deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search

CHEN ZENG, YUE YU, and SHANSHAN LI, School of Computer, National University of Defense Technology, China
XIN XIA, College of Computer Science and Technology, Zhejiang University, China
ZHIMING WANG, MINGYANG GENG, LINXIAO BAI, WEI DONG, and XIANGKE LIAO, School of Computer, National University of Defense Technology, China

With the rapid increase of public code repositories, developers maintain a great desire to retrieve precise code snippets by using natural language. Despite existing deep learning-based approaches that provide end-to-end solutions (i.e., accept natural language as queries and show related code fragments), the performance of code search in the large-scale repositories is still low in accuracy because of the code representation (e.g., AST) and modeling (e.g., directly fusing features in the attention stage).

In this paper, we propose a novel learnable *de*ep *G*raph for *C*ode *S*earch (called **deGraphCS**) to transfer source code into variable-based flow graphs based on an intermediate representation technique, which can model code semantics more precisely than directly processing the code as text or using the syntax tree representation. Furthermore, we propose a graph optimization mechanism to refine the code representation and apply an improved gated graph neural network to model variable-based flow graphs. To evaluate the effectiveness of **deGraphCS**, we collect a large-scale dataset from GitHub containing 41,152 code snippets written in the C language and reproduce several typical deep code search methods for comparison. The experimental results show that **deGraphCS** can achieve state-of-the-art performance and accurately retrieve code snippets satisfying the needs of the users.

CCS Concepts: • **Software and its engineering → Software development techniques**;

Additional Key Words and Phrases: Intermediate representation, graph neural networks, code search, deep learning

ACM Reference format:
Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2023. deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 34 (March 2023), 27 pages.
https://doi.org/10.1145/3546066

This work is supported by National Key R&D Program of China (2020AAA0103504), National Natural Science Foundation of China (No. 61690203 and No. 61872373), and the Major Key Project of PCL.

Authors' addresses: C. Zeng, Y. Yu (corresponding author), S. Li (corresponding author), Z. Wang, M. Geng, L. Bai, W. Dong, and X. Liao, School of Computer, National University of Defense Technology, Changsha, China; emails: {zengchen15, yuyue, shanshanli, wangzhiming14, gengmingyang13, linxiao_b, wdong, xkliao}@nudt.edu.cn; X. Xia, College of Computer Science and Technology, Zhejiang University, Hangzhou, China; email: xin.xia@acm.org.

1049-331X/2023/03-ART34 $15.00
https://doi.org/10.1145/3546066

1 INTRODUCTION

Code search has received increasing attention in recent years [7, 10, 20, 41, 42, 58, 64]. The goal of code search is to retrieve code fragments that best meet developers' needs by performing natural language queries over a large code corpus. With the availability of immense and rapidly growing source code repositories such as GitHub[1] and GitLab,[2] it is more convenient for developers to search the needed code with certain functionality and reuse it in their programs. However, increasingly complex and diverse code implementations also create considerable challenges in performing a precise code search.

In the early stage, code search approaches are proposed on the basis of information retrieval techniques, especially keyword matching mechanisms [4, 8, 9, 28, 34, 39, 45–47]. However, a common problem in these works is the lack of structural or semantic information from the source code since they simply consider code and queries as plain texts. Recently, deep learning technologies have been applied to represent code and queries as vectors for code search [7, 10, 20, 58, 64] to address the above issues. The typical approach, called DeepCS [20], presents a code search engine by learning a joint embedding of a method description and its corresponding code snippet. Moreover, Wan et al. [64] design a **multimodal attention network (MMAN)** to capture various code features simultaneously, such as code tokens, **abstract syntax trees (ASTs)** and **statement-based control-flow graphs (S-CFGs)**.

However, existing deep learning-based approaches are still limited in two major aspects. First, code with different syntax may achieve the same functionality, while code with similar structural features may express totally different code semantics. Thus, the token (e.g., method name or identifiers) and structural features (e.g., AST or S-CFG) have difficulty in precisely expressing the in-depth semantics of source code in various forms (as shown in Figures 1 and 2). Second, existing methods cannot fully exploit multiple valuable features extracted from the source code. Specifically, some models do not fuse different source code modalities effectively, which does not bring much improvement yet increases the complexity. For example, in [64], MMAN uses the single token-based modality and gets **MRR (mean reciprocal rank)** of 0.437 and SuccessRate@1 of 0.327. In contrast, MMAN proposes an attention network to assign learnable weights to three different source code modalities (i.e., Token + AST + S-CFG) and gets MRR of 0.452 and SuccessRate@1 of 0.347. As the results show, the improvement to the single token-based modality is not very significant, about 4.63% and 4.62% in terms of MRR and SuccessRate@1.

These aforementioned limitations inspire us to design a model to effectively integrate deep semantic information and learn a precise code representation. In our work, we explore a novel code representation based on data and control flow extracted from LLVM **IR (intermediate representation)** [35], one type of intermediate code acquired from source code. Compared with the existing statement-based data and control flow representation method [3], we refine the variable-based flow graph construction to better describe the dependencies between code variables. Specifically, the graph nodes represent the tokens that appear in LLVM IR, and the edges represent the data and control dependencies between the tokens. Furthermore, we design an optimization mechanism while modeling the graph to remove the redundant information created by LLVM IR without changing the semantics. Finally, we employ an attentional gated graph neural network to embed the flow graph into a high-dimensional vector space to further perform code search tasks. Through this procedure, code multiple semantic features, i.e., tokens, variable-based data and control flow, can be simultaneously represented and accurately express the deep semantics of code.

[1] https://github.com.
[2] https://gitlab.com.

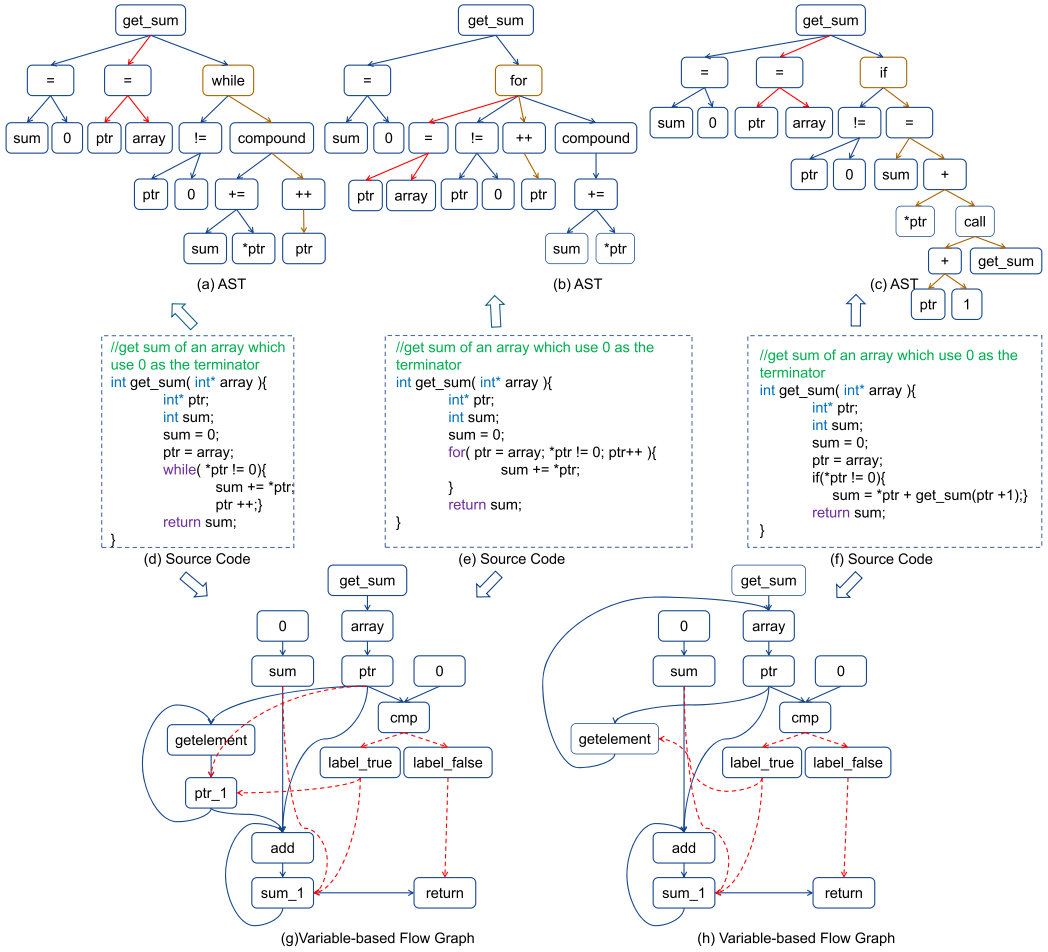


Fig. 1. The first illustrative example shows the code snippets with the same semantics and their corresponding ASTs and variable-based flow graphs. The difference between the ASTs is highlighted in red and yellow. The nodes in the variable based flow graph are tokens of the intermediate code and the edges either represent data dependencies (shown in solid line of light blue) or control dependencies (shown in dotted line of red).

To evaluate the effectiveness of our proposed model, we collect our dataset from GitHub containing 41,152 code snippets written in the C language and perform code search experiments. Experimental results show that DEGRAPHCS improves the top-1 code search hit rate from 34.05% to 43.05% when compared with the state-of-the-art methods. To simulate an actual code search scenario, we design an online code search tool, which takes 50 practical descriptions randomly chosen from the test set as candidate queries. For each query, five experienced participants manually label the relevant results they need returned by our proposed model DEGRAPHCS and three competitive approaches (i.e., DeepCS, UNIF and MMAN). The results of automatic evaluation and manual evaluation both confirm the effectiveness of DEGRAPHCS.

