Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Intelligent code search aids edge software development

Fanlong Zhang[1], Mengcheng Li[1], Heng Wu[2*] and Tao Wu[3*]

Abstract

The growth of multimedia applications poses new challenges to software facilities in edge computing. Developers must effectively develop edge computing software to accommodate the rapid expansion of multimedia applications. Code search has become a prevalent practice to enhance the efficiency of the construction of edge software infrastructure. Researchers have proposed lots of approaches for code search, and employed deep learning technology to extract features from program representations, such as token, AST, graphs, method name, and API. Nevertheless, two prominent issues remain: 1) there are only a few studies on the effective use of graph representation for code search (especially in Java language), and 2) there is a lack of empirical study on the contributions of different program representations. To address these issues, we conduct an empirical study to explore program representations, especially program graphs. To the best of our knowledge, this is the first attempt to conduct code search with mixed graphs representation for Java language, containing the control flow graph and the program dependence graph. We also present a hybrid approach to capture and fuse the features of a program with representations of *Token*, *AST*, and *Mixed Graphs (TAMG)*. The results of our experiment show that our approach possesses the best ability (R@1 with 37% and R@10 with 67.1%). Our graph representation exhibits a positive effect, and the token and AST also have a significant contribution to the code search. Our findings can aid developers in efficiently searching for the desired code while constructing the software infrastructure for edge computing, which is crucial for the rapid expansion of multimedia applications.

Keywords Cloud computing, Code retrieval, Multi-modal, Attention mechanism, Deep learning

Introduction

Given the rise of multimedia applications, such as video streaming and computer games, there is a need to efficiently construct software infrastructure in edge computing to address the diverse issues encountered by modern interactive media applications [1, 2]. Meanwhile, in the software engineering (SE) community, the research has entered the era of "Big code" with the assistance of open-source resources. Researchers have begun applying artificial intelligence (AI) technologies to software engineering tasks [3, 4] (such as code search [5]), in which developers retrieve an intent code snippet from websites during the development process. Therefore, code search not only helps developers substantially in boosting the productivity of edge computing development efficiency but also improves edge software quality and reliability by reusing high-quality source code [6].

In the initial stages of code search research, traditional technology was utilized to excavate intent codes from software repositories [7, 8]. Linstead et al. [7] employed IR techniques that incorporated source-specific heuristics to search for and discover reusable software

*Correspondence:
Heng Wu
heng.wu@foxmail.com
Tao Wu
doctorwutao@163.com
[1] School of Computer Science and Technology, Guangdong University of Technology, Guangzhou 510006, China
[2] School of Automation, Guangdong University of Technology, Guangzhou 510006, China
[3] Guangdong Provincial Corps Hospital of the Chinese People's Armed Police Forces, Guangzhou 510507, China

