calculate the importance of each node because different nodes have different contributions to the semantics. We first allocate weights for each node v_i as:

$$\alpha_i = sigmoid(f(h_{v_i}) \cdot u_{vfg}) \quad (2)$$

α_i represents the weight of node v_i, $f(\cdot)$ represents the linear function, $\cdot$ represents the inner project function and u_{vfg} represents the semantic representation of the whole graph. Then, we obtain the embedding of the whole graph h_{vfg} by the following equation:

$$h_{vfg} = \sum_{v_i \in V} (\alpha_i h_{v_i}) \quad (3)$$

C. Query Representation

We apply LSTM [27] to learn the representation of queries. The embedding h_i^{des} of each word in the query is calculated as $h_i^{des} = LSTM(h_{i-1}^{des}, w(d_i))$, where $i = 1, ..., |d|$, $|d|$ represents the length of the query description, and w represents the word embedding layer to embed each word into a vector. We also exploit an attention mechanism [28] to capture the fine-grained relevance between the hidden states and the final query representation. In concrete, we apply an attention layer to calculate the attention score $\alpha^{des}(i)$:

$$\alpha^{des}(i) = \frac{exp(f(h_i^{des}) \cdot u^{des})}{\sum_{k=1}^{n} exp(f(h_k^{des}) \cdot u^{des})} \quad (4)$$

where $\cdot$ denotes the inner project of h_i^{des} and u^{des}, $f(\cdot)$ denotes a linear layer and u^{des} denotes the context vector of the whole query. The context vector is randomly initialized and jointly learned during training. Then, the final representation of the query description $E_{|d|}^{des}$ can be calculated as:

$$E_{|d|}^{des} = \sum_{i=1}^{|d|} \alpha^{des}(i) h_i^{des} \quad (5)$$

D. Multi-Language Code Contrastive Learning

How to effectively capture snippet-level semantic information between different program languages during training is certainly essential for multilingual code search models. However, it has not been well studied by previous work.

After obtaining code representation (C) and query representation (Q), we introduce a multi-language code contrastive learning module to capture the relationship between the multi-language code and query representation. Contrastive learning enforces similar object representation to be closer while dissimilar object representation are further apart. Actually, the queries corresponding to programs with the same semantics written in different languages are often similar. Conversely, the different semantics programs written in other languages often correspond to different queries. Many semantically similar programs in our multilingual corpus, so the semantic relationship between multilingual code can be effectively bridged through their corresponding similar queries. At the same time, how to effectively use multilingual data to capture the relationship between programs with different semantics is

also crucial. Motivated by this, we propose a Multi-Language Negative Sample Augmentation module (MNA) in this work, as shown in Figure 3. Specifically, we randomly select N samples of different languages as a mini-batch from large-scale code snippets. For a given positive pair, the other N-1 code snippets of different languages are treated as negative samples and forced away by the models. At training time, MNA can better close the semantic relationship between different languages, making queries and multi-language code with the same semantics closer in the representation space, while those with different semantics are further apart.

Similar to [29], we use InfoNCE [30] as our contrastive loss, which computes the probability of selecting the positive (query and the associated correct code) by taking the softmax of projected embedding similarities across a batch of multi-language negatives examples obtained by MNA. Eq. (6) shows InfoNCE is a function whose value is low when the query q is similar to the matching code embedding c^+ but not similar to the unmatching code embedding c^-. The temperature hyperparameter [31] is indicated as t.

$$\mathcal{L}_{q,c^+,c^-} = -log\frac{exp(q \cdot c^+/t)}{exp(q \cdot c^+/t) + \sum_{c^-} exp(q \cdot c^-/t)} \quad (6)$$

We use the dot product to measure the similarity between two normalized vector representation (i.e., cosine similarity). Intuitively, our multi-language code contrastive learning module encourages the similarity between a query and its correct code snippet to be larger and the similarity between a query snippet and the incorrect code across multi-language to be smaller.

IV. EXPERIMENT

On the effectiveness of our multi-language code search model, we perform several experiments to explore the following research questions:

RQ1: Does our proposed approach improve code search performance than state-of-the-art methods?

To evaluate whether our semantic graph-based approach can better unify representation across different programming languages compared with state-of-the-art methods, we separately train different baseline models in multiple languages and analyze the effectiveness of different baseline models.

RQ2: Does our proposed approach improve the performance of realistic queries across multi-languages?

