Intermediate Representation-based Semantic Graph for Cross-language Code Search

Mingyang Geng*, Shangwen Wang*, Dezun Dong*, Shanzhi Gu†, Weijian Ruan‡,
Xiaoguang Mao*, Xiangke Liao*
*National University of Defense Technology, China,
{gengmingyang13, wangshangwen13, dong, xgmao, xkliao}@nudt.edu.cn
†Hunan huishiwei Intelligent Technology Co., Ltd, China, gushanzhi@huishiwei.cn
‡Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China, rweij@whu.edu.cn

***Abstract*—Cross-language code-to-code search has potential to boost software development and software maintenance since it can provide developers with the references to implement a specific functionality or help them migrate the code from one programming language to another. Existing approaches have specific shortcomings: the static ones that rely on the textual or syntactical information have relatively low effectiveness since such information differs significantly across different languages; the one that uses dynamic input-output information has low generality because its application requires the code to be executable, which is hard to guarantee for real-world projects. In this paper, to cope with the aforementioned limitations, we propose a fully static cross-language code searcher,** COSER**. Our key insight is that code snippets in different languages that implement the identical functionality, although may differ significantly with respect to the token sequences or the syntactic structures, could share certain similarities regarding to their Intermediate Representations (IRs). Therefore, we propose to build the semantic graph that can effectively model the semantics of code in different languages based on the data flow and control flow information from their IRs. After that, we design a customized hierarchical pooling mechanism to represent the graph structure, which mimics the human developers' practice of hierarchically understanding a code snippet. Our large-scale evaluation results show that** COSER **can achieve scalability (because it can work for arbitrary code snippets), and effectiveness (because it outperforms the state-of-the-art static approaches by up to 20% with respect to the Mean Reciprocal Rank value) at the same time. Finally, our ablation study shows that all the modules in our approach contribute to the overall effectiveness of** COSER**.**

***Index Terms*—code search, intermediate representation, hierarchical pooling**

I. INTRODUCTION

Software is dominating the world. With the increasing number of software, there may exist similar functions in different software systems. Code-to-code search refers to the task of given a query code snippet, searching for similar code from a repository. Therefore, it holds the potential to boost software development since by doing so developers do not need to implement programs from the scratch and can reuse the knowledge hidden in existing code [1]–[5]. Beyond that, nowadays, there are hundreds of programming languages (PLs) in widespread use [6]. Sometimes, even if references written in the same PL as the current system cannot be found, recommending code snippets in different PLs to the developers can also make sense [7]. Besides, developers often need to migrate an existing code-base written by obsolete PLs to a modern PL [8] (*e.g.*, migrating from `COBOL` to `Java`). Providing developers with the similar code written in the target languages could also boost this software maintenance process. Thus, cross-language code-to-code search is becoming a critical way for programmers to improve their coding productivity during daily routines.

Existing approaches of cross-language code search mainly exploit the textual features or syntactic features, which mainly have the following two limitations. First, the textual information cannot represent structural logic and sometimes would be useless or even noisy (relying on coding style and experience of developers). For example, the variables of the solutions in programming Q&A sites are always named casually (like `a` and `b`), which cannot reflect the semantics of the code. Second, code snippets that implement the same functionality may be quite syntactically different (*e.g.*, summation can be achieved by both *for-loop* and *recursion*), and this problem becomes even worse for cross-language code pairs since different PLs have their own syntactic conventions. It is thus less likely to build a unified and effective representation model for multiple PLs by only using lexical and syntactic analysis, *e.g.*, AST-based approach [9], [10]. To compensate for such imprecision, Mathew *et al.* [11] proposed to use the behavioural features (i.e., the input and output of a code snippet). Nonetheless, such features can only be obtained by executing the code snippets, which is impractical for code from real-world projects whose dependencies are usually bloated and thus the executability is hard to guarantee. Indeed, as reported in the original study, their approach could only work for 9% of code snippets that are from real-world open-source projects. To deepen the understanding towards this limitation, we also performed a pre-study experiment where we tried to execute their tool on the 496,688 Java code snippets of the CodeSearchNet dataset [12], which is a well-known dataset from GitHub open-source projects. Results reveal that only 3.2% of the code snippets can be executed, which further shows the low generality of

*Shangwen Wang is the corresponding author.
This work is partially supported by the Excellent Youth Foundation of Hunan Province under Grant No. 2021JJ10050 and the National Natural Science Foundation of China (Grant No. 61872445, 61672529).

utilizing the dynamic features.

In this paper, we propose COSER, a fully static Cross-language cOde SearchER, to overcome the aforementioned limitations. Our key insight is that code snippets that are written in different programming languages but implement the same functionalities, although may differ significantly with respect to the textual or syntactical features, may share certain similarities regarding to their Intermediate Representations (IRs). Specifically, we observe that both the data flow and control flow information reflected by such IRs can be semantically similar. Therefore, the first step of our approach is to generate IRs for code in different languages and then transform such IRs into a unified format. We then design a customized graph structure (i.e., the **semantic graph**) that is derived from the data and control flow of the IRs, which aims at accurately capturing the semantic information of the code. After that, we design a graph pooling mechanism that models the graph information hierarchically, and such a process mimics the human developers' practice to understand a code snippet where the code is usually comprehended from a hierarchical manner (*e.g.*, from statements to code blocks, and then to the whole code snippets) [13]. Upon obtaining the vectorized representation of a code snippet, we use it to retrieve similar code snippets in the search repository.

Our COSER can currently support three widely-used programming languages, i.e., `C`, `Java`, and `Python`. To evaluate the effectiveness of our proposed COSER, we use the code snippets collected from a program contest dataset. We perform extensive experiments where we totally design six different query-answer language pair settings (i.e., Java-C, C-Java, Java-Python, Python-Java, C-Python, and Python-C). The comparison with two state-of-the-art static cross-language code search tools, COSAL [11] and InferCode [14], shows that our COSER outperforms them under each setting. Specifically, the MRR (Mean Reciprocal Rank) value of COSER can exceed those of the state-of-the-art by more than 20% under specific settings. The ablation study also shows that both the hierarchical pooling mechanism and the critical information in the semantic graph contribute to the overall effectiveness of COSER. Furthermore, an extended evaluation on code snippets from real-world projects shows that compared with those achieved on the contest code, COSER obtains similar performances on such code, which demonstrates the scalability of COSER.

The main contributions of this paper are summarized as follows:

- We propose a general intermediate representation based graph structure (i.e., the **semantic graph**) to effectively capture the semantics of code across different programming languages.
- We implement COSER, which is a fully static cross-language code search approach based on the semantic graphs. The comprehensive experiment results demonstrate the effectiveness of COSER.
- We open source our replication package at **https://github.com/gmy2013/Coser** including the source code of COSER and our evaluation dataset, for follow-up studies.