The main contributions of this paper are summarized as follows:

- We propose a novel semantic code representation method called DEGRAPHCS, which can integrate tokens, data flow, and control flow into a variable-based graph to represent the

semantic of the code more precisely than traditional approaches (e.g., AST). All of our code and data are available at https://github.com/degraphcs/DeGraphCS.

- We design a graph optimization mechanism to streamline the graph representation by reducing 51.88% of the redundant nodes, which significantly improves the DEGRAPHCS performance by 13.77% and 18.92% in terms of MRR and SuccessRate@1.
- We collect a large-scale dataset from GitHub containing 41,152 code snippets written in the C language and reproduce several competing code search models to make comparisons.
- We conduct experiments on the trained models, and the results of automatic evaluation demonstrate that DEGRAPHCS outperforms the state-of-the-art method (i.e., MMAN) by 19.14% and 26.43% in terms of MRR and SuccessRate@1. In addition, DEGRAPHCS achieves the best performance in our qualitative user study.

The remainder of this paper is organized as follows. In Section 2, we present two motivating examples. In Section 3, we first provide an overview of our proposed model and then describe the details of each part in our model. In Section 4, we describe the experimental setup and report the experimental results. In Section 5, we briefly review the related works. Finally, in Section 6, we conclude our study and present future work.

2 MOTIVATING EXAMPLE

Due to the diversity of syntax and programming styles of code, the gaps are widespread between syntax and semantics in code. The gaps bring the following challenges to traditional code representations: (1) The same semantic code is represented as different because of different code style, resulting in false negatives during searching code. (2) Different semantic code is represented as similar because of similar code style and model limitations, resulting in false positive during searching code. To overcome these challenges, we propose a semantic-based code representation method, which can eliminate gaps caused by syntax and code style and widen gaps caused by semantics. In this section, we show two motivating examples in Figures 1 and 2 to illustrate our semantic based code representation can represent programs better in dealing with two challenges.

Figure 1 shows three C example code snippets with the same semantics (sum the values in the specified array) and corresponding ASTs and VFGs (abbreviation of our proposed code representation: Variable-based Flow Graph). Figure 2 includes two C example code snippets with different semantics and corresponding ASTs and VFGs. AST consists of a basic syntax unit. For the convenience of observation, the difference between the ASTs is highlighted in red and yellow. Our variable-based flow graph is constructed from LLVM IR. Here, nodes in the flow graph are tokens of the intermediate code and the edges either represent data dependencies (shown in a solid line of light blue) or control dependencies (shown in a dotted line of red). Data dependencies of tokens refer to tokens that have dependencies through data processing statement. For example, in Figure 1, "`sum_1`" has data dependency with "`ptr`" because of the data processing statement "$sum+ = ptr$". Control dependencies of tokens refer to the execution of some tokens depends on other tokens. For example, in Figure 1, the execution of statement "$sum+ = ptr$" depends on the condition statement "$ptr! = 0$", and the execution of token "`sum_1`" also depends on the result of "`cmp`". Therefore, token "`sum_1`" has a control dependency relationship with token "`cmp`".

In Figure 1(d)–(f), we can see that the three codes with the same semantics have different written formats ("`for`", "`while`" and recursive loops separately), resulting in completely different ASTs in Figure 1(a)–(c). However, in Figure 1(g) and (h), when exploiting our variable-based flow graph, the three code snippets are represented almost the same. We can see that Figure 1(g) and (h) are very similar. In Figure 1(g) and (h), except for node "`ptr_1`" and "`ptr`", every node in Figure 1(g) can find a corresponding node in Figure 1(h). Compared with the node in Figure 1(g), the

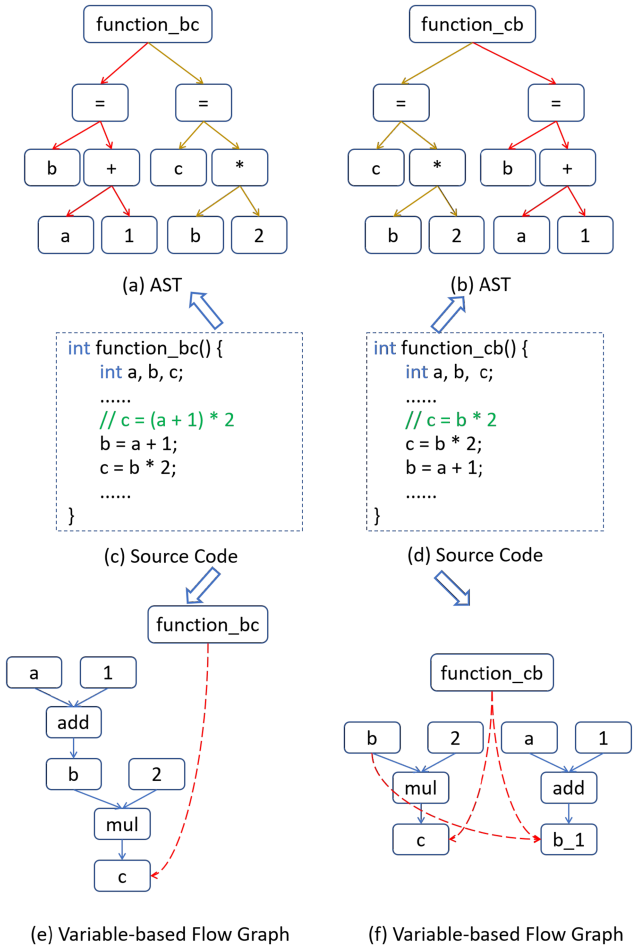


Fig. 2. The second illustrative example shows the code snippets with different semantics and their corresponding ASTs and flow-based graphs. The difference between the ASTs is highlighted in red and yellow. The nodes in the variable based flow graph are tokens of the intermediate code and the edges either represent data dependencies (shown in a solid line of light blue) or control dependencies (shown in a dotted line of red).

corresponding node has the same control dependency and data dependency. It illustrates that our VFG can represent different programs better.

In Figure 2(c) and (d), although the two code examples consist of the same statements, the difference of order will lead to different functionalities. In Figure 2(a) and (b), we can see that the only difference of two corresponding ASTs is the positions of the two subtrees (connected in red/yellow, respectively). In the existing works, such as MMAN, two corresponding ASTs obtain the same representation after applying Tree-LSTM [62]. Therefore, we can regard the representations of ASTs are almost the same. However, the corresponding variable-based flow graphs differ significantly in Figure 2(e) and 2(f), because of the underlying data dependencies between variables. It demonstrates that our variable-based flow graph can distinguish different semantic codes more accurately. The two examples show that our proposed variable-based flow graph representation can represent precise semantics of code while the syntactic structures fail.

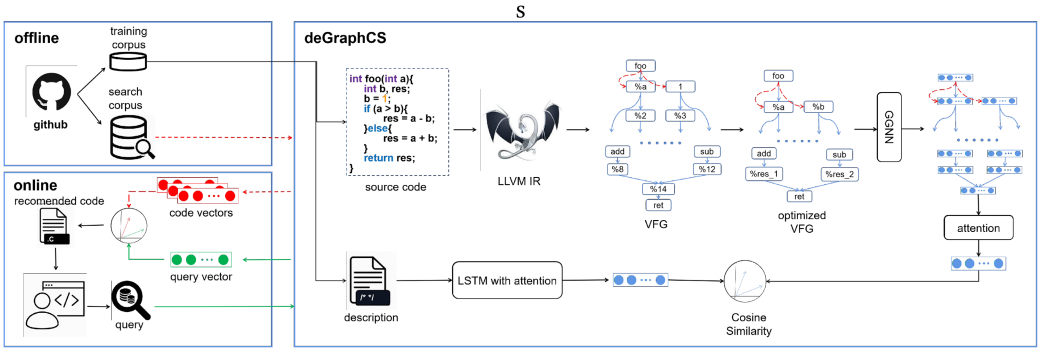


Fig. 3. The overall workflow of deGraphCS, containing offline data preparation, online inference, and the network architecture.

Inspired by the above two examples, we conclude that data and control dependencies can complement the drawbacks of structural feature-based code representation methods. However, the existing works [3, 61] extract data and control flows on the basis of statements, obtaining vector representations of code snippets by applying word embedding techniques such as skip-grams. A prominent problem is that coarse-grained statements usually cannot accurately capture the correlation between the code and query tokens. Therefore, we propose our fine-grained variable-based flow graph method to precisely model the relationship between tokens in code snippets.

3 THE PROPOSED MODEL

In this section, we first introduce an overview of our proposed network architecture. Then, we present our neural code representation mechanism, including the compilation background and LLVM IR, the variable-based flow graph building mechanism and the optimization mechanism. Finally, we present our comment description representation and model learning mechanism in detail.

3.1 An Overview

Figure 3 is an overview of the workflow of the deGraphCS model, which is composed of three parts: the upper left part denotes the offline data preparation process, the bottom left part denotes the online inference process, and the right part denotes the details of the network architecture. For the network architecture, deGraphCS first embeds the neural code and comments to the vector representations and then learns the relationship by minimizing the ranking loss function in the training process. We describe each part of the architecture in the following sections.