To examine whether our semantic graph-based method can unify the representation across different programming languages and achieve satisfied performance on realistic queries, we perform a case study and a series of analyses.

RQ3: What is the effectiveness of the contrastive learning module in our approach?

Since the multi-language code contrastive learning module is designed to use multi-language data better, we analyze its effectiveness through a series of ablation and contrastive experiments.

RQ4: What is the performance of our proposed approach when combined with existing pre-trained models?

Recently, many large-scale pre-trained code models have been proposed for code representation. These models utilize the Masked Language Model task to pre-train the Transformer, and have achieved promising results on downstream code tasks such as code search and code generation. To explore the possibility of combining with the pre-training technique, we combine our method with two typical pre-trained models, *i.e.*, CodeBERT [20] and CodeT5 [32], and fine-tune those models using our multi-language dataset.

A. Experiment Setup

1) Dataset: As described in § III, our model requires a large dataset that consists of source code and the corresponding queries. We first acquire Java and Python code snippets and the corresponding queries from CodeSearchNet [33] released by Github, the largest widely-used dataset for evaluating the performance of code retrieval. Due to the absence of C query-code dataset, we collect C dataset by ourselves. First, we download high-stars (over 300) C projects from Github by the rules in [33] to filter high-quality code snippets. For each programming language, we obtain 39,000 code snippets, queries and corresponding IR by exploiting tools described in § II-C.

2) Baseline Model: In this paper, we use the existing state-of-the-art code search approaches as the baselines for comparison.

- DeepCS [8]. DeepCS exploits function name, API sequence and other code tokens for learning the code representation, through an RNN-based neural network, to jointly embed input queries and code snippets into a high-dimensional vector space.
- CodeSearchNet [33]. CodeSearchNet regards code as tokens and uses different models to learn representation of code tokens, *i.e.*, Natural Bag of Words (NBoW), 1D Convolutional Neural Network (1D-CNN), Bidirectional RNN (biRNN) and Self-Attention (SelfAtten).
- TabCS [34]. TabCS exploits function name, API sequence, tokens, and AST for learning the code representation and embeds the code representation via a two-stage attention network to learn the text features and structural features of the code.

3) Evaluation Metrics: For automatic evaluation, we choose two common metrics to measure the performance of code search: SuccessRate@k, and Mean Reciprocal Rank (MRR). The SuccessRate@k represents the percentage of queries for which more than one correct snippet succeed to exist in the top k ranked snippets returned by a search model, which is calculated as : $SuccessRate@k = (\frac{1}{|Q|}\sum_{q=1}^{Q}\delta(Rank_q \leq k))$, where Q denotes a set of queries, $Rank_q$ denotes the highest rank of the hit snippets in the returned snippet list for the query; $\delta(\cdot)$ denotes an indicator function that returns 1 if the Rank of the q_{th} query ($Rank_q$) is smaller than k otherwise returns 0. SuccessRate@k is important because a better code search engine should allow developers to find the desired snippet by inspecting fewer results. We

TABLE III: Comparison of the overall performance between our model and baselines on MRR metrics (Best scores are in boldface)

Model	C	Java	Python	Avg
DeepCS	0.556	0.362	0.315	0.411
*MuL*DeepCS	0.510(-8.2)	0.368(+1.7)	0.346(+9.4)	0.408 (0.7 ↓)
TabCS	0.676	0.516	0.583	0.592
*MuL*TabCS	0.646(-4.3)	0.505(-2.1)	0.562(-3.6)	0.571 (3.5 ↓)
1D-CNN	0.666	0.517	0.412	0.532
*MuL*1D-CNN	0.616(-7.5)	0.465(-10.1)	0.435(+5.6)	0.505 (5.1 ↓)
biRNN	0.606	0.508	0.378	0.497
*MuL*biRNN	0.615(+1.5)	0.480(-5.5)	0.427(+13.0)	0.508 (2.0 ↑)
SelfAtten	0.702	0.590	0.593	0.628
*MuL*SelfAtten	0.645(-8.1)	0.535(+9.3)	0.559(-5.7)	0.580 (7.6 ↓)
NBoW	0.713	0.612	0.605	0.643
*MuL*NBoW	0.734(+2.9)	0.616(+0.7)	0.612(+1.2)	0.654 (1.7 ↑)
*Single*MulCS	0.721	0.619	0.634	0.658
MulCS	**0.786(+9.0)**	**0.667(+7.8)**	**0.719(+13.4)**	**0.724 (10.0 ↑)**

evaluate SuccessRate@1, SuccessRate@3, SuccessRate@5 respectively and a higher SuccessRate@k value implies a better performance of the code search model. MRR is the average of the reciprocal ranks of all queries Q. The reciprocal rank is the inverse of the highest rank of hit code, *i.e.*, Rank. The computation of MRR is : $MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{Rank_q}$, where Q denotes the set of queries in the automatic evaluation; $Rank_q$ denotes the rank of the ground-truth code corresponding to the q_{th} query. The higher the MRR value, the better the code search performance is.