II. BACKGROUND AND MOTIVATION

A. Background

Intermediate Representations are the data structures or code used internally by a compiler or virtual machine to represent source code [15]. An intermediate representation is composed by a number of machine language instructions. One instruction typically has two elements: `oprands` (*e.g.*, `i` and `%len` in Fig. 1) and `opcodes` (*e.g.*, `cmp` in Fig. 1). The `oprand` denotes any object that is capable of being manipulated. For example, in the expression "1 + 2", the "1" and "2" are the oprands. The `opcode` (operation code) denotes the content of an instruction that specifies the operation to be performed, such as "add", "bitcast", and "ret". Several instructions together constitute a block. Such blocks are the basic units of the Control Flow Graph (CFG) of the source code, whose ending instructions usually point out the control flow of the program (i.e., indicating which block should be executed next). Each basic block starts with a `label` that gives the block a symbol table entry.

Currently, most programming languages have their own Intermediate Representations. For instance, in the widely-known LLVM infrastructure for C language, the IR is given in the Static Single Assignment (SSA) form [16]–[19]. Due to its capability of effectively modeling the data flow and control flow of source code, IR has been widely used in diverse software engineering tasks [16], [20].

B. Motivation

For multiple language code search, we need an approach to effectively represent code semantics of different programming languages. Previous studies mainly exploit textual or syntactical features of each specific programming language by analyzing token sequence or AST structure of a code snippet [21], [22]. However, we observe that code snippets that implement the identical functionality can differ significantly with respect to their token sequences and AST structures, which means the effectiveness of existing approaches could be compromised. A motivating example is illustrated in Fig. 1. In concrete, we list three code snippets in `C`, `Java`, and `Python` languages that implement the ``count the number of positive integers'' functionality, along with their ASTs. From the perspective of tokens, the textual differences between languages are huge, even if their intended functionality is the same. Specifically, all the three code snippets have their unique tokens (i.e., tokens that only occur in this snippet but do not occur in others). For instance, the token `number` in the Java code snippet is a unique one. Furthermore, since the effectiveness of token-based approaches relies heavily on the quality of identifiers, over-simplified identifier names (*e.g.*, `s` in the C code snippet) could impede the effectiveness of such approaches. From the perspective of ASTs, the depicted ASTs are also significantly different. Specifically, the AST for the Python code snippet only contains five levels, less than that of the Java code snippet which is seven. Also, if we count the

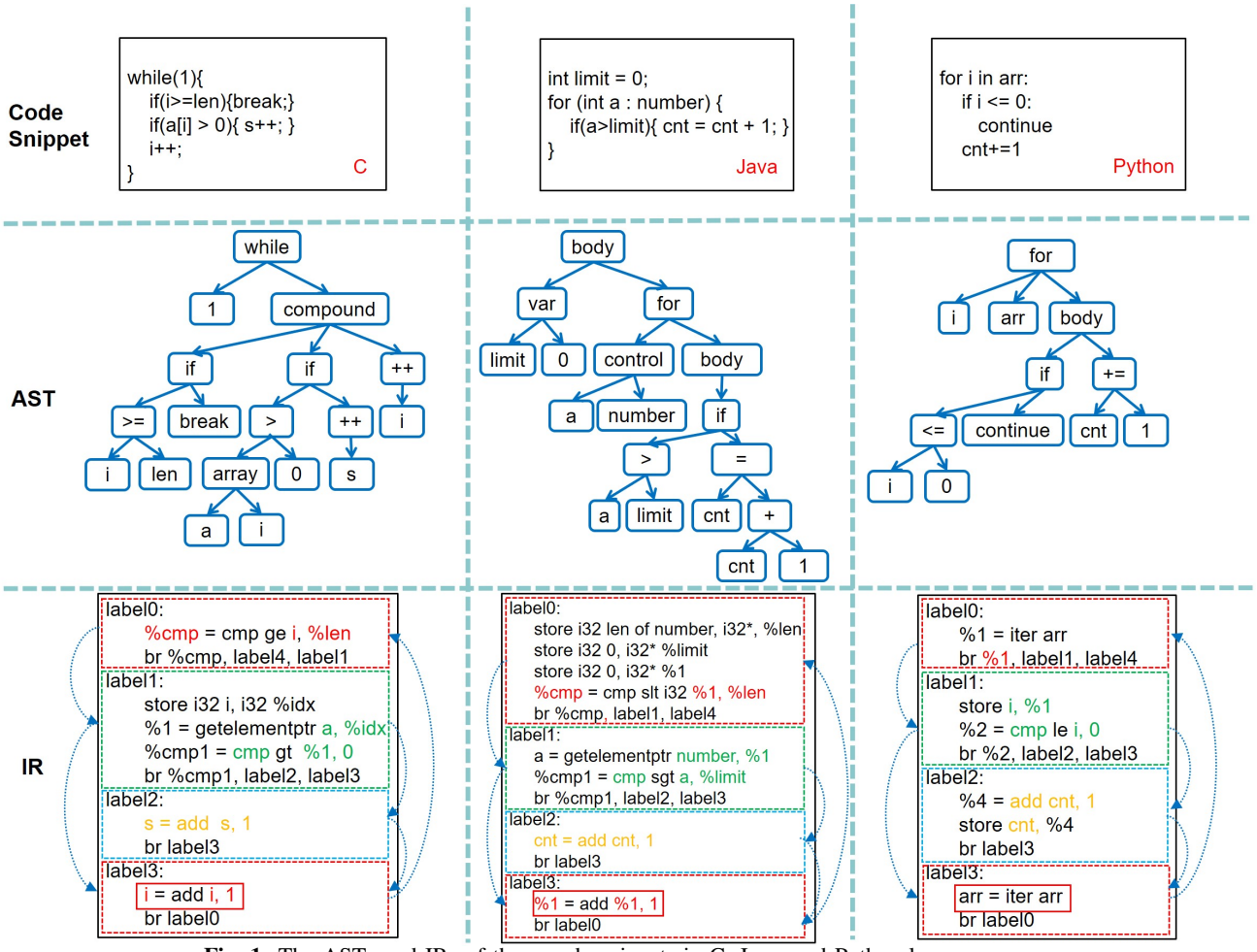

Fig. 1: The ASTs and IRs of three code snippets in C, Java, and Python languages.

number of nodes in the AST, we note that both the ASTs of the C and Java snippets contain totally 18 AST nodes, while such a number is 12 for the Python snippet. Such phenomenons are caused by the different ways to traverse arrays in these three snippets. For instance, the C code snippet uses a `while` loop so it needs another conditional statement (i.e., `if (i >= len)`) that judges if it needs to break this loop, while such a conditional statement does not occur in the other two code snippets. Therefore, leveraging AST for representing code may still fail to capture similar semantics among different PLs.