3.2 Neural Code Representation

For code representation, we first integrate data dependencies and control dependencies into graphs by analyzing different kinds of LLVM IR instructions. Specifically, we construct the data dependencies based on the address operation instructions (e.g., "`load`", "`store`") and the computation-related constructions (e.g., "`add`" and "`sub`"). In addition, the control dependencies are constructed based on the jump instructions (e.g., "`br`") and address operation instructions. The goal of code search is to better match the code semantics with the keywords in the queries, and excessive information may hinder the model from learning the fine-grained relationship between the source code and queries. Therefore, we propose several mechanisms to optimize the graph to decrease the noise in the model training process and improve the training efficiency. Finally, we feed the graph into a

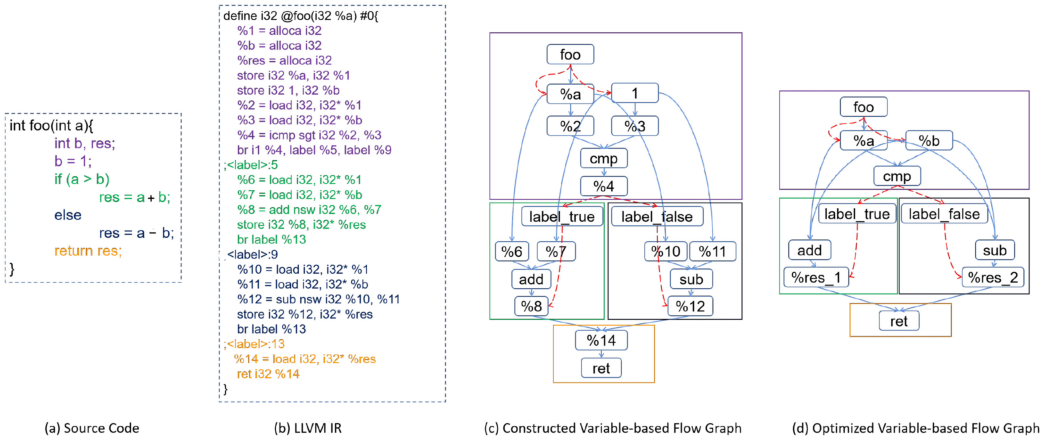


Fig. 4. An illustrative example shows a code snippet with its LLVM IR equivalent and our graph building as well as the optimization.

GGNN [40] with an attention mechanism to learn the vector representation of the code. To better explain our proposed neural code representation method, an illustrative example code associated with its IR and our variable-based flow graph building and optimizing result is shown in Figure 4.

3.2.1 Compilation and LLVM IR. Most popular compilers, such as LLVM and GCC, support multiple programming languages and hardware targets. To avoid duplications in code optimization techniques, the compilers require a strict separation among the source language, IR, and the target machine code, which is mapped to a specific hardware. LLVM IR supports various architectures and can inherently represent optimized code. Compared with source code, LLVM IR can help us analyze syntax and semantic of code conveniently. The IR in LLVM is given in **static single assignment (SSA)** form [13], which guarantees that every variable is assigned only once. SSA can help us to infer where the variables in code are defined and where they are used. As shown in Figure 4(b), LLVM divides the IR statements into several blocks represented by the corresponding labels shown in different colors. For the instructions, regarding the third line in "*label 9*", an instruction (IR statement) in our algorithm is composed of three parts: opcode ("sub"), operand (%10, %11), and result (%12).

3.2.2 Building Variable-based Flow Graph (VFG). To derive a precise semantic representation of the source code, we construct the graph at the variable granularity to capture source code token information. Specifically, we build data dependencies and control dependencies between variables to capture the data and control flow information from the source code. It is noted that our graph construction is an intra-procedural method since our dataset does not provide implementation details of called function. An illustrative example of the variable-based flow graph is shown in Figure 4(c), which is the initial graph constructed from the LLVM IR shown in Figure 4(b). The nodes in our graph can be variables, opcodes, or label identifiers, appearing in the figure as rectangles. Correspondingly, an edge either represents a data dependency (in blue solid line) or control dependency (in red dotted line). Given the LLVM IR, the whole graph building process is recorded in Algorithm 1. Specifically, we first extract the identifiers in each IR instruction as nodes. Then, we build data dependencies and control dependencies between nodes according to different types of instructions as follows.

Data dependency. Data dependencies exist in the computation-related constructions (e.g., "add", "sub") and the address operation instructions (e.g., "load", "store"). First, we build data dependencies according to instructions that are related to computation such as "add" and "sub".

ALGORITHM 1: Variable-based flow graph building process

Input: LLVM IR (shown in Figure 4(b)).
Output: the constructed variable-based flow graph (shown in Figure 4(c)).

```
for read the IR instructions by line do
    Case Computation instructions:
        Case "call/invoke":
            Build an edge parameters⟶ function name.
        Default:
            Build an edge operands ⟶ opcode.
            Build an edge opcode⟶ result.
    Case Address instructions:
        for each load instruction operated on addr (i.e., a = load addr) do
            value_list = SearchCFG(addr, inst).
            Build edge value_list ⟶ a.
        end for
        for all instructions "store x addr" do
            Connect x sequentially according to the order in CFG.
        end for
    Case "br" (i.e., br %val, label 1, label2):
        Build edge condition ⟶ labels.
        Build edge labels ⟶ "store" variables.
end for
```

For example, regarding the third line in the second block shown in green in Figure 4(b) ("%8 = add nsw i32 %6, %7"), we build data dependencies by linking the operands (%6 and %7) to the opcode ("add") and then linking the opcode ("add") to the result (%8) shown in the green square in Figure 4(c). It is noted that "nsw" is refer to "No Signed Wrap" which is related to the variable type. We overlook these keywords such as "nsw" since these keywords are not relevant to code semantics. We specifically deal with the "call/invoke" instruction since the operands are the parameters of the corresponding function. Specifically, when we build a graph for the current function (e.g., "get_sum"), we treat the function call instruction in two situations. First, if the called function is an external function that is not the current function ("get_sum"), we treat the name as opcode instead of "call/invoke" and link the operands to the called function name. Second, if the called function is the current function, it is regarded as a recursive call. Thus, we regard a node that links to the "return" node as the result of the called function, and we link the result to the nodes that use the called function. Moreover, we regard the input of the called function as passing to the parameter of the current function; thus, we link the input node to the parameter node. For example, if we construct data dependency for the function "foo(param1)", which have one parameter "foo(param1)" and use a statement "$res = foo(a)$" to call itself recursively in function body. We deal with the function call "foo(param1)" according to the second situation. Specifically, "return" will be linked to "res", which implies "foo" returns a value and assigns the value to "res". At the same time, we link variable "a" to "param1", which implies function "foo" inputs variable "a" into parameter.

Second, we build the variable data dependencies in "load" and "store" instructions. Specifically, when the variables need to be used, LLVM loads the corresponding values from the allocated address using the "load" instruction. Similarly, when variables need to be assigned new values, IR stores the new values in am address using the "store" instruction. The reason to treat "load" and "store" instructions separately is that in these two instructions, the address may store multiple values from different variables, and it is difficult to build a one-to-one mapping relationship

ALGORITHM 2: Search all variables from *addr* in *inst*

Input: *addr*, *inst*.
Output: value_list.

```
pre_list ← the last instructions pointed to inst.
for each instruction pre_inst in the pre_list do
    if pre_inst has been searched then
        continue
    else if pre_inst is "store x addr" then
        Append the variable x to the value_list.
    else
        list = SearchCFG(addr, inst)
        Append all the values in list to value_list
    end if
end for
```

between each address and variable. Therefore, we build data dependencies only between variables. For example, regarding the "`load`" instructions of the second block shown in green in Figure 4(b), although the variables %6 and %7 are loaded from addresses %1 and %*b*, the edges are connected from variables %*a* and 1, which are the true sources. To achieve this, we need to traverse all the "`load`" instructions and handle each instruction following the function in Algorithm 2.

Control dependency. Control dependencies could be found in the jump instructions (e.g., "`br`") and the address operation instructions (e.g., "`load`", "`store`"). We exploit address operation instructions since multiple variables stored in an address usually maintain a sequential order. Thus, we complete variable control dependencies through two aspects as follows:

First, we build control dependencies between variables and the condition identifiers. These conditions appear in condition jump instructions such as the "`br`" instruction, as shown in the last purple line in Figure 4(b). Label identifiers are the entries of basic blocks, and the condition determines which label to jump to. Thus, we construct control dependencies by linking the condition to all label identifiers. After that, to connect the whole graph, we build control dependencies between the label and the corresponding basic block variables. In our algorithm, we link the label identifier to the variables in the "`store`" instruction, as shown by the red line from label_true to %8 of Figure 4(c) because if a variable updates its value, a new value will be stored in the corresponding address by a store instruction. To avoid missing some important control dependency, we also link the label identifier to "`cmp`" opcode and "`function name`" in the instruction. Because the result of compare instruction or function call instruction may not be stored in the address.

Second, multiple assignments of the same variable will generate different variables to be stored in the address, and the variables usually maintain a sequential order. Therefore, we build control dependencies between these variables with the same address. As shown in line 13 of Algorithm 1, we need to traverse every store instruction which stores a variable to address **x**, and find the next "`store`" instruction that stores a new variable to address **x**, then build a connection from the previous variable to the latter variable until all the variables are connected in sequential order.

3.2.3 Optimizing Variable-based Flow Graph. After constructing the variable-based flow graph, we need to optimize the graph to decrease the noise and improve the training efficiency. The optimization method is composed of the following four steps.

(1) First, in LLVM IR, variables in the "`store`" instruction are named with numbers (e.g., %1, %2). Since the goal of code search is to better match the tokens (e.g., identifiers, function names) in code with the keywords in queries, excessive numbers may prevent the model from

filtering the critical information of the flow graph. Therefore, we replace the numbers with the corresponding variable names.

(2) Second, many opcodes are too trivial to represent the code semantics needed for code search. Therefore, we remove these opcode nodes from the graph by linking the predecessors of the opcode to their successors. Specifically, the trivial opcode can mainly be divided into three kinds. The first kind of opcode is that related to memory access and addressing since they operate on the variable address. The second is opcode related to conversion since they aim to transform the type of the data (e.g., change "`int`" type to "`string`" type). The third is opcode related to operations on exceptions, since the exceptions operations aim to monitor the status of variables at run-time, and are always not related to the function semantics.

(3) Third, in LLVM IR, many temporary registers are generated to store the intermediate values, and these registers have no corresponding variables. Thus, we remove these variable nodes by linking their predecessors to their successors.

(4) Fourth, we compress the control flow graph by merging the "`isolated`" blocks. In our constructed variable-based flow graph, assume block a is the predecessor of block b; if block a has only one successor and block b has only one predecessor, then we merge the two blocks to compress the control flow.