4) Implementation Details: In our implementation, we build two separate vocabularies for queries and IR tokens and limit their vocabulary sizes to 10,000 and 15,000, respectively. We set the mini-batch size to 320. For each batch, queries are padded with a special token "PAD" to the maximum length, which is set to 30 in our experiments. All tokens in our dataset are converted to lower case and parsed into a sequence of tokens according to camel case and "_" if exists. We set the word embedding size to 300. For LSTM and GGNN units, we set the hidden size to 512. Besides, we set 5 rounds of iteration for GGNN. For the contrastive loss in § III-D, we set the temperature parameter t to 0.05. We update the parameters via AdamW optimizer [35] with the learning rate 0.001. All the models in this paper are trained for 200 epochs. All the experiments are conducted on a server with one Nvidia Tesla V100 GPU, running on Ubuntu 18.04.

B. Experimental Results

RQ1: Effectiveness of multilingual model.

We investigate whether MulCS can better unify representation across different programming languages than state-of-the-art methods. We train different baseline models using monolingual and multilingual data respectively and analyze the effectiveness of baseline models leveraging multilingual data. The test sets for the three languages consist of 2,000 pairs of code snippets and corresponding queries, respectively. This automatic evaluation treats each query as an input and the corresponding code snippet as ground truth. The other 1,999 code snippets are distractor snippets for each input query. We use the same test set to compare monolingual and multilingual models.

Table III shows the overall performance of our model and other baselines, measured in terms of MRR. The first line of each model, such as DeepCS, TabCS and 1D-CNN, results from training with monolingual data(*Single*MulCS represents the result of MulCS training on a single language), the second line of each model, *MuL*DeepCS, *MuL*TabCS and so on are the results of training with multilingual data. Numbers (%) in parentheses indicate the performance improvement using multi-language training over single-language training. The last column Avg represents the average performance across three languages. From this table, we can observe that in all settings (C, Java and Python), the performances of our method are better than the baseline methods. Furthermore, we can see that our multilingual model achieves better improvements than the monolingual model. It shows that MulCS can better unify the representation of different programming languages and better utilize multilingual data to improve model performance. In addition, some token-based code representation models, such as NBoW and biRNN in CodeSearchNet, also achieve improvements in multilingual, similar to the results of [22]. The main reason is that there are some similar identifiers in human-written code (which perform the same function) in different languages. These similar identifiers enable token-based models to utilize multilingual data better, but the results are not significantly improved. Among them, the NBoW model performs the best results among all baselines, which is similar to the results in [22], [36], which shows that the bag of words model is particularly good at keyword matching, which seems to be a necessary facility in implementing search methods. Finally, we find that the multilingual performance of the AST-based model (*i.e.*, TabCS) decreases, the main reason is the differences in the syntactic structure of ASTs in different languages, indicating that the AST-based structural information cannot well unify the representation between different languages.

RQ2: Case study on the queries across multi-languages.

In RQ2, we focus on comparing the effectiveness of these methods on the same realistic query across multi-languages. We construct the same queries across C, Java, and Python since these languages use different training and testing corpus. To achieve this, we need to collect the same queries across C, Java and Python and corresponding multilingual code snippets from CodeSearchNet [33] and our constructed C dataset. Considering this process involves a lot of human effort, we collect 15 widely-used pairs of queries and corresponding multi-language code snippets. We add these 15 pairs to our testing corpus as our case study benchmark. For each query, we calculate the ranking score returned by our method and other baselines(*i.e.*, MuLNBoW and MuLTabCS, which are token-based and AST-based multilingual models. Both models achieve better performance on multilingual baselines).