In Fig. 1, we also illustrate the corresponding IRs of the three code snippets, where the blocks are highlighted in dotted lines with different colors. Note that the illustrated IRs in this figure are all in the *three-address* data structure format, whose processing procedure will be detailed in Section III. We find that these IRs show some certain similarities with respect to the semantics. For instance, we find all of them have a comparison between the value of a cyclic variable and the length of an array (the corresponding contents are illustrated in red color). Theoretically, these contents relate to the loop condition in the source code (*e.g.*, `for (int a : number)` in the Java code). Moreover, another two instances that share the identical semantics are also shown in green and orange colors which represent the comparison between the value of a variable and 0, and the addition to the value of a counter by 1, respectively. We also find that the IRs have similar data and control flows. For data flow, the oprands usually have similar operations. For instance, all of these IRs have an instruction that increases the value of a variable by one (cf. the three red solid frames in the figure). Besides, the three IRs share the same control flows (cf. the blue dotted lines in the figure). Specifically, their control flow includes five jumping relations among different labels: $\{< 0, 1 >, < 1, 2 >, < 1, 3 >, < 2, 3 >, < 3, 1 >\}$, where the number represents the order of the label in the top-down manner.

According to the above analysis, code snippets that implement the same functionality but in different programming languages may ① differ significantly with respect to their token sequences or AST structures; and ② share certain similarities in their IRs. Therefore, IR may be a reasonable approach to represent the semantics of code in different programming languages and we are motivated to use this representation in our approach.

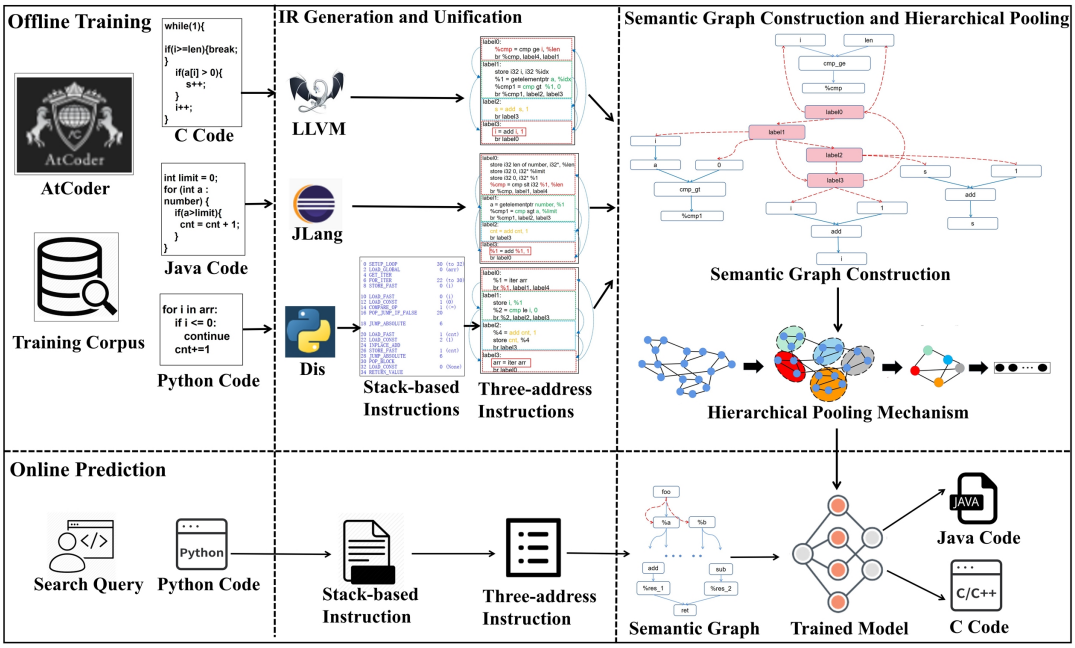

Fig. 2: The Workflow of COSER which illustrates how the code snippets of specified languages are embedded into vectors for searching semantic-relevant snippets of other languages.

III. METHODOLOGY

A. Approach Overview

As shown in Fig. 2, our approach includes two phases: the off-line training phase and the on-line prediction phase. In the training phase, given a code snippet, we first generate its IRs. Since different programming languages have different IRs, we then transform such IRs into a unified format. Then, we build a graph structure which is named as the **semantic graph** to capture the data flow and control flow from IRs. After that, we design a hierarchical pooling mechanism to generate vector representations of the code, which is further utilized for training. After training, given a code snippet in a specific language, COSER can return semantic-relevant code snippets in different languages.

B. IR Generation and Unification

The first step of our approach is to generate Intermediate Representations of code snippets in different languages. We exploit JLang [1], LLVM [2], and the `Dis` module [3] to acquire the IRs for the Java, C, and Python programming languages, respectively. For Java and C languages, one prerequisite for generating the IR is that the code is compilable. To this end, we use the well-known tool named JCoffee [23] for Java and implement the same functionality for C by ourselves.

[1]https://polyglot-compiler.github.io/JLang/
[2]https://github.com/llvm/llvm-project
[3]https://pypi.org/project/dis/

After acquiring the IRs of different programming languages, one challenge we face is that such IRs are in different formats. More specifically, the IRs of the Java and C programming languages are in the *three-address data structure* format, while the IR of the python language is in the *stack-based bytecode* format, as shown in Fig. 2. The three-address instruction is in the format of "result = opcode [list of oprands]". For example, "`s = add s, 1`" is a typical three-address instruction that corresponds to the source code `s = s + 1`. In contrast, the stack-based instruction is in the format of "opcode [oprands]". It usually needs four instructions to represent the above source code which are (1) "LOAD_FAST 0 (s)", (2) "LOAD_CONST 1 (1)", (3) "BINARY_ADD", and (4) "STORE_FAST 0 (s)". Furthermore, different kinds of IRs have different semantic properties. For instance, from the perspective of control flow, the three-address instructions usually jump to the next block that will be executed later at the end of each block, while the stack-related instructions usually explicitly jump to the next specified instruction if it will not execute sequentially. Therefore, to facilitate the following steps of our approach, we need to transform the IRs into a unified format. In our study, we choose to transform the stack-based instructions into the three-address format.