3.2.4 Graph2Vec. Since our constructed graph is a directed graph with multiple types of edges, we utilize an improved ***Gated Graph Neural Network*** (GGNN) with an attention mechanism to learn the vector representation of the code. GGNN is a neural network architecture for embedding graphs with multiple types of edges. In our graph $G = (V, E)$, V denotes a set of nodes(v, l_v), and E denotes a set of edges$(v_i, v_j, l_{(v_i,v_j)})$. l_v denotes the label of node v, which consists of the variables in the IR instructions. $l_{(v_i,v_j)}$ denotes the label of the edge from v_i to v_j, which includes two types: data dependency and control dependency.

GGNN learns the vector representation of G by the message passing mechanism as follows. First, we initialize each node $v \in V$ with a one-hot embedding vector (h_v^0) according to l_v. Then, we train the embeddings of all nodes through multiple iterations. In iteration t, each node v_i obtains message $m_t^{v_j \mapsto v_i}$ from neighbor v_j as $m_t^{v_j \mapsto v_i} = W_{l_{(v_i,v_j)}} h_{v_j}^{t-1}$, where $W_{l_{(v_i,v_j)}}$ is the weight matrix specified by the type of edges. It maps messages from neighbor v_j into a shared space. The weight matrix is learned during the training process. Then, all messages from the neighbors of v_i are aggregated in the following equation:

$$m_t^i = \sum_{v_j \in Neibour(v_i)} (m_t^{v_j \mapsto v_i}) \tag{1}$$

Then, GGNN uses **GRU (gated recurrent unit)** [12] to update the embedding of each node v_i. GRU uses 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, since different nodes contribute differently to the code semantics, we exploit the attention mechanism to calculate the importance of different nodes. We first allocate weights for each node v_i as:

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

where α_i denotes the weight of node v_i, $f(\cdot)$ denotes the linear layer, $\cdot$ denotes the inner project function, and u_{vfg} denotes the context vector, which is a high-level representation of all nodes in the graph. u_{vfg} is realized as a linear layer that is randomly initialized and jointly learned during training. Then, we obtain the embedding of the whole graph h_{vfg} as:

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

3.3 Comment Description Representation

For comment representation, we apply LSTM [27] to learn the corresponding representations. The embedding h_i^{des} of each word in the comment is calculated as $h_i^{des} = LSTM(h_{i-1}^{des}, w(d_i))$, where $i = 1, \ldots, |d|$, $|d|$ denotes the length of the comment description, and w denotes the word embedding layer to embed each word into a vector. Since different parts of the comment make different contributions to the final vector representation, we adopt the attention mechanism [1] to capture the fine-grained relevance between the hidden states and the final comment representation. Specifically, we apply the attention layer to calculate the attention score $\alpha^{des}(i)$:

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

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, which is a high-level representation of all tokens in the comment. The context vector u^{des} is randomly initialized and jointly learned during training. Then, the final representation of the comment description $E_{|d|}^{des}$ is calculated as:

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

3.4 Model Training

Now, we obtain all code representations (C) and description representations (D). To search the code precisely for each query, the model makes the code representation similar to the correct description representation and make the code representation different from the incorrect description representation. Specifically, for each code snippet representation $c \in C$, we regard the only associated description in D as the correct description d^+, and during each training iteration we randomly choose one description from other descriptions($D - d^+$) as incorrect description d^-. Through removing the semantically duplicated code and randomly selection, the probability that d^+ and d^- have the same semantics is reduced to a very low level. To make the vector representation of the pair $< c, d^+ >$ similar and the vector representation of the pair $< c, d^- >$ different, we train the model by minimizing the loss function $L(\theta)$ in the formulation of:

$$L(\theta) = \sum_{c \in C} \sum_{d^+, d^- \in D} max(0, \beta - cos(c, d^+) + cos(c, d^-)) \tag{6}$$

where θ denotes the model parameters, d^+ denotes the correct description representation, and d^- denotes the incorrect description representation. The $cos(,)$ function measures the cosine similarity between two vector representations, and β denotes the constant margin.

4 EXPERIMENTS AND RESULTS

In this section, to evaluate the code search performance of deGraphCS, we perform several experiments to answer the following research questions:

- RQ1: How effective is our proposed deGraphCS?
- RQ2: How effective is our VFG representation compared with other code representation on code search task?
- RQ3: Is our VFG representation the better choice to integrate tokens, data flow and control flow, compared with existing attention-based multi-modal learning method?
- RQ4: How does the attention mechanism in GGNN affect the performance of the model?
- RQ5: How does our graph optimizing mechanism affect the final retrieval performance?

- RQ6: How does deGraphCS perform when varying the comment length, code length, VFG node number, and longest path length of VFG?
- RQ7: How does deGraphCS perform for helping developers in actual programming?

RQ1 investigates whether deGraphCS outperforms the state-of-the-art deep code search models. RQ2 evaluates whether VFG representation outperforms other representations on code search task. RQ3 verifies that using VFG representation is better than using multi-modal learning to fuse the token, data flow, and control flow information. RQ4 verifies the effect of attention mechanism in GGNN. RQ5 investigates how our graph optimizing mechanism improves the training results. RQ6 tests and verifies the robustness of our proposed model when varying the comment length, code length, VFG node number, and longest path length of VFG. RQ7 evaluates the performance of deGraphCS compared with the state-of-the-art models in manual evaluation.

4.1 Experimental Setup

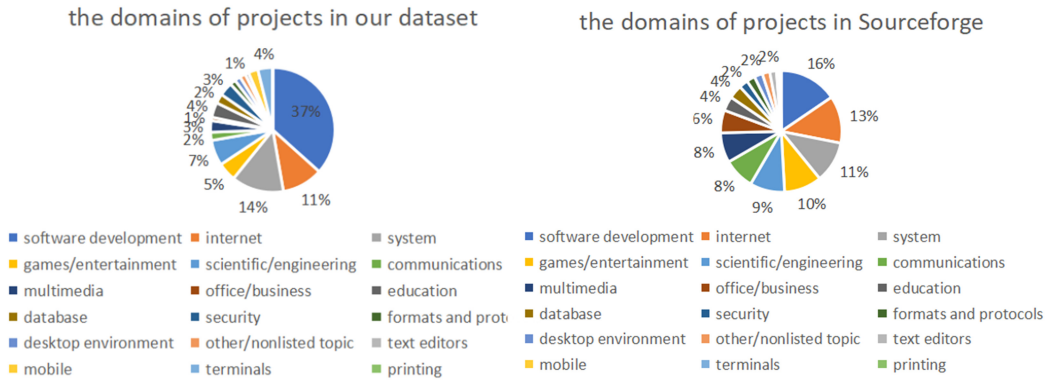
Here, we first describe our experimental dataset and present three widely used evaluation metrics. Then, we describe the implementation details and introduce the baseline models for comparison.

4.1.1 Data Collection. As described in Section 3, our deGraphCS model needs a large-scale training corpus that contains sufficient code fragments and the corresponding comment descriptions. Unfortunately, we cannot obtain access to the datasets collected by the existing works such as MMAN and UNIF. We also considered the dataset released by DeepCS, even though this dataset only contains cleaned Java code. It is hard for us to generate the LLVM IR without raw data. Therefore, to verify the performance of our proposed VFG representation method and code search model, we re-construct a corpus of C code snippets, which were crawled from GitHub. We chose C code snippets because the C language is popular and LLVM IR has been widely used on the C language.

To build the required dataset, we collected high-star C projects from GitHub (a popular open source project hosting platform). Then, we collected our dataset by selecting the C methods that contain the corresponding comment descriptions and can be compiled into LLVM IR from projects. For each C method c, we treat the first sentence that appears in the comment as the corresponding natural language query q since it typically describes the functionality implemented by the method [29, 31]. To verify the first sentence appeared in the comment contains the program semantic information, we randomly selected 100 pairs of query and code from our extracted dataset, and we asked three people to check. In the result, we find most (93%) of program semantic can be found in the first sentence. To reduce as many bad comments as possible, we use regular expressions to delete some comments that are not related to the method function. For example, we delete the comments which are start with "`Copyright`" since these comments always describe the copyright information of source code. Furthermore, we filter the (q, c) pairs by the following rules:

- (q, c) are filtered out if the code snippet c is a constructor or a test method.
- (q, c) are filtered out if the length of the query q is fewer than 3 words or longer than 30 words. The query which is fewer than 3 words is filtered out since we do not expect such a query to be informative. The query which is longer than 30 words is filtered out since such a query is rare and usually not related to the code function.
- (q, c) are filtered out if the length of the code snippet c is fewer than 5 lines or longer than 30 lines. The code which is fewer than 5 lines or longer than 30 lines is filtered out since such a code is rare, and the model training time will greatly increase.
- If a (q, c) pair appears multiple times in the dataset, we remove the duplication.

After collecting the corpus of commented code snippets, we then extract LLVM IR for our proposed model and other source code features for the baseline models, i.e., method name, tokens,



(a) the domains of projects in our dataset (b) the domains of projects in SourceForge

Fig. 5. The distribution of project domains.

AST and S-CFG. Finally, we obtained 41,152 samples, which is more than the 28,527 samples in MMAN [64]. Since the 41,152 samples are distributed in 1,554 open source projects, we believe it is general enough to train our model. To verify the generality of our dataset, we recruited three experienced graduated students from our school to investigate the domains of all projects in our dataset. They spent two days reading official documents and classifying each project in our dataset. Each project is classified into 18 categories according to the categories of projects in SourceForge.[3] As shown in the Figure 5, we can see that projects in our dataset cover all categories and have similar distribution with domains of projects in SourceForge. Therefore, we believe that the domains of our dataset is general. Following [64], we shuffle our dataset and split it into a training set with 39,152 pairs and a test set with 2,000 pairs. In the experiments, we utilized one correct candidate and 1,999 distractors to test models. To avoid bias resulting from evaluation on isolated datasets, we resort to automatic evaluation metrics on corpora with the ground truth.