As shown in the Table IV, the numbers represent the rank of the correct code snippet corresponding to the query, and

TABLE IV: The ranking of the same queries returned by different methods

query	MulCS				*MuL*NBoW				*MuL*TabCS			
	C	Java	Python	Avg	C	Java	Python	Avg	C	Java	Python	Avg
1.parse command line arguments	1	1	1	**1***	11	7	1	6	2	1	3	2
2.return the max value of a collection	1	1	1	**1***	1	1	1	**1***	1	2	1	2
3.compute the squared distance from a point to a line	1	2	1	**2***	2	2	1	**2***	2	1	12	5
4.matrix multiplication function	1	1	1	**1***	3	1	1	2	1	2	2	2
5.sort the given range of items using quick sort	2	1	2	**2***	1	1	2	**2***	10	5	75	30
6.calculates the binary tree height	1	1	1	**1***	1	19	1	7	1	1	1	**1***
7.create a new list	1	1	1	**1***	13	1	1	5	1	1	1	**1***
8.parse the given json string	1	1	1	**1***	1	1	160	54	1	1	2	2
9.find the levenshtein distance between two strings	2	1	1	**2***	2	2	1	**2***	7	1	1	3
10.extract an html tag	1	1	1	**1***	1	22	31	18	1	145	1	49
11.gets a random number in the range min to max	1	3	2	**2***	1	1	2	**2***	1	54	8	21
12.algorithm for depth first searching the vertices of a graph	4	1	1	2	1	1	1	**1***	3	1	3	2
13.return the full path of the database	1	1	21	8	1	1	2	**2***	1	9	15	9
14.compute aes encryption	10	6	4	7	3	4	2	**3***	4	1	2	**3***
15.find the longest common prefix	4	1	6	**4***	1	39	18	19	4	21	1	8

[1] Numbers represent the ranking of the correct code snippet corresponding to the query, and the marker $*$ indicates the best average performance across three languages.

TABLE V: The performance on the same queries returned by different methods

model	R@1	R@3	R@5
MulCS	**0.711**	**0.844**	**0.911**
*MuL*NBoW	0.533	0.778	0.800
*MuL*TabCS	0.489	0.733	0.778

TABLE VI: Effects of contrastive learning

Method	C	Java	Python	Avg
MulCS	**0.786**	**0.667**	**0.719**	**0.724**
-w/o.CL	0.706	0.579	0.525	0.603
*MuL*1D-CNN-w.CL	0.708	0.590	0.570	0.623
*MuL*biRNN-w.CL	0.702	0.610	0.570	0.629
*MuL*NBoW-w.CL	0.728	0.618	0.638	0.661
*MuL*SelfAtten-w.CL	0.760	0.653	0.708	0.707

the marker $*$ indicates the best average performance across three languages. For example, the number 1 means that the correct code snippet searched by the model ranks first among all code snippets in the test set for a given query. We can observe that MulCS outperforms other methods in most cases (achieves the best average performance on 12 of the 15 queries). Furthermore, our approach is the most stable among all methods. In concrete, the ranking scores returned by MulCS only range from 1 to 21, while the ranks of *MuL*NBoW range from 1 to 160 and *MuL*TabCS from 1 to 145. To more comprehensively evaluate model performance, we compute SuccessRate@k for 15 queries across three languages (*i.e.*, 45 queries per model) based on Table IV. As shown in the Table V, the columns R@1, R@3 and R@5 show the results of the average SuccessRate@k over all queries when k is 1, 3 and 5, respectively. These results indicate that our method achieves satisfied and stable results on the widely-used queries. In contrast, baseline methods retrieve the code based on the textual features and syntactic features, making their results unstable since they highly rely on whether the code snippet contains the specific tokens and specific structure.

RQ3: Effectiveness of multilingual contrastive learning. To demonstrate the effectiveness of multi-language code contrastive learning, Table VI ablates the effect of the contrastive learning module, we replace the multilingual code contrastive learning module with the softmax loss used in CodeSearchNet [33] for comparison. Overall, MulCS achieves the best performance when contrastive learning is used, indicating the usefulness of the designed module in our model. This also confirms that the multi-language code contrastive learning module can better learn a unified semantic representation and close the semantic relationship between different languages. Meanwhile, we further apply the contrastive learning module using the same batch size on CodeSearchNet baselines (*i.e.*, 1D-CNN, NBoW, biRNN and SelfAtten) because NBoW achieves better results than other baselines under multilingual training, and biRNN has better multilingual performance gain over different baselines. The results show that the CodeSearchNet models with contrastive learning do not perform as well as our model. This also reflects that our model's unified semantic graph architecture is also significantly beneficial.