The way to achieve such transformations is briefly illustrated in Algorithm 1. In general, given an instruction in the stack-based format, we first determine whether it is related with the data flow or control flow. Generally speaking, this is performed by analyz-

Algorithm 1: Transform the stack-based instructions into the three-address format.

Input: a set of stack-based instructions I_1.
Output: a set of three-address instructions I_2.

```
Function Transform(I1, I2):
    I2 → φ
    for i in I1 do
        stack → φ
        /* i is data-related.                                   */
        if i.opcode.find("JUMP") == False and i ∉ Non-essential
          Instructions then
            if i.opcode.startwith("LOAD") then
                push i.oprand to stack
            if i.opcode.startwith("STORE") then
                pop i.oprand from stack
            if i.opcode ∈ UNARY or BINARY then
                I2 ← ConstructInstruction(i.opcode, stack)
        /* i is control-related.                                */
        else if i.opcode.find("JUMP") == True then
            add label → {j | j ∈ I2 ∧ j is transformed from
              i.oprand}
            I2 ← CreateInstruction(br label)
        /* i is non-essential                                   */
        else
            continue
    return I2
```

ing if the instruction contains the keyword "JUMP" (*e.g.*, "JUMP_ABSOLUTE","POP_JUMP_IF_FALSE") since, typically, an instruction containing the keyword "JUMP" means the execution will jump to another location, while other instructions usually deal with the data and do not affect the sequential execution of the instructions. Then, for data-related instructions, we maintain a stack to record the related data. For instructions starting with "LOAD", we push the corresponding *oprand* into the stack, while for instructions starting with "STORE", we pop the *oprand*. After encountering unary or binary instructions (*e.g.*, "UNARY_NOT" and "BINARY_POWER"), we combine the *opcode* with the *oprands* in the stack to form a three-address instruction. A concrete example is illustrated in Table I. In this example, the *oprands* "a" and "b" are stored in the stack before the third stack-based instruction. Then, we combine them with the *opcode* "BINARY_ADD" to form the corresponding three-address instruction by storing the result into the *oprand* of the "STORE" instruction (i.e., "c"). Then, the stored contents in our stack will be popped up by the forth instruction. For control-related instructions, we will add labels in front of the instructions that are transformed by the original stack-based instructions pointed to by such instructions, since such instructions are the first of the block that will be executed later. We also create a new instruction (*e.g.*, "br label0") to represent this control flow relation. Another concrete example is shown in Table I. After encountering the "JUMP" instruction, we first identify the instruction it points to (which is the one with the offset of 6), and then identify the three-address instruction that is transformed by this one (i.e., "%1 = iter arr"). Then we add a lable in front of it and also a corresponding jump instruction.

TABLE I: Two instances of the instruction transformation.

Type	Stack-related	Three-address
Data-related	0 LOAD_FAST a 2 LOAD_FAST b 4 BINARY_ADD + 6 STORE c	c = add a, b
Control-related	6 FOR_ITER 22 (to 30) ... ... 18 JUMP_ABSOLUTE 6	label0: %1 = iter arr ... br label0

Note that we also define several non-essential instructions for which we directly ignore. For instance, those start by "POP" or "PUSH" have no relation to the data flow or control flow and thus are ignored by our approach.

C. Semantic Graph Construction

After IR unification, we design a graph structure to represent the semantics of the programs, which is named as the **semantic graph**. Inspired by our motivating example, which shows that code snippets implementing the same functionality share certain similarities regarding to both the data flow and control flow, we take both kinds of information into consideration when building such graphs. Therefore, the nodes in our graph are the opcodes, oprands in the IR as well as the labels, and the edges in the graph represent data dependencies or control dependencies. We next introduce how to build the data and control dependencies.

Data Dependencies. Generally, the data dependencies are built based on the computing-related instructions (*e.g.*, "ADD", "SUB", and "MUL") and function invocation-related instructions (*e.g.*, "CALL" and "INVOKE"). Computing-related instructions are the common instructions shared by different IRs. They usually include unary operations and binary operations. The data flows of such instructions often start from the oprands and point to the calculation result through the opcode, representing that the value of the oprand is passed to the result via a specific kind of calculation. Regarding the instruction "res = add a, b", we link the operands "a" and "b" to the opcode "add", and then link the opcode "add" to the calculation result "res" to build the data flow information. An invocation-related instruction usually has the keyword "CALL/INVOKE", representing that the value of the oprand is passed through the called function. For instance, for the instruction "res = funcname, p1, p2", we link the parameters "p1" and "p2" to "funcname" (i.e., the name of the called function), and then link "funcname" to the return value "res".

To enrich the semantic information of our graph, we also have a special design. For the C and Java languages, each variable has explicit type information, which is also a critical resource for understanding code semantics. Therefore, we add the variable type information as the prefix of the node content (*e.g.*, "int_a" and "double_b") to make the node information more comprehensive.

Control Dependencies. The instructions in each block are executed sequentially. Therefore, the control dependencies are mainly reflected by the jumping relations among different "label" nodes. We first link the "label" node to the nodes whose in-degrees are 0 inside the corresponding block, which indicates that the flow of the corresponding nodes depend on the entrance of the block. After guaranteeing that each "label" node has been connected to the other nodes within its block, we build connections for the "label" nodes according to "br/goto" instructions. For example, if the instruction "br %cmp1, label2, label3" is inside label1, then we link label1 to label2 and label3.

After such a process, we build semantic graphs where the data flow information is reflected by the connections among nodes within the same block, while the control flow information is reflected by the connections among different blocks.

D. Hierarchical Graph Pooling Mechanism

Upon building the semantic graphs, we need to decide which models to use for learning the vector representations of the program semantics. The previous study by Liu *et al.* [13] has revealed that developers comprehend the program in a hierarchical manner. That is to say, developers usually understand the semantics of a statement first, and then the block constituted by several statements (*e.g.*, the FOR loops). Finally, by combining the semantics of all the blocks, they may understand the whole code snippet. Inspired by this finding, we decide to model the graph also in a hierarchical manner. We recall that our graph captures the data flow within each block and the control flow among different blocks. Therefore, we can readily model the individual blocks first and then aggregate different blocks, imitating the manner of human developers.

Following the well-known DiffPool [24], a graph pooling module that can generate hierarchical representations of graphs, we design our own pooling mechanism. The key idea is to enable the construction of deep, multi-layer GNNs (Graph Neural Networks) [25] by providing a differentiable module to hierarchically infer and aggregate the semantic information. Then, the nodes at the l_{th} layer correspond to the clusters assigned by the GNN module at the $l-1_{th}$ layer.