Furthermore, to perform the manual evaluation, we first randomly select 100 descriptions from the test set, and then we carefully choose the 50 descriptions that are the easiest to understand. We rewrite the 50 descriptions (e.g., add some conjunction) as test queries to ensure that they are similar enough to real-world user queries. We construct a search codebase containing 30,799 C code snippets without the training samples to guarantee fairness. Five experienced participants select the relevant results returned by each model and record the scores.

4.1.2 Evaluation Metrics. We choose two common metrics to measure the code search performance: SuccessRate@k and **Mean Reciprocal Rank (MRR)**.

For the automatic and manual evaluations, we adopt both SuccessRate@k and MRR to assess the performance of a code search model with respect to a set of queries. SuccessRate@k represents the percentage of queries for which more than one correct snippet exists in the top k ranked snippets returned by a search model, which is calculated as: SuccessRate $@k = (\frac{1}{|Q|}\sum_{q=1}^{Q}\delta(\text{Rank}_q \leq k))$, where Q denotes the set of queries in our automatic evaluation, $Rank_q$ denotes the highest rank of the hit snippets in the returned snippet list for the query, and $\delta()$ denotes an indicator function that returns 1 if the rank of the q_{th} query ($Rank_q$) is smaller than k and returns 0 otherwise. SuccessRate@k is important because a better code search engine will allow developers to find the desired snippet by inspecting fewer results. Following MMAN, we evaluate SuccessRate@1,

[3] https://sourceforge.net/directory/.

SuccessRate@5, and SuccessRate@10; a higher SuccessRate@k value implies better performance of the code search model.

We also use MRR to measure the search result ranking of each model. MRR is the average of the reciprocal ranks of all queries Q. The reciprocal ranks are the inverse of the highest rank of the hit code, i.e., rank. The computation of MRR is: $\text{MRR} = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{\text{Rank}_q}$, where Q denotes the set of queries in the automatic evaluation, and $Rank_q$ denotes the rank of the ground-truth code corresponding to the q_{th} query. The higher the MRR value is, the better the code search performance. It is noted that we use following formula to evaluate the improvement of performance.

$$improvement = \frac{P_{new} - P_{old}}{P_{old}} \tag{7}$$

In the equation, the P_{new} is the performance of improved model, the P_{old} is the performance of origin model.

4.1.3 Baseline Models. We compare the effectiveness of DEGRAPHCS with three state-of-the-art deep learning-based code search methods:

- **DeepCS:** DeepCS [20] is a deep code search engine using deep neural networks. Instead of matching text similarities as traditional works, DeepCS learns a unified vector representation of both source code and the corresponding natural language query. We use the official implementation provided by the authors.[4]
- **CARLCS-CNN:** CARLCS-CNN [58] uses a a co-attention mechanism to improve the neural networks of DeepCS. Generally, CARLCS-CNN learns a correlation matrix between embedded code and query, and co-attends their semantic relationship.
- **Self-Attention:** Self-Attention is a baseline model used by [31]. It encodes code and query with BERT [16] encoder and then computes a similarity score between the representations of code and query. We use the official implementation by the authors. In self-attention, the model uses same structure of neural network as the pre-training model in BERT [16].
- **UNIF:** UNIF [7] is a supervised extension of the base NCS technique [56]. UNIF maintains significantly lower complexity than previous sequence-of-words-based networks by using a bag-of-words-based network.
- **MMAN:** MMAN [64] is a novel multimodal neural network for code search. MMAN uses an attention mechanism to incorporate multiple features, including code tokens, AST, and S-CFG, to learn a more comprehensive representation for code understanding.

In addition, we combine our representation method with current popular pre-training techniques. We use two typical pre-trained models, i.e., CodeBERT [18] and CodeT5 [69], and add fine-tune those models using our dataset.

To answer RQ2 (how effective VFG representation, compared with other code representation on code search task), we compare our method with different code representations on code search task. To eliminate the influence of the model, all the code representations are embedded into vector by GGNN. The details of different code representations are as follows:

- **CFG:** CFG is the control flow graph of source code, which consists of statements and control flow relationships between statements. We refer to MMAN [64], use LLVM to generate the CFG of the code, and embed the CFG into a vector with GGNN.
- **AST:** AST is abstract syntax tree of source code. We use LLVM to generate AST.

[4] https://github.com/guxd/deep-code-search.

- **FA-AST:** FA-AST is graph representation of programs called flow-augmented abstract syntax tree. [68] constructed FA-AST by adding extra control and data flow edges to augment original AST. Referring to the construction mechanism of [68], we construct FA-AST of C language code.
- **IVFG:** IVFG is a graph representation of programs called interprocedural value-flow graph. [61] used the statement of LLVM IR as node of graph and used Andersen's pointer analysis to construct the edge of graph. Followed by [61], we construct IVFG of our dataset. To better migrate IVFG to code search, we add variable names on the nodes instead of statements in [61].

To answer RQ3 (how effective is our graph-based integration compared with multimodal attention) and RQ4 (how does graph optimization in deGraphCS affect its effectiveness), we compare deGraphCS with some of its variants as follows:

- **MMAN (Token+V-DFG+V-CFG):** In this variant, we exploit the features of deGraphCS, i.e., code tokens, variable-based data and control flow graphs, and fuse them in a multimodal neural network with an attention mechanism. In other words, the features used in MMAN are replaced with the features used in deGraphCS. This variant is used to validate that our graph building and optimizing mechanism is more effective than the previous multi-modal incorporation mechanism.
- **deGraphCS-noGO:** In this variant, we remove the graph optimization mechanism from deGraphCS. This variant is used to validate that it is necessary to remove the redundant information in the initial constructed graph and verify how this mechanism can improve the training results.

4.1.4 Implementation Details. To keep more semantic information in source code, we set LLVM compiler optimization level into "O0". We use LLVM compiler 9.0 and add argument "-fno-discard-value-names" to retain variable name in source code. To train our proposed model, we first shuffle the training data and set the mini-batch size to 16. We build two separate vocabularies for comments and LLVM IR tokens and limit their vocabulary size to 10,000 and 15,000, respectively, to store the most frequently appearing tokens in the training dataset. For each batch, comments are padded to the maximum length with a special token "*PAD*", 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 the LSTM and GGNN units, we set the hidden size to 512. In addition, we set five rounds of iteration for GGNN. The margin is set to 0.6. We update the parameters via the AdamW optimizer [43] with a learning rate of 0.0003. It is noted that we follow [64] and do not fine-tune the hyper-parameters of the GGNN and LSTM model. We mainly fine-tune other hyper-parameters like the learning rate of model. For example, we set value of learning rate from 0.01 to 0.0001, and observe the performance changes of the model. We choose the best model performance which is stable in a certain range of values. All the models in this paper are trained for 200 epochs. All the experiments are implemented using the PyTorch 1.6 framework with Python 3.6, and the experiments are conducted on a server with one NVIDIA Tesla V100 GPU running on Ubuntu 18.04.

4.2 Experimental Results

4.2.1 RQ1: Comparison with Baselines. RQ1 investigates whether deGraphCS outperforms the state-of-the-art deep code search models. We evaluate deGraphCS on the test set, which consists of 2,000 pairs of code snippets and the corresponding descriptions. In this automatic evaluation, we treat each description as an input query, and its corresponding code snippet as the ground

Table 1. Comparison of the Overall Performance Between our Model and Baselines on Automatic Evaluation Metrics

Method	R@1	R@5	R@10	MRR
DeepCS	0.2350	0.4185	0.5045	0.3268
UNIF	0.3250	0.5175	0.5980	0.4193
CARLCS-CNN	0.2560	0.4825	0.5670	0.3517
Self-Attention	0.3965	0.5910	0.6570	0.4905
MMAN	0.3405	0.5325	0.6130	0.4342
deGraphCS	**0.4305**	**0.6175**	**0.6810**	**0.5173**
CodeBERT_FT	0.4595	0.6616	0.7421	0.5545
CodeBERT_deGraphCS	0.4647	0.6889	0.7584	0.5680
CodeT5_FT	0.5889	0.7637	0.8189	0.6694
CodeT5_deGraphCS	**0.5974**	**0.7668**	**0.8242**	**0.6758**

truth. We report our evaluation results in Table 1. Columns R@1, R@5, and R@10 show the values of SuccessRate@k over all queries when k is set to 1, 5, and 10, respectively. The column MRR presents the MRR value of each model.

From Table 1, we can clearly find that deGraphCS beats existing code search methods on all the metrics. Specifically, deGraphCS obtains an MRR of 51.73%, which is better than that of DeepCS (32.68%), UNIF (41.93%), CARLCS-CNN (35.17%), Self-Attention (49.05%), and MMAN (37.97%). For SuccessRate@k, deGraphCS improves the state-of-the-art R@1 score from 34.05% (obtained by MMAN) and 39.65% (obtained by Self-Attention) to 43.05%. For 43.05%/61.75%/68.10% of the test queries, the relevant code snippets can be found within the top 1/5/10 returned results by deGraphCS. The results indicates the effectiveness of deGraphCS.

In addition, to combine our representation method with current popular pre-training techniques, we use CodeBERT [18] and CodeT5 [69] models to obtain vector embeddings of query and code. CodeBERT generates high-quality text and code embeddings by pre-training on the CodeSearchNet [31] corpus with two tasks, masked language modeling and replaced token detection. CodeT5 is a unified pre-trained encoder-decoder transformer model that better supports both code understanding and generation tasks and allows for multi-task learning. To fuse our semantic map in the fine-tuning stage, we transform the semantic map into sequences via a depth-first search algorithm, which is used as additional information for code representations. 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 depth-first search traversal of the semantic graph as input. The data used in the fine-tuning phase is the same as the DeGraphCS training data. The results are listed in the second row of Table 1. Among them, CodeBERT_FT and CodeT5_FT fine-tune the pre-trained model on our data, CodeBERT_deGraphCS and CodeT5_deGraphCS add semantic graph sequences as additional information in fine-tuning. The results show that: (a) the pre-trained model greatly improves the performance of code search compared to other deep learning models (e.g., CodeT5_FT can achieve more than twice as much performance in MRR compared to DeepCS); (b) adding our semantic graph representation of IR can get additional enhancement even in a simple way during the fine-tuning stage. It would indicate a promising direction of understanding source code by combining intermediate representation and pre-training techniques. We leave a deep investigation in the future work.