RQ4: Effectiveness of combined with existing pre-trained models.

To combine our method with existing pre-trained models, we use CodeBERT [20] and CodeT5 [32] models to obtain vector embeddings of query and code. CodeBERT [20] and CodeT5 [32] are obtained by self-supervised pre-training on a large-scale corpus. To fuse our unified semantic map in the fine-tuning stage, we transform the unified semantic map into sequences via a depth-first search algorithm, which is used as additional information for code representation. Specifically, we use two pre-trained models, CodeBERT and CodeT5, as encoders, respectively. When encoding the code, the input of the original model is the token sequence of the code, and the modified model uses the token sequence of the code and the sequence obtained from the semantic graph as input. The data used in the fine-tuning phase is the same as the MulCS training data. The results are listed in Table VII. Among them, *Mul*CodeBERT and *Mul*CodeT5 fine-tune the pre-trained model on our data, **+IRGraph** add semantic graph sequences as additional information input and **+CL** use multi-language contrastive learning module in fine-tuning stage.

TABLE VII: combined with the pre-trained models

Model	C	Java	Python	Avg
MulCS	0.786	0.667	0.719	0.724
*MulC*odeBERT	0.735	0.643	0.679	0.686
*MulC*odeBERT+IRGraph	0.738	0.656	0.718	0.702
*MulC*odeBERT+CL	0.859	0.750	0.813	0.807
CodeBERT_MulCS(+CL+IRGraph)	**0.867**	**0.756**	**0.869**	**0.831**
*MulC*odeT5	0.736	0.653	0.712	0.700
*MulC*odeT5+IRGraph	0.749	0.653	0.732	0.711
*MulC*odeT5+CL	0.858	0.754	0.828	0.813
CodeT5_MulCS(+CL+IRGraph)	**0.864**	**0.757**	**0.864**	**0.828**

Note that due to memory limit, batch_size is set to 8 for CodeT5 series models and 32 for CodeBERT series models. The results show that combining our method with existing pre-trained models (*i.e.*, adding our multilingual contrastive learning module and unified semantic graph representation in the simply way during the fine-tuning stage) gains additional enhancements. It would indicate a promising direction of multi-language code representation by combining our unified semantic representation model and pre-training techniques. We leave a deep investigation in the future work.

V. DISCUSSION

A. Cost of Preprocessing

In order to obtain a unified representation of multilingual code, we need to build a unified semantic graph. These preprocessing steps can take some time, but can all be processed offline before code search. Specifically, we first preprocess a large amount of code to obtain semantic graphs and then input them into the model to get their vector representation and store them offline. When searching online, we only need to encode the queries and then calculate the similarity with the candidate code to obtain the search results. So the preprocessing involved in this paper has little impact on the performance of the search.

B. Threats to validity

One threat to validity is the extensibility of our proposed method. Our model uses different compilers to obtain the IR of the corresponding language, and there may be non-negligible differences in the form of the initial IRs directly converted by different compilers. For the unification of IR, a lot of work will be put into this step. To minimize this threat, we choose C, Java, and Python, three typical programming languages, and represent structural, object-oriented, and just-in-time languages. Therefore, for adding a new language, we can directly construct it according to the method in this paper. For interpreted languages, if IR is a three-address structure, then the semantic graph can be built using the construction rules in the article. If it is in bytecode format, then the three-address structure can be obtained using Algorithm 1. For compiled languages, the wrapper can be constructed in the same way as C/Java in Table 3.

VI. RELATED WORK

A. Code Search

Traditional code search methods mainly exploit information retrieval and natural language processing technologies [1]–[3], [37]–[41]. The methods utilize textual or syntactic structures to model code. Bajracharya *et al.* [37] proposes a code search engine Sourcerer that can extract fine-grained structural information from source code. Lv *et al.* [3] proposes CodeHow, a code search technique that measures APIs and the queries based on text similarity, and applies an extended boolean model to retrieve code. Recently, with the development of deep learning technologies, plenty of works have been proposed to precisely model the source code [4]–[9], [42]–[47]. CODEnn [8] extracts tokens, filenames and API sequences of code, and uses CNN and RNN to embed this information and queries into a shared space. To utilize diverse features of source code, Wan *et al.* [7] proposed MMAN which uses a multi-modal (tokens, AST and CFG) to represent code. DeGraphCS [23] uses variable-based DFG and CFG to learn code semantics. However, for multilingual code search, graph-based methods are not well utilized due to language-specific syntactic structure, and we tackle this challenge with an IR-based unified semantic graph.