Formally, suppose the cluster assignment matrix at the l_{th} layer is $S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$, where n_l denotes the number of clusters in the l_{th} layer and so does n_{l+1}, $S^{(l)}$ provides a soft assignment of each cluster at the l_{th} layer to a cluster in the next coarsened layer (i.e., the $l+1_{th}$ layer). Since we already have the hierarchy of the semantic graph (i.e., the different blocks), the $S^{(l)}$ is calculated based on such information (*e.g.*, nodes which belong to the same block should be classified into the same cluster). We denote the input adjacency matrix at the l_{th} layer as $A^{(l)} \in \mathbb{R}^{n_l \times n_l}$, the input cluster features for the l_{th} layer as $X^{(l)} \in \mathbb{R}^{n_l \times d}$, the cluster embedding at the l_{th} layer as $Z^{(l)} \in \mathbb{R}^{n_l \times emb\_sz}$, where d and emb_sz are two hyper-parameters. Then, the cluster embedding of the l_{th} layer ($Z^{(l)}$) can be calculated using the following equation:

$$Z^{(l)} = \mathrm{GNN}\left(A^{(l)}, X^{(l)}\right)$$

where we concatenate the adjacency matrix $A^{(l)}$ with the pooled features for the clusters $X^{(l)}$, and pass them through a standard GNN to get new embeddings $Z^{(l)}$ for the clusters. After that, $X^{(l+1)}$ and $A^{(l+1)}$ could be iteratively calculated as:

$$X^{(l+1)} = {S^{(l)}}^T Z^{(l)}$$

$$A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)}$$

Note that when $l = 0$, the inputs to the first equation (i.e., $A^{(0)}$ and $X^{(0)}$) are the adjacency matrix of the input graph A and the node features of the original graph X. In our study, X is calculated as a one-hot vector that represents the index of each node in a fixed vocabulary.

Suppose the total number of the layers is L, the assignment matrix $S^{(L-1)}$ is set to a vector of 1's because all nodes at the final layer are assigned to a single cluster generating a final embedding vector corresponding to the entire graph. The final output embedding (*e.g.*, $Z^{(L)}$) can then be used as the representation of the semantic graph, as well as the features for the similarity comparison.

We only briefly introduce our hierarchical pooling mechanism here due to the page limitation. Readers can refer to the previous study [24] for detailed information. Note that although our approach design is inspired by the DiffPool, our mechanism has two main differences compared with it. The first is that in DiffPool, the cluster assignment matrix revealing the connectivity between layers are predicted by a GNN, while in our mechanism, the cluster assignment matrix is calculated in advance based on the hierarchies of our semantic graph. That is to say, the aggregation of the graph neural network is determined once the semantic graph is built, rather than learned from the model. The second difference is that our pooling mechanism can support multiple edge types (*e.g.*, the data dependencies and control dependencies). This is fulfilled by formulating a convolution-like operation on graph signals performed in the spatial domain where filter weights are conditioned on edge labels [26]. In contrast, all the edges in Diffpool have only one single type.

E. Training & Prediction

To train the whole architecture end-to-end, we exploit the cosine similarity function to back-propagate the gradients. Given a code snippet from a specific language, our model is expected to make its semantic-relevant code in other languages have similar representations with it, while make its semantic-irrelevant code have different representations. Formally, for a code snippet representation $c \in C$, a semantic-relevant code snippet of c in another language $d^+ \in D$ and a randomly chosen semantic-irrelevant code snippet $d^- \in D$, we need to make the vector representations of c and d^+ similar to each other and the vector representations of c and d^- different to each other. Therefore, we train the model by minimizing the loss function $L(\theta)$:

$$L(\theta) = \sum_{c \in C} \sum_{d^+, d^- \in D} \max\left(0, \beta - \cos\left(c, d^+\right) + \cos\left(c, d^-\right)\right)$$

$$\cos(\mathbf{c}, \mathbf{d}) = \frac{\mathbf{c}^T \mathbf{d}}{\|\mathbf{c}\| \|\mathbf{d}\|}$$

Here, the value of the offset, β, is set to 0.05 following the setting in the previous study [27].

For the prediction, given a code snippet in a specific language, we need to select the semantic-relevant ones in other languages from the search corpus. In concrete, we first extract the IR of the given code snippet and ensure it is in the unified three-address format. Then, the semantic graph is built and fed into the hierarchical graph pooling module to get the embedding of the code snippet. Similarly, for each candidate code snippet from the search corpus, we get the final embedding using the pipeline above and calculate its similarity with the query code snippet. Finally, the candidate code snippets with high similarity scores will be returned to the user.

IV. Experiment Results

- **RQ1: How effective is COSER on cross-language code search compared with the baseline methods?**
- **RQ2: How do our hierarchical graph pooling mechanism and control flow information affect the final search results?**

RQ1 aims to investigate whether COSER outperforms the state-of-the-art cross-language code search approaches. Specifically, we set totally six query-answer pairs (i.e., Java-C, Java-Python, C-Python, C-Java, Python-Java, and Python-C) to comprehensively answer this question. RQ2 aims to investigate to what extent the critical components contribute to the effectiveness of COSER.

A. Experimental Setup

1) Baselines and Variants: In RQ1, we choose to compare COSER with the state-of-the-art cross-language code search approaches, i.e., InferCode [14] and COSAL [11].

- **InferCode.** InferCode is a self-supervised code representation approach that supports generating semantic representations for code written in diverse programming languages. To apply this approach on the cross-language code search task, we call the InferCode API [4] to generate embeddings for the code snippets and then calculate cosine similarity between the query code snippet and those in the search corpus, following the original study [14].
- **COSAL.** Code-to-Code Search Across Languages (COSAL) exploits both static and dynamic analyses to identify similar code. Code snippets are ranked using non-dominated sorting based on three kinds of information: the code token similarity, structural similarity, and behavior similarity. Note that the behavior similarity can only be obtained after executing the code. Therefore, in our study, we compare with three variants of COSAL: COSAL_token, which denotes COSAL with only the token similarity; COSAL_AST, which denotes COSAL with only the AST similarity; and COSAL_static which denotes the combination of both token and AST similarity. Our comparison is fair since all the included approaches are fully static. Indeed, the executability of the code snippet is hard to guarantee in practice as we have illustrated in the Introduction, which could impede the generality of COSAL significantly. Note that this approach does not require a training process, and we directly reuse the source code released by the authors [5] to calculate the similarity between the query code snippet and each code snippet in the search corpus.