4.2.2 RQ2: Effects of VFG Representation. To demonstrate VFG is a better representation than other representation (AST, FA-AST, CFG, IVFG) on the code research task, we feed different

Table 2. Effect of VFG Representation Compared to other Typical Representations

Method	R@1	R@5	R@10	MRR
CFG	0.0945	0.2010	0.2450	0.1455
AST	0.2435	0.4185	0.4910	0.3276
FA-AST	0.2370	0.4115	0.4875	0.3222
IVFG	0.3025	0.4550	0.5710	0.3984
deGraphCS	**0.4305**	**0.6175**	**0.6810**	**0.5173**

Table 3. Effect of Graph Integration

Method	R@1	R@5	R@10	MRR
MMAN(Token+V-DFG+V-CFG)	0.3380	0.5250	0.5990	0.4277
deGraphCS	**0.4305**	**0.6175**	**0.6810**	**0.5173**

representations into the same code search model (i.e., the improved GGNN in this paper). Table 2 shows the performance of different representations.

Our results show that VFG is much better than other representations. In terms of SuccessRate@1, the performance of deGraphCS is almost twice the performance of AST and FA-AST. The CFG representation is the worst, it is less than the half of the performance of AST and FA-AST. The IVFG is based on instructions of LLVM IR, we can see that our variable based method is better than IVFG. In terms of other metrics, VFG also performs better than other representations. For MRR, the improvements to CFG, AST, FA-AST, IVFG are 255.53%, 57.91%, 60.55%, and 29.84%, respectively. The results show that VFG can capture code semantics more accurately.

4.2.3 RQ3: Effects of Integration. To evaluate the advantage of our approach in the aspect of integrating tokens, data flow and control flow, we perform experiments on two models, i.e., deGraphCS and MMAN (Token+V-DFG+V-CFG). deGraphCS integrate token, data flow and control flow into one graph, and use GGNN to obtain the representation of graph. MMAN (Token+V-DFG+V-CFG) use multi-modal to obtain different representations of token, data flow and control flow, and fuse different representations into one representation. Table 3 shows the performance of the two approaches.

In Table 3, we can observe that the MRR of MMAN (Token+V-DFG+V-CFG) decreases by 8.96% compared with deGraphCS. In terms of SuccessRate@k, MMAN (Token+V-DFG+V-CFG) achieves a SuccessRate@1/5/10 of 33.80%, 52.50%, and 59.90%, respectively, much lower than those of deGraphCS. This means that more relevant code snippets are returned by deGraphCS. Therefore, integrating the three features into one graph is a better choice, instead of roughly fusing them by a single attention layer.

4.2.4 RQ4: Effect of Attention Mechanism. To evaluate the advantage of attention mechanism compared with sum mechanism and mean mechanism, we perform experiments on three models, i.e., deGraphCS(attention), deGraphCS(sum) and deGraphCS(mean). deGraphCS(attention) utilize attention network to learn different weights for nodes' vectors. deGraphCS(sum) sums all nodes' vectors and deGraphCS(mean) averages all nodes' vectors.

Those mechanisms are all used to integrate the nodes' information into one graph vector. From Table 4, we can see that deGraphCS(attention) achieves the best performance on R@1, R@5, R@10, and MRR. We believe that the attention mechanism can focus on the most important nodes in graph and abandon useless nodes. However, the sum mechanism and mean mechanism may confuse important information.

Table 4. Effect of Attention Mechanism

Method	R@1	R@5	R@10	MRR
deGraphCS(mean)	0.3380	0.5425	0.6210	0.4516
deGraphCS(sum)	0.3315	0.5275	0.5980	0.4234
deGraphCS(attention)	**0.4305**	**0.6175**	**0.6810**	**0.5173**

Table 5. Effect of Graph Optimization

Method	R@1	R@5	R@10	MRR
deGraphCS-noGO	0.3620	0.5585	0.6210	0.4547
deGraphCS	**0.4305**	**0.6175**	**0.6810**	**0.5173**

4.2.5 RQ5: Effect of Graph Optimization Component. To demonstrate the effectiveness of the flow graph optimizing mechanism constructed from LLVM IR, we perform experiments by comparing deGraphCS with the version with the graph optimization mechanism removed deGraphCS-noGO. We present the overall comparison results in Table 5.

In Table 5, we can see that deGraphCS-noGO achieves an average success rate of 36.20%, 55.85%, and 62.10% when the top 1, 5, and 10 results are inspected, respectively. In contrast, deGraphCS achieves average success rate of 43.05%, 61.75%, and 68.10%. The result shows that deGraphCS returns more relevant code snippets than deGraphCS-noGO. For SuccessRate@1, SuccessRate@5 and SuccessRate@10, the improvements to deGraphCS-noGO are 18.92%, 10.56%, and 9.66%. For MRR, the improvement to deGraphCS-noGO is 13.77%.

The results demonstrate that removing redundant information and optimizing the node contents of the initial constructed flow graph focuses our proposed model more on the useful and fine-grained correlations between the source code and the comment descriptions. In addition, we performed statistical analysis and found that the total number of nodes in the optimized graph decreases by 51.88% compared with graph without optimization, which decreases training time by approximately half.

4.2.6 RQ6: Model Robustness. To analyze the sensitivity of deGraphCS, we explore four parameters (i.e., comment length, code length, number of VFG nodes, the length of the longest path in VFG) that may have an impact on the code and comment representation. The comment length reflects the complexity of comment, and code length, number of VFG nodes, the length of the longest path in VFG both reflect the complexity of source code. The length of the longest path in VFG is the maximum distance between the pair of nodes in graph. Figure 6 illustrates the performance of deGraphCS based on different evaluation metrics with varying parameters. From Figure 6(a)—(c), we can find that in most cases, deGraphCS maintains a stable performance even though the comment length, code length or node number increases dramatically, which can be attributed to the superiority of our variable-based flow graph. The length of the longest path between any two nodes is used to measure the complexity of our flow graph. From Figure 6(d), we can see that as the length of the longest path increases (i.e., the difficulty of the graph embedding increases), the performance of deGraphCS decreases slightly but overall maintains a stable level. We can also see some zigzag behaviors in the four charts, but the fluctuation is very small in Figure 6(a)(b)(d). The reason why the fluctuation in Figure 6(c) is more obvious than other figures is that the length of the longest path can better reflect the code complexity than the number of VFG nodes. Therefore, we believe that the performance decreases slowly with the increase of code complexity or query complexity. Overall, the results in this subsection further verify the robustness of our proposed model.

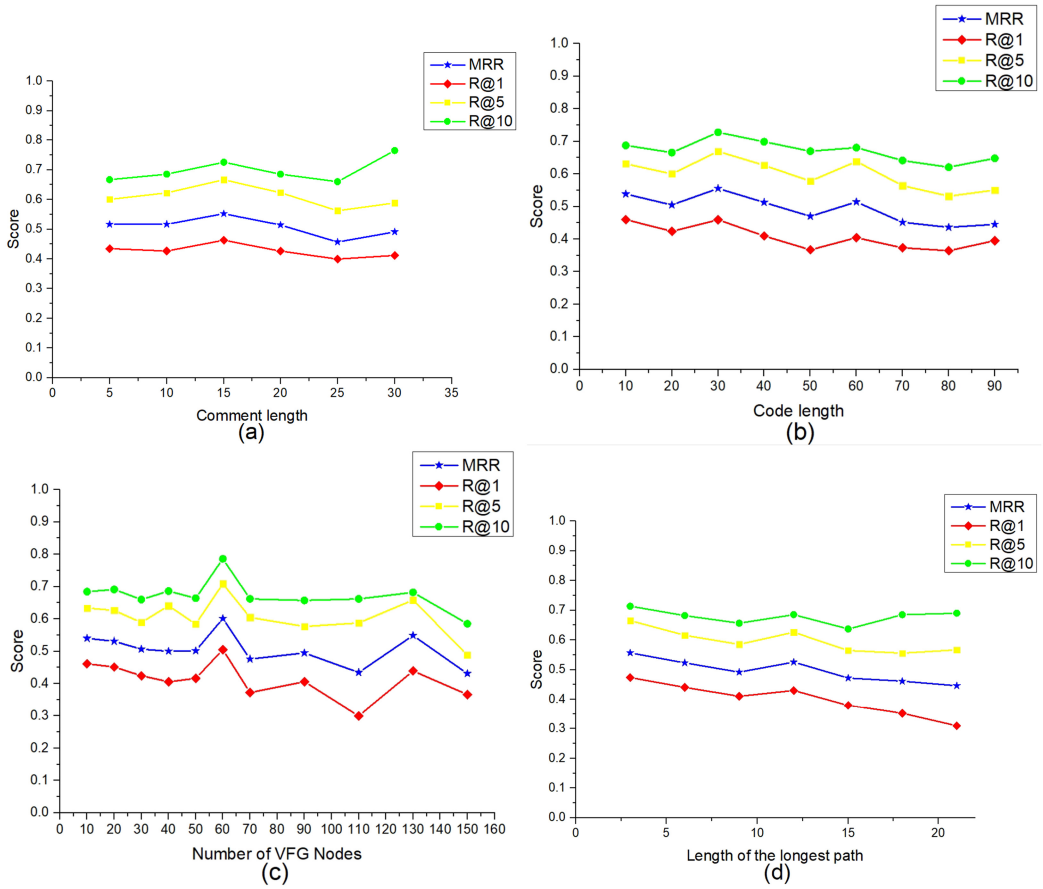


Fig. 6. Experimental results of deGraphCS on different metrics w.r.t. varying numbers of nodes and code lengths.