B. Contrastive Learning for Code Representation

Contrastive learning approach learns code representations by closing the distance between similar example (positive) representations while maximizing the distance between different example (negative) representations [48]. Inspired by this, some works [12]–[14], [49] try to apply the contrastive learning method to code representation learning. Bui *et al.* [12] and Jain *et al.* [14] mainly use semantic-preserving program transformations to generate functionally equivalent code snippets and train the model with a contrastive learning technique to identify semantically equivalent (positive examples) and non-equivalent (negative examples) code snippets. Huang *et al.* [13] proposed CoCLR, which incorporates code contrastive learning into the CodeBERT. CoCLR obtains more artificially generated training instances by rewriting query statements, such as randomly deleting a word and randomly swapping. However, for multilingual code search, contrastive learning-based methods are not well utilized due to language-specific syntactic structures, and we use a unified semantic graph representation to address this challenge.

VII. CONCLUSIONS AND FUTURE WORK

We propose a contrastive learning framework for multi-language code search named MulCS. MulCS integrate contrastive learning into the improved gated graph neural network to learn the code representation. To learn more comprehensive semantic information, MulCS extracts data flow and control flow from intermediate representation of multilingual code. We put forward a general semantic graph construction strategy based on the characteristics of various typical programming languages to narrow the gap between grammars of different languages. Furthermore, to acquire the semantic information

of source code, we proposed a snippet wrapper to make the incomplete code without third-party libraries compilable. Our experimental study has shown that the proposed approach can better unify representation across different programming languages and outperforms state-of-the-art methods.

For future work, it would be interesting to investigate how our unified semantic graph approach can be combined with other orthogonal techniques like pre-training, and we have initially demonstrated its effectiveness, which will facilitate the utilization of larger-scale multilingual data. Moreover, we might extend our method to solve other software engineering problems, *e.g.*, cross-language code clone detection.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This research was funded by NSFC No. 61872373 and No. 62272473.

REFERENCES

[1] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 664–675.

[2] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 690–701.

[3] 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)*. IEEE, 2015, pp. 260–270.

[4] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[5] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 826–831.

[6] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," *arXiv preprint arXiv:1802.07779*, 2018.

[7] 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.

[8] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[9] 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*, 2020, pp. 196–207.

[10] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, 2013.

[11] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[12] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[13] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," *arXiv preprint arXiv:2105.13239*, 2021.

[14] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[15] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, W. Qi, D. Jiang, W. Chen, and N. Duan, "Coderetriever: Unimodal and bimodal contrastive learning," *arXiv preprint arXiv:2201.10866*, 2022.

[16] Z. Sun, L. Li, Y. Liu, and X. Du, "On the importance of building high-quality training datasets for neural code search," *arXiv preprint arXiv:2202.06649*, 2022.

[17] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 563–573.

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

[19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[21] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[22] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," *arXiv preprint arXiv:2112.02043*, 2021.

[23] 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.

[24] P. Gupta, N. Mehrotra, and R. Purandare, "Jcoffee: Using compiler feedback to make partial code snippets compilable," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 810–813.

[25] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[26] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[29] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 9729–9738.

[30] A. Van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv e-prints*, pp. arXiv–1807, 2018.

[31] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3733–3742.

[32] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708.

[33] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[34] 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 *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 342–353.

[35] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," 2018.

[36] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," *arXiv preprint arXiv:2111.02671*, 2021.
[37] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 681–682.
[38] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.
[39] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger, "The end-to-end use of source code examples: An exploratory study," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 555–558.
[40] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 475–484.
[41] 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*, 2011, pp. 111–120.
[42] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.
[43] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 163–174.
[44] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.
[45] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 31–41.
[46] D. Wang, Y. Yu, S. Li, W. Dong, J. Wang, and L. Qing, "Mulcode: A multi-task learning approach for source code understanding," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 48–59.
[47] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 287–298.
[48] Y. Liu, X. Yang, S. Zhou, X. Liu, Z. Wang, K. Liang, W. Tu, L. Li, J. Duan, and C. Chen, "Hard sample aware network for contrastive deep graph clustering," *arXiv preprint arXiv:2212.08665*, 2022.
[49] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" in *International Conference on Machine Learning*. PMLR, 2021, pp. 8476–8486.