To answer RQ2, we compare COSER with some of its variants as follows:

- **-hierarchical pooling**: In this variant, we remove the hierarchical graph pooling mechanism from COSER and directly use a plain GNN (Graph Neural Network) [25] for embedding graphs. This variant is used to investigate the effectiveness enhancement brought by the hierarchical graph pooling mechanism.
- **-control flow**: In this variant, we remove the control flow in the semantic graph and only the data flow is saved. We do not investigate the contribution from data flow since removing such data will make the graph unconnected.

2) Dataset: Following the previous study [11] on cross-language code search, we exploit the dataset collected from AtCoder[6] which contains similar code snippets in most popular programming languages, such as C, Java, Python, and C#. AtCoder has open problems where users can freely submit their solutions. Those with syntax errors or do not pass the test suite are filtered automatically, and the accepted submissions for the same problem implement an identical functionality, which makes the ground truth readily available.

Test set. To compare with COSAL, we also focus on the latest 398 problems of AtCoder following their evaluation. Note that their study does not include C language so we crawled the solutions in C language along with Java and Python languages for these problems. Totally, for the 398 problems, 314 of them have both Java and Python solutions, 327 of them have both Java and C solutions, and the number for Python and C languages is 308.

Training set. To train our COSER, we crawled the data from another 742 problems. These problems ensure that for each query-answer language pair (*e.g.*, Java-C), there are 600 problems that have solutions from both languages. Then, for example, given a code snippet in Java language, all code snippets in C language targeting the same problem are considered as semantic-relevant ones while others in C language are considered as semantic-irrelevant ones. After training, the model can be used bidirectionally, i.e., it can accept code snippets in Java language as inputs and output similar snippets in C language, and vice versa.

[4]https://github.com/bdqnghi/infercode

[5]https://doi.org/10.5281/zenodo.4968705

[6]https://atcoder.jp/

TABLE II: Statistics of our AtCoder Dataset.

Metric	Training			Validating			Testing		
	Java	Python	C	Java	Python	C	Java	Python	C
#Problems	679	635	688	71	65	74	361	325	385
#Snippets	46,304	44,175	38,754	4,567	5,063	4,109	20,343	21,517	22,458
Avg. Snippets/Problems	65	70	53	63	76	54	57	67	58
Avg. Lines/Snippet	53	13	46	52	16	40	51	14	43

Validation set. We also crawled the data for 84 extra problems to be used as our validation set. This dataset has 60 problems which have solutions from both languages of each query-answer language pair. The statistics of our dataset are shown in Table II.

3) Evaluation Metrics: For the cross-language code search task, we use the widely-used metrics Precision@k, SuccessRate@k and MRR for our evaluation, following the previous study [11]. Precision@k (P@k) denotes the average percentage of relevant results in the top-k searched results for a query [27], [28]. It reflects on average, how relevant the top-k results are to the query. SuccessRate@k (SR@k) denotes the percentage of queries for which one or more relevant result exists among the top-k searched results [27], [29]. It reflects if there is any relevant code in the top-k results. MRR denotes the Mean Reciprocal Rank of the top-ranked relevant result for a query [27]–[29]: the higher, the better.

Consider a query q in a set of queries Q, we denote R_q^k as the set of all relevant results in the top k results for q. Since there are multiple relevant results for a given query q, we denote $BR(q)$ as the rank of the first relevant search result for q, δ_k as an indicator function which returns 0 if the input is larger than k and 1 otherwise. Then the metrics can be calculated as:

$$\text{P@k} = \frac{\sum_{q \in Q} \frac{|R_q^k|}{k}}{|Q|} \qquad \text{SR@k} = \frac{\sum_{q \in Q} \delta_k(BR(q))}{|Q|}$$

$$\text{MRR} = \frac{\sum_{q \in Q} \frac{1}{BR(q)}}{|Q|}$$

Note that we do not take *recall* into account since as a recommendation tool, the usefulness of COSER is mainly reflected by how many cases a user should manually check before finding the relevant answers. Therefore, we focus more on the rank of the relevant answers (*e.g.*, MRR and SR@k) than how many relevant answers are involved in the returned results (which is targeted by *recall*).

4) Training and Prediction: To train our proposed model, we first shuffle the training data. All the hyper-parameters are reused from the DiffPool study. Specifically, the mini-batch size is set to 20, the dropout value is set to be 0.1 for keeping the model free from overfitting, the number of clusters in the next layer is set to 10% of the number of clusters in the current layer, the dimension of the cluster features as well as the dimension of the cluster embedding (i.e., d and emb_{sz} in Section III-D) are set to 256, and we update the parameters via Adam optimizer [30] with the learning rate 0.001. For each language, we build a vocabulary using camelCase casing conventions and BPE (Byte Pair Encoding) [31] for the graph nodes to store the most frequently appeared nodes in the training dataset and we limit the size to be 16,000. The maximum number of input nodes in our semantic graph is set to 500. This is empirically determined since, according to the node number distribution in our dataset, 95% of the graphs have no more than 500 nodes. The model is trained for 1000 epochs with early stopping mechanism. All the experiments are implemented using Pytorch 1.4 framework with Python 3.6, and the experiments are conducted on a DGX1 server with eight Nvidia Tesla V100 GPUs, running on Ubuntu 18.04. For the baselines, we reproduce the COSAL approach using the replication package provided by the authors.

Following the evaluation manner of [11], in our study, the prediction is performed using the "leave-one-out" cross-validation [32] strategy where each code snippet is exploited as a query against all code snippets in another specific language in the code corpus.

B. Results

1) RQ1-The effectiveness of COSER*:* The effectiveness of COSER is illustrated in Table III, together with those of the baselines. We observe that COSER outperforms the baselines in all metrics under all the six query-answer language pair (*e.g.*, Java-C) settings. For instance, when the search query is in Java language and the target language is Python (i.e., the query-answer language pair is Java-Python), the MRR value of COSER reaches 0.51 while the highest value of the baselines is 0.42 from COSAL_static. We witness an increase of 21.4% (0.09/0.42) for COSER under this experiment setting. As for the $Precision@K$ and $SuccessRate@K$, the performances of COSER also exceed those of the baselines systematically. Specifically, the $Precision@10$ of COSER is 50%, which indicates that on average, 50% of the code snippets within the top-10 search results are semantically identical to the query code, while the best performance of the baselines is 48%. Similarly, the $SuccessRate@10$ of COSER reaches 90%, which means that under most conditions (i.e., 90% of the total cases), at least one semantically identical code snippet occurs in the top-10 search results. As for comparison, the highest value of the baselines is 85%, comparably lower than that of COSER.