4.2.7 RQ7: Human Evaluation. deGraphCS shows great utility in the aforementioned automatic evaluation experiments. However, in reality, the questions of the usefulness of a returned code snippet are likely best answered by human programmers. Thus, we conduct a user study where five experienced programmers are asked to grade the utility of code fragments returned by the baseline methods, i.e., DeepCS, UNIF, MMAN, and our proposed model deGraphCS.

Specifically, we first build a search platform that includes a search codebase of 30,799 C language functions and 50 queries selected from the test set as benchmarks. By using different models, the platform will recommend the top 10 code snippets to users according to the query. Then, we recruit five graduate students from our school who have rich experience in C projects and are competent enough to perform the user study to evaluate the effectiveness of these models. To evaluate the models fairly, the five recruited students are not any of the authors.

To simulate real scenarios, we allowed the five participants (denoted as P1 to P5) to use the four models in turn to search the related code of 50 queries. All participants had over 2 years/5 projects in C programming experience. During the evaluation, we ensured that participants did not know which model returned the results. For each query, they inspected the top 10 results returned by each model and labeled those results they believed were relevant to the query. Table 6 shows the overall performance achieved by each model in terms of SuccessRate@10 and MRR.

Table 6. Comparison of the Overall Performance between our Model and Baselines on Manual Evaluation (SuccessRate@10|MRR)

Method	P1	P2	P3	P4	P5	Avg.
DeepCS	0.40|0.21	0.34|0.18	0.34|0.17	0.52|0.36	0.48|0.35	0.42|0.26
UNIF	0.52|0.36	0.52|0.27	0.48|0.29	0.60|0.39	0.58|0.39	0.54|0.34
MMAN	0.58|0.41	0.52|0.33	0.48|**0.37**	0.64|0.43	0.64|0.44	0.57|0.40
deGraphCS	**0.66|0.47**	**0.62|0.46**	**0.56**|0.36	**0.70|0.51**	**0.70|0.61**	**0.65|0.48**

```
alloc_fds(int **fds, int n_fds){
    if (n_fds) {
        *fds = (int *)malloc(sizeof(int) * n_fds);
        if (*fds == NULL) return OP_ERROR;
    }
    return OP_SUCCESS;
}
```

(a) deGraphCS

```
void cmd_mkexrc(frommark, tomark, cmd, bang, extra){
    int fd;
    ......
    fd = creat(extra, FILEPERMS);
    if (fd < 0){
        msg("Couldn't create a new \"%s\" file", extra); return;}
    savekeys(fd); saveopts(fd);
    ......
    close(fd); msg("Created a new \"%s\" file", extra);
}
```

(b) DeepCS

```
struct bmp_decdata *bmp_alloc(void){
    struct bmp_decdata *bmp = malloc_tmphigh(sizeof(*bmp)); return bmp;
}
```

(c) UNIF

```
int ffwopen(const char *fn, struct buffer *bp){
 int fd; mode_t fmode = DEFFILEMODE;
 ......
 fd = open(fn, O_RDWR | O_CREAT | O_TRUNC, fmode);
 if (fd == -1) {
  ffp = NULL; ewprintf("Cannot open file for writing : %s", strerror(errno));
  return (FIOERR);
 }
 ......
 return (FIOSUC);}
```

(d) MMAN

Fig. 7. An illustrative example shows the comparison between the searched results of (a) deGraphCS, (b) DeepCS, (c) UNIF, and (d) MMAN on the query "*allocate memory for the file descriptors*".

From Table 6, we can draw the following conclusions: (a) under the experimental setting, deGraphCS answers more user queries with an average SuccessRate@10 of 0.65, and the improvements compared with MMAN, UNIF, and DeepCS are 14%, 20%, and 55%, respectively; (b) deGraphCS achieves a better code search performance with an average MRR of 0.48. In conclusion, our proposed model maintains a higher practical value in the simulated code search scenario.

We further chose several examples to illustrate the superiority of deGraphCS on the search results. Figure 7 shows the searched results of deGraphCS, DeepCS, UNIF, and MMAN on the query "*allocate memory for the file descriptors*". We can see that the result returned by deGraphCS

is exactly what the users need. However, the results returned by DeepCS, UNIF, and MMAN do not realize the queried function and only focus on the shallow information (i.e., keyword "`fd`", which is related to file descriptor). Specifically, the core functionality related to the keywords in the query is outlined in red. The baseline methods can only retrieve the functions that realize file creation, open, or memory allocation for other data structures instead of file descriptors. In addition, Figure 8 shows the rank 1 and rank 2 searched results of deGraphCS on the query "`calculate checksum of checkpoint`". We can see that both retrieved code snippets realize the function of buffer checking and computing. However, compared with the rank 1 result, whose tokens are well named (i.e., checksum, buffer) and obviously matched with the keywords in the query, the variables in the rank 2 code snippet are much more obscure (i.e., cksum, b). Since the tokens in the rank 2 code are not named following natural language rules, it is hard for the models to utilize the token information for matching. The fact that deGraphCS can retrieve the obscure code snippet related to the query on the semantics demonstrates that the data and control flow features are essential in code search and that deGraphCS can fully exploit the useful information in our variable-based flow graph.

4.3 Threats to Validity

deGraphCS may suffer from three threats to validity. The first lies in the scalability of our proposed approach. The LLVM IR can only be extracted from a whole program with complete dependencies. Therefore, it is difficult to extend our precise code representation model to some sources where LLVM IR cannot be successfully extracted, such as many single code snippets in StackOverflow. The difficulty can be solved by automatically adding an interface for missing class objects and methods. We plan to overcome this compilation problem to generate more datasets in the near future. The second threat is that deGraphCS is currently trained and tested only on C programs. However, our work is based on the LLVM IR of the code, which is independent of the source programming language. Thus, our code representation method can be easily transferred to other languages such as Python and Java. For example, to transfer our work to the Java language, we can use Soot[5] (a Java optimization framework) to generate Jimple, which is an intermediate representation of Java. To transfer to the Python language, we can analyze and utilize the Python bytecode. We can build a variable-based flow graph based on the above intermediate representations. The third threat lies in the model generalization ability. We constructed a test dataset consisting of only 2,000 C code snippets, which may not be sufficient to represent most programming tasks. We plan to extend the dataset in the near future. The last threat is the reliability of dataset. According to our statistics, few (7%) of program semantics cannot be found in the first sentence of the comment, which would exert few negative impacts on our model.

5 RELATED WORK

5.1 Code Search

As vast repositories of open source code have become available, many works have been proposed to search code to help developers. There are many works that attempt to search code according to user queries. Traditional code search methods are mainly based on information retrieval and natural language processing technologies [2, 9, 28, 34, 39, 45–47], which consider source code text and structural characteristics. [2] proposes a code search engine Sourcerer that extracts fine-grained structural information from source code. [45] proposes CodeHow, a code search technique that reveals APIs related to user queries according to text similarity, and then applies an extended Boolean model to utilize API information in code searching. NCS [56] utilizes natural language processing technologies to embed the code and query into vectors, and then searches the code snippet

[5] https://github.com/soot-oss/soot.

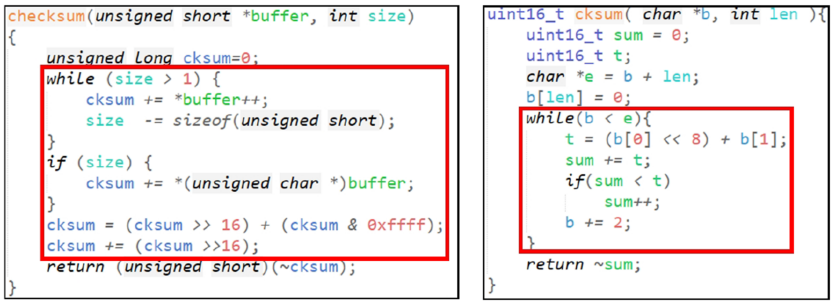


Fig. 8. The rank 1 and rank 2 searched results of deGraphCS on the query "*calculate checksum of checkpoint*".

by comparing the similarity of the vector representations. The above information retrieval-based methods treat both source code and query as natural language. It is difficult to deeply understand the semantic information of source code.

Since traditional code search based on syntactics often returns vague or irrelevant results, many works [36, 38, 55, 59, 67] have been proposed for semantic code search, which depends on specifications. For example, to search semantically related code, [55] proposes an architecture for semantics-based code search. The architecture utilizes many different specifications, including keywords, method signatures, and test cases. The semantic search based on specifications performs well for finding relevant code but requires developers to write complex specifications. To reduce the specification requirement, [59] proposes a new approach in which programmers are required only to specify lightweight, incomplete specifications that are in the form of input/output pairs and/or partial program fragments. Then, the approach uses an SMT solver to automatically identify programs. However, it also requires extra specifications to understand code semantics. Compared with the prior semantic code search, our method uses a static code-level analysis on source code, without the need to input extra specifications or run code snippets. Our work focuses on scenarios in which users can simply use natural language to describe their intent. Finally, we obtain code semantics by using a deep learning model to learn a code representation instead of depending on the input and analysis of specifications.

Many works have been proposed to enrich query information by refining queries, including query expansion and reformulation [17, 22, 25, 26, 37, 44, 66, 67]. Specifically, [22] trains a recommender (Refoqus) based on machine learning technologies, and Refoqus can recommend a reformulation strategy according to query properties. [44] utilizes synonyms generated from WordNet to extend queries. [67] utilizes user feedback to reformulate queries, and [66] generates semantic enriched queries using reinforcement learning.

Recently, to understand the deep semantics of the code and queries, deep learning technologies have been applied to code search [7, 10, 15, 20, 21, 24, 30, 56, 58, 64]. [10] proposes BVAE including two **variational autoencoders (VAEs)** to model source code and natural language, and jointly trains two models to capture the similarity between the latent variables of the code and description. Many works measure the semantic similarity of the source code and query through joint embedding and deep learning technologies. CODEnn [20] extracts code tokens, filenames and API sequences as the features and embeds this information and queries into a shared space so that code snippets could be retrieved by query vectors. CARLCS-CNN [58] uses a co-attention mechanism to improve the neural networks of DeepCS. UNIF [7] extends NCS and further fine-tunes code and query embeddings by joint deep learning. [31] proposes multiple models (Bag of Words (Neural BoW), RNN, 1D-CNN, and Self-Attention) to encode tokens and mine code semantic

for code search task. To utilize more source code information, [64] proposes MMAN, which uses a multimodal (tokens, AST and CFG) to represent source code. [23] proposes a multi-perspective cross-lingual neural framework and inputs the code sequence and AST sequence to train model. Similarly, we embed the source code and query into a common space to mine the semantic relationship. However, these works cannot precisely represent source code semantics.

5.2 Code Representation

Deep learning techniques for program analysis have attracted increasing attention. Many works focus on source code representation to perform further software engineering research [14, 32, 33, 48, 53, 54, 63, 65, 70, 71]. [54] adopts an RNN and n-gram model for code completion. To capture code structural information, [48] proposes a novel **tree-based convolutional neural network (TBCNN)** to represent source code ASTs. [70] proposes a framework (CDLH) that incorporates an AST-based LSTM to exploit lexical and syntactical information. More recently, to extract more information, [64] proposes MMAN to combine multiple semantic information of code, which includes source code tokens, AST, and CFG. These studies focus on different source code representations to capture the structural, syntactical and semantic information. However, they cannot precisely represent the code on semantics. Therefore, we focus on the code semantics and propose our variable-based flow graph method.

Several recent works [6, 11, 57, 60] have attempted to utilize intermediate representation to represent code. [60] constructs a dependency graph to represent code based on Java bytecode instructions. They use graphs to record data and control dependencies between instructions to detect code similarity by using a subgraph isomorphism algorithm to analyze the similarity of the dependency graph. [11, 61] both focus on building a graph based on LLVM IR to represent code. [11] uses a skip-gram model to learn the graph representation and apply to code task, [61] uses Andersen's pointer analysis to construct graph and represents code to the code summary and code category task. They achieve good performance by utilizing intermediate representation to represent code. [19] also utilizes the data dependency and control dependency of statements and constructs a dependency matrix. Through the dependency matrix, [19] improves the performance of code search. Different from these works, we aim to make a more precise code search; thus, we construct a different graph based on variables instead of instructions, which ensures that the granularity of both comments and code representations in code search are consistent. Compared with the previously used methods, we use a deep learning model (GGNN) to learn the code semantic representation. Furthermore, to obtain a more precise code representation, we optimize the graph to decrease the noise and improve the training efficiency.

Many works [49–52] have been proposed to mine Object/API usage. They recommended Object/API patterns to assist developers in real-time programming. These works are similar to the techniques of code search, which also retrieve useful code for code developers. Many Object/API usage miners represent code as graphs, and construct the graphs based on the usage orders of objects/API's actions. Then, the traditional methods like graph isomorphism algorithm or boolean model are used to mine object/API usage pattern on graph. Compared with the approaches of searching object/API usage, we focus on all useful tokens including API, variable names, and opcode names, since we believe all of these tokens are related to code semantic. Then we propose a novel variable-based graph construction method to represent control- and data- flow information.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a deep graph neural network named deGraphCS for code search. Instead of considering source code structural features such as AST, deGraphCS proposes a new

code representation method for code search task which can represent the semantics of code more precisely. Furthermore, we propose an optimization to remove the redundant information of the graph, followed by a gated graph neural network with an attention mechanism to capture the critical code information. In addition, we use a unified framework to learn the representation of natural language queries and corresponding code snippets. We conduct several experiments on trained models, and the results of automatic evaluation and manual evaluation both demonstrate that our proposed approach is effective and outperforms the state-of-the-art approaches.

In the future, we plan to overcome the compilation problems of code snippet and investigate the performance of DEGRAPHCS on other datasets of different programming languages, e.g., Python and Java. We aim to apply our variable-based flow graph construction method to represent more complicated code snippets precisely. We also plan to extend the variable-based flow graph we designed to solve other software engineering problems, e.g., code translation [11], API recommendation [5, 30], and code clone detection [71]. As the pre-trained model develops, we also plan to train a unified large-scale pre-trained model for multiple languages including the source code corpus of C, C++, Java, Python, and so on. Also, we would further explore the fine-tuning and prompt strategies to improve the performances of pre-trained models by combining the source code token and semantic-based graph data.

REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[2] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. 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*. 681–682.

[3] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 3585–3597. http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf.

[4] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522.

[5] Liang Cai, Haoye Wang, Qiao Huang, Xin Xia, Zhenchang Xing, and David Lo. 2019. BIKER: A tool for Bi-information source based API method recommendation *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3338906.3341174

[6] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada. 2020. Improving syntactical clone detection methods through the use of an intermediate representation. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. 8–14. https://doi.org/10.1109/IWSC50091.2020.9047637

[7] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. 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*. 964–974.

[8] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[9] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[10] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 826–831.

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2547–2557. http://papers.nips.cc/paper/7521-tree-to-tree-neural-networks-for-program-translation.pdf.

[12] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR* abs/1412.3555 (2014). arXiv:1412.3555 http://arxiv.org/abs/1412.3555.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[14] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[15] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779* (2018).

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Timothy Dietrich, Jane Cleland-Huang, and Yonghee Shin. 2013. Learning effective query transformations for enhanced requirements trace retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 586–591.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. 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 (Findings of ACL)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.), Vol. EMNLP 2020. Association for Computational Linguistics, 1536–1547.

[19] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. CRaDLe: Deep code retrieval based on semantic dependency learning. *Neural Networks* 141 (2021), 385–394.

[20] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[22] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 842–851.

[23] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A Multi-Perspective Architecture for Semantic Code Search. (2020). arXiv:cs.SE/2005.06980

[24] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. 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*. 163–174.

[25] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 524–527.

[26] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. 2014. NL-based query refinement and contextualized code search results: A user study. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 34–43.

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[28] Reid Holmes, Rylan Cottrell, Robert J. Walker, and Jorg Denzinger. 2009. The end-to-end use of source code examples: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 555–558.

[29] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[30] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.

[31] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436.

[32] Hamel Husain and Ho-Hsiang Wu. 2018. How to create natural language semantic search for arbitrary objects with deep learning. Retrieved November 5 (2018), 2019.

[33] Hamel Husain and Ho-Hsiang Wu. 2018. Towards natural language semantic code search. Retrieved November 5 (2018), 2019.

[34] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. 664–675.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[36] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology* 53, 4 (2011), 294–306. https://doi.org/10.1016/j.infsof.2010.11.009

[37] Otávio A. L. Lemos, Adriano C. de Paula, Felipe C. Zanichelli, and Cristina V. Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 212–221.

[38] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. 2007. CodeGenie: A tool for test-driven source code search. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 917–918. https://doi.org/10.1145/1297846.1297944

[39] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 690–701.

[40] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[41] Zhixing Li, Yue Yu, Tao Wang, Gang Yin, Shanshan Li, and Huaimin Wang. 2021. Are you still working on this? An empirical study on pull request abandonment. *IEEE Transactions on Software Engineering* (2021).

[42] Zhixing Li, Yue Yu, Minghui Zhou, Tao Wang, Gang Yin, Long Lan, and Huaimin Wang. 2020. Redundancy, context, and preference: An empirical study of duplicate pull requests in OSS projects. *IEEE Transactions on Software Engineering* (2020).

[43] Ilya Loshchilov and Frank Hutter. 2017. Fixing weight decay regularization in Adam. *CoRR* abs/1711.05101 (2017). arXiv:1711.05101 http://arxiv.org/abs/1711.05101.

[44] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[45] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective code search based on API understanding and extended Boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[46] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2011. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2011), 1069–1087.

[47] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.

[48] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2014. Convolutional neural networks over tree structures for programming language processing. *arXiv preprint arXiv:1409.5718* (2014).

[49] Sergio Mover, Sriram Sankaranarayanan, Rhys Braginton Pettee Olsen, and Bor-Yuh Evan Chang. 2018. Mining framework usage graphs from app corpora. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 277–289.

[50] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2012. GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1407–1410.

[51] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A graph-based approach to API usage adaptation. *ACM SIGPLAN Notices* 45, 10 (2010), 302–321.

[52] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 383–392.

[53] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969* (2015).

[54] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[55] S. P. Reiss. 2009. Semantics-based code search. In *2009 IEEE 31st International Conference on Software Engineering*. 243–253. https://doi.org/10.1109/ICSE.2009.5070525

[56] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.

[57] A. Schäfer, W. Amme, and T. S. Heinze. 2020. Detection of similar functions through the use of dominator information. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 206–211. https://doi.org/10.1109/ACSOS-C51401.2020.00057

[58] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *28th International Conference on Program Comprehension (ICPC)*.

[59] K. T. Stolee. 2012. Finding suitable programs: Semantic search with incomplete and lightweight specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. 1571–1574. https://doi.org/10.1109/ICSE.2012.6227034

[60] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 702–714. https://doi.org/10.1145/2950290.2950321

[61] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428301

[62] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075.

[63] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.

[64] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[65] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[66] Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. 2022. Enriching query semantics for code search with reinforcement learning. *Neural Networks* 145 (2022), 22–32.

[67] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 677–682.

[68] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[69] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708.

[70] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*. 3034–3040.

[71] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

Received 22 December 2020; revised 7 May 2022; accepted 20 May 2022