For other query-search language pair settings, we observe similar phenomena. Concretely, from the perspective of the MRR, the value of COSER outperforms that of the state-of-the-art baseline by at least 3% (i.e., under the Python-C setting) and 7% under most conditions (i.e., the Java-C, C-Python,

TABLE III: The effectiveness of CODER (in %).

Method	Java→ C Retrieval			Java→ Python Retrieval			C→ Python Retrieval		
	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10
InferCode	29	27/29/32/30	28/47/54/65	27	24/27/25/28	25/39/51/60	25	22/26/29/28	21/33/42/57
COSAL_token	34	30/33/46/51	33/52/63/77	30	28/33/38/41	26/45/56/69	30	25/28/36/39	26/42/54/66
COSAL_AST	39	36/44/51/49	37/48/64/84	35	33/42/46/43	35/43/57/83	32	33/39/44/42	32/40/56/77
COSAL_static	46	44/49/53/51	42/74/87/90	42	41/47/45/48	41/73/83/85	39	38/42/44/47	39/69/81/84
CODER	**53**	**46/51/55/54**	**47/79/91/93**	**51**	**44/49/47/50**	**44/77/88/90**	**46**	**41/45/46/50**	**43/74/83/87**
Method	C→ Java Retrieval			Python→ Java Retrieval			Python→ C Retrieval		
	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10
InferCode	27	25/26/29/28	26/39/52/63	30	23/29/32/35	27/40/54/62	24	23/28/26/31	24/37/45/60
COSAL_token	32	30/32/45/48	31/50/61/74	33	26/33/42/40	29/47/62/76	29	27/31/34/37	28/44/52/69
COSAL_AST	38	35/42/50/48	36/47/60/81	35	36/42/41/43	38/46/59/84	34	32/37/46/45	33/42/54/78
COSAL_static	44	45/48/51/49	41/73/84/88	45	42/44/43/47	41/74/86/88	40	39/44/46/45	40/67/82/86
CODER	**51**	**47/49/53/51**	**43/77/87/90**	**52**	**45/48/47/49**	**45/77/87/91**	**43**	**42/46/48/47**	**42/71/84/89**

C-Java, and Python-Java settings), leading to the increase ratios range from 7.5% to 21.4%. When it comes to the $Precision@K$ and $SuccessRate@K$, we note that generally, the $Precision@10$ and $SuccessRate@10$ of CODER can exceed 50% and 90% respectively, while those of the state-of-the-art usually reach 45% and 85%.

The effectiveness of CODER *is systematically better than those of the baselines. The MRR value of* CODER *can exceed that of the state-of-the-art by more than 20% under specific experiment settings.*

2) RQ2-The contribution of different modules: The results of our ablation study are shown in Table IV. We find that both the hierarchical graph pooling mechanism and the control flow information contribute to the overall effectiveness of CODER. For instance, when the query-answer language pair is C-Java, the MRR value of CODER is 0.51, which will decline to 0.48 (a decrease of 5.9%) when the hierarchical graph pooling mechanism is not used to embed the graph information and 0.47 (a decrease of 7.8%) when the control flow information is excluded from the graph information.

Another notable phenomenon is that the effectiveness of CODER when the control flow information is excluded is systematically lower than that of the approach variant where the graph is normally embedded. Specifically, under the six different query-answer language pair settings, the MRR values of the variant without the hierarchical graph pooling mechanism exceed those of another approach variant by at least 1% (at most 3%, and 2% in most cases). This may indicate that the jumping relations of different code blocks, which are reflected by the control flow information, play more critical roles in representing the code semantics than the way of embedding the graph.

Both the hierarchical graph pooling mechanism and the control flow information contribute to the overall effectiveness of CODER. *The latter seems to be a more rewarding factor.*

V. DISCUSSION

A. The Effectiveness of CODER on Real-World Projects

Our evaluation dataset, AtCoder, is from a programming contest, where the code may differ from those in the real-world projects. We therefore propose to investigate the effectiveness of CODER on code snippets from open-source projects. Specifically, we exploit the *CodeSearchNet Challenge dataset* [12] which consists of 137 Java code snippets and 543 Python code snippets that are related to 99 general natural language search queries. This dataset is manually annotated by experts and the semantic relevance of our selected snippets to the queries is labelled as *exactly match*. Therefore, code snippets related to the same query can be considered as semantically identical. The dataset does not contain code snippets of C language. For each search query, we randomly select one answer in Java language and one answer in Python language. We also use the "leave-one-out" strategy and we discard the metric $Precision@K$ since there is only one ground-truth answer for each query.

Results are shown in Table V. We observe CODER achieves similar effectiveness as reported in Table III. Specifically, its MRR values are around 0.5 and the SuccessRate@10 can reach nearly 90% under both settings. All these values are significantly higher than those from the baselines. For instance, the highest MRR value of the state-of-the-art approach is 0.4 under the Java-Python setting, outperformed by CODER by 20%. We conclude that CODER achieves promising generality: it can work well for both contest code snippets and real-world projects.

B. The Efficiency of CODER

As a fully static approach, we discuss the efficiency of CODER in this section. Our working machine is with Intel Xeon CPU (1T memory) and NVIDIA V100 GPU (32G memory). In the Python-Java setting, it takes on average 0.023s for each Python snippet in the test set to construct the semantic graph, and 0.004s to search for the results using the trained model. We have tested on other different settings and obtained similar results. The results indicate that once trained, the time consumption of CODER is affordable in practice.

TABLE IV: The results of the ablation study (in %).

Method	Java→ C Retrieval			Java→ Python Retrieval			C→ Python Retrieval		
	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10
- hierarchical pooling	50	43/46/52/50	44/73/85/87	47	42/45/44/45	41/68/83/85	44	38/42/40/43	39/66/80/83
- control flow	48	41/45/51/49	42/70/83/85	45	41/43/42/44	40/67/80/83	42	37/40/38/42	38/61/75/80
COSER	**53**	**46/51/55/54**	**47/79/91/93**	**51**	**44/49/47/50**	**44/77/88/90**	**46**	**41/45/46/50**	**43/74/83/87**
Method	C→ Java Retrieval			Python→ Java Retrieval			Python→ C Retrieval		
	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10	MRR	P@1/3/5/10	SR@1/3/5/10
- hierarchical pooling	48	40/45/48/47	42/71/80/83	50	42/46/45/47	41/70/83/86	42	39/42/40/41	41/70/81/83
- control flow	47	39/43/47/45	40/68/77/80	48	40/44/42/46	39/69/80/84	39	37/41/38/39	39/66/76/81
COSER	**51**	**47/49/53/51**	**43/77/87/90**	**52**	**45/48/47/49**	**45/77/87/91**	**43**	**42/46/48/47**	**42/71/84/89**

TABLE V: The effectiveness of COSER on code snippets from real-world projects (in %).

Method	Java→ Python Retrieval		Python→ Java Retrieval	
	MRR	SR/1/3/5/10	MRR	SR/1/3/5/10
InferCode	28	16/28/39/62	27	15/33/45/52
COSAL_token	31	18/32/42/68	29	14/37/49/59
COSAL_AST	33	20/36/49/74	34	18/42/56/76
COSAL_static	40	24/39/56/87	38	21/48/63/73
COSER	**48**	**33/51/69/89**	**49**	**34/50/73/88**

C. Threats to Validity

External threats. Our current evaluation focuses on three widely-used languages where C and Java are representatives for the compiled programming languages, and Python is a typical interpreted programming language. COSER can generalize to any programming language that supports IR extractions. The IRs in other formats could also be transformed to the universal three-address representation via customized processes.

Internal threats. COSER relies on three open-source tools (i.e., LLVM, JLang, and DIS) to generate the IRs for code snippets. The performance of these tools may affect the validity of COSER. Nonetheless, these tools are widely-used and well-maintained, which alleviates such threats.

VI. Related Work

A. Code Search

Traditionally, code search approaches mainly exploit information retrieval and natural language processing techniques [33]–[42]. Such approaches take natural language descriptions as the queries. For instance, Lv *et al.* [37] proposed CodeHow, a code search technique which measures APIs and the queries based on text similarity, and applied an extended boolean model to retrieve code. Some code search tools also take code snippets as the inputs [11], [28], [29], [43], [44]. For example, FACOY [28] is a novel approach for statically finding code snippets which may be semantically similar to the user input code by implementing a query alternation strategy.

Recently, cross-language code-to-code search has received increasing attentions from researchers due to its potential of reducing the software maintenance efforts. Except for the baseline approaches in our study, COSAL and InferCode, AROMA [29] also uses static analysis to support search across a number of different languages but is not included in this study since it is not open sourced. Some studies also focus on finding similar code snippets across different languages. CLCDSA [7] detects cross-language similar code snippets based on the API call similarities. Perez *et al.* [45] performs such a task by using an unsupervised learning approach. In our work, we propose a fully static cross-language code-to-code search approach that compensates for the limitation of the current approach, COSAL, whose promising effectiveness requires the executability of the code snippets as the prerequisite.

B. Modeling Source Code by Graphs

The graph structures are widely-used in software engineering studies to represent the program. Researchers propose to use the data/control flow graph [21], [46]–[48] that captures the data/control flow relations of the program, and the program dependency graph [49] that captures the dependency relations among different program entities to model the program semantics. Upon building such graphs, a number of downstream tasks could be addressed with the help of the graph information, such as automatic code summarization [50], vulnerability detection [51], [52], flaky test detection [53], and automatic bug localization and program repair [47], [54].

Despite promising results achieved, one thing that prevents the existing approaches being applied to the task in this study is that the graph structures typically rely on the AST to serve as the backbone. For instance, on the basis of the AST, Allamanis *et al.* [47] add some additional nodes and edges to construct a graph. However, approaches derived from the AST may not be suitable for our task since we observe that code snippets in different languages may have different ASTs. Therefore, our semantic graph is built on top of the IR, which is a reasonable way to fulfill our target.

VII. Conclusion

In this paper, we propose an intermediate representation based cross-language code-to-code search approach. The key observation is that code snippets in different languages which implement the same functionality may differ significantly with respect to the token sequences or ASTs but share certain similarities in the IRs. We therefore propose to build the semantic graphs that capture the data flow and control flow information from the IRs. After that, a customized graph neural network is utilized to generate representations for each code snippet and further to fulfill the search. Comparisons with existing static approaches show promising results: our COSER can systematically outperform the state-of-the-art under six different query-answer language pair settings.

REFERENCES

[1] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *28th International Conference on Program Comprehension (ICPC)*, 2020.

[2] 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.

[3] S. P. Reiss, "Semantics-based code search," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 243–253.

[4] 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.

[5] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, "Code search based on context-aware code translation," *arXiv preprint arXiv:2202.08029*, 2022.

[6] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *2013 IEEE 37th annual computer software and applications conference*. IEEE, 2013, pp. 303–312.

[7] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[8] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[9] 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.

[10] 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.

[11] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 205–217.

[12] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[13] F. Liu, L. Zhang, and Z. Jin, "Modeling programs hierarchically with stack-augmented lstm," *Journal of Systems and Software*, vol. 164, p. 110547, 2020.

[14] N. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[15] "Intermediate representation," https://en.wikipedia.org/wiki/Intermediate_representation.

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

[17] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada, "Improving syntactical clone detection methods through the use of an intermediate representation," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 8–14.

[18] M. Kim, J.-K. Park, S. Kim, I. Yang, H. Jung, and S.-M. Moon, "Output-based intermediate representation for translation of test-pattern program," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[19] M. Menshikov, "Midair: An intermediate representation for multi-purpose program analysis," in *International Conference on Computational Science and Its Applications*. Springer, 2020, pp. 544–559.

[20] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-sql in cross-domain database with intermediate representation," *arXiv preprint arXiv:1905.08205*, 2019.

[21] 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.

[22] 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.

[23] 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.

[24] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," *arXiv preprint arXiv:1806.08804*, 2018.

[25] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[26] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3693–3702.

[27] 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.

[28] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.

[29] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

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

[31] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, "Byte pair encoding: A text compression scheme that accelerates pattern matching," 1999.

[32] T.-T. Wong, "Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation," *Pattern Recognition*, vol. 48, no. 9, pp. 2839–2846, 2015.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2011.

[39] 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.

[40] 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.

[41] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems & Software*, vol. 89, no. MAR., pp. 51–62, 2014.

[42] X. Zhou, H. Wang, W. Peng, B. Ding, and R. Wang, "Solving multi-scenario cardinality constrained optimization problems via multi-objective evolutionary algorithms," vol. 062, no. 009, pp. 69–86, 2019.

[43] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[44] K. Krugler, "Krugle code search architecture," *Finding Source Code on the Web for Remix and Reuse*, pp. 103–120, 2013.

[45] D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 518–528.

[46] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[47] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, 2018.

[48] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *arXiv preprint arXiv:1805.08490*, 2018.

[49] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[50] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[51] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[52] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 378–389.

[53] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. F. Bissyandé, "Peeler: Learning to effectively predict flakiness without running tests," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[54] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676.