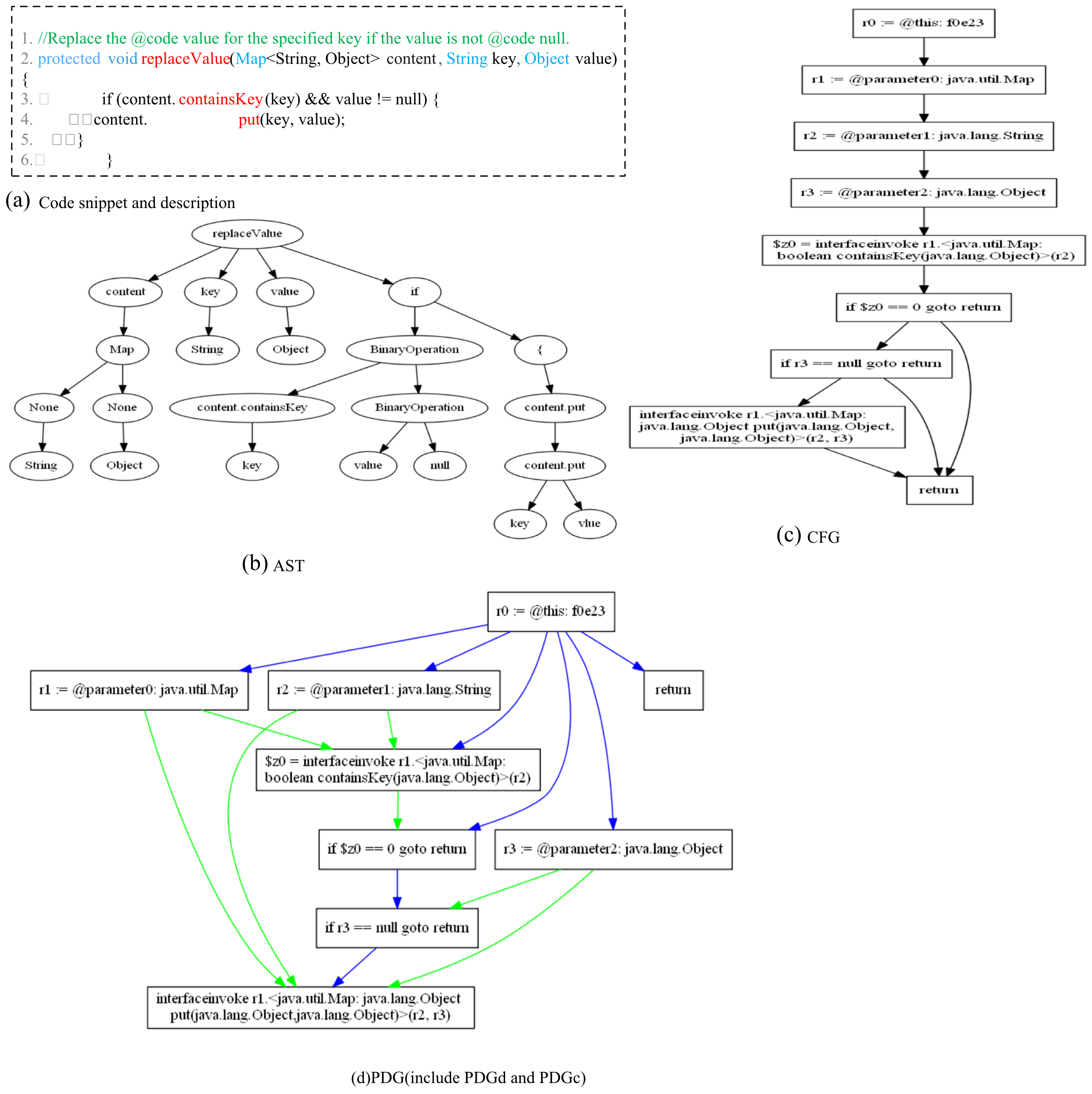
components. Mishne et al. [8] proposed an approach for answering queries focused on API usage based on static mining and temporal API specifications. However, these methods inevitably rely on expert knowledge, making it challenging to navigate the massive searchable code base. Consequently, researchers utilize deep learning technology to enhance the ability of code search, by learning semantic features from source code and matching the source code to the corresponding description. The first approach was proposed by Gu et al. [9], which represented the code snippets and the queries as vectors with a neural network. They captured code semantic information from method name, code token, and API from code snippets. Meanwhile, researchers proposed additional techniques for extracting semantic features from various program representations, such as token [10], tree [11], and graph [12]. Sachdev et al. [10] build neural code search tool (NCS) by using a combination of word embedding and TF-IDF techniques. Sun et al. [11] build a structure-sensitive model named PSCS based on the abstract syntax tree for code search. Ling et al. [12] proposed a method called deep graph matching and searching (DGMS), in which they realized a graph generation approach to represent query texts and source codes.

To enhance the ability of code search, researchers have made further attempts to extract enriched semantic features with multiple representations of the program. The forms of the representation can be method name, API, tokens, abstract syntax trees (AST), the program graph. Gu et al. [9] explored the method name, API invocation, and code tokens to embed code snippets for Java program language and their methods went beyond the traditional tools. Shuai et al. [13] improved the work of Gu et al. by extending different neural networks for encoding the same representation. Another attempt was made by Meng et al. [14], and that employed three independent encoders, including a lexical encoder for the token, a name encoder for the method name, and a structural encoder for AST. There are also some approaches that employ graph representation as well as token and AST representations, such as MMAN [15]. They proposed a deep model for semantic code search, that represented source code on token, AST, and CFG (control flow graph) only for C programming languages. However, in the current code search research, there is a lack of studies on program graph representations, especially for the Java programming language. Nonetheless, there are a number of graph representations for code, such as the control flow graph (CFG) and the program dependency graph (PDG). In Fig. 1, we provide an example of these graph representations. Figure 1a depicts a straightforward example of Java code and its natural language description. The generated AST and CFG are depicted in Fig. 1b and c, respectively. The PDG of the code is depicted in Fig. 1d, which incorporates control dependence flow (represented by blue lines) and data dependence flow (represented by green lines). This example demonstrates that multiple code graphs contain distinct semantic information. This motivates us to *investigate the efficacy of multiple program representations on code search, especially the graph representations.* There are two main differences between us and Wan et al., on the one hand, Wan et al. focus on the C programming language, while we pay attention to JAVA programming language; on the other hand We have additionally add PDG, with which CFG are combined into a mixed graph, compared with Wan et al. work with the token, AST, and CFG. More specifically, we list here our research issues that need to be addressed in the code search task:

- Whether graph representation of programs has a positive effect on code search for edge computing software?
- Whether each of these representations of the program has a positive contribution, and which ones provide a more significant effect?
- In practice, how do developers determine their representations, and how to train their models to obtain optimal performance to enhance the efficiency of edge software?

Therefore, in this work, we propose a hybrid approach to capture the features with several representations of *T*oken, *A*ST, and *M*ixed *G*raphs *(TAMG)* for code search, that explores the effectiveness of these representations for java programming language. The mixed graph representation contains control flow graph (CFG) and program dependence graph (PDG). Specifically, with the help of program analysis tools, we parse the source code into several representation forms including token, AST, and graphs. Then, we employ neural networks to learn the syntax and semantic features of each representation. We use long short-term memory (LSTM), tree-based LSTM, gated graph neural networks (GGNN), and recurrent neural network (RNN) to characterize token, AST, graph respectively. Furthermore, we also employ attention mechanism for each modality and use LSTM for the description. To evaluate the efficacy of our approach, we experiment on the dataset from three perspectives, including the effectiveness experiment, the significant experiment, and the performance experiment.

The results show that our hybrid approach possesses the best effectiveness, that combines token, AST, and mixed graph representations. Each of these representations plays a positive effect, and the employed ones in our approach have a significant contribution to code



```
//Replace the @code value for the specified key if the value is not @code null.
protected void replaceValue(Map<String, Object> content, String key, Object value)
{
        if (content.containsKey(key) && value != null) {
            content.put(key, value);
        }
}
```

(a) Code snippet and description
replaceValue
content
key
value
if
Map
String
Object
BinaryOperation
{
None
None
content.containsKey
BinaryOperation
content.put
String
Object
key
value
null
content.put
key
vlue
(b) AST
r0 := @this: f0e23
r1 := @parameter0: java.util.Map
r2 := @parameter1: java.lang.String
r3 := @parameter2: java.lang.Object
$z0 = interfaceinvoke r1.<java.util.Map: boolean containsKey(java.lang.Object)>(r2)
if $z0 == 0 goto return
if r3 == null goto return
interfaceinvoke r1.<java.util.Map: java.lang.Object put(java.lang.Object, java.lang.Object)>(r2, r3)
return
(c) CFG
r0 := @this: f0e23
r1 := @parameter0: java.util.Map
r2 := @parameter1: java.lang.String
return
$z0 = interfaceinvoke r1.<java.util.Map: boolean containsKey(java.lang.Object)>(r2)
if $z0 == 0 goto return
r3 := @parameter2: java.lang.Object
if r3 == null goto return
interfaceinvoke r1.<java.util.Map: java.lang.Object put(java.lang.Object,java.lang.Object)>(r2, r3)
(d)PDG(include PDGd and PDGc)

Fig. 1 An example of a program with different representations

search. When building their models, we recommend developers retain all of them if possible and employ the AST or graph while retaining at least the token. We believe that our approach can assist developers in building the software infrastructure for edge computing, a critical component for the accelerated expansion of multimedia applications.

Our contributions to this work are as follows:

- We conduct an empirical investigation on code search to examine the efficacy of various representations in assisting advanced edge software development in the field of multimedia applications.
- We propose a hybrid approach (TAMG) for code search to extract and fuse features of programs, especially the graph representations to enhance effectiveness.

- Through experimentation, we assess their effectiveness and answer proposed research issues. Our hybrid approach has the best ability and our mixed graph has a positive effect on code search. To achieve the best performance, we recommend the token, AST, and the graph when building the models, at least of AST or graph while keeping the token.

Our paper is organized as follows. Related work section is present some other works related to this paper. Methodology section describes the details of our approach. In Experimental methodology section, we show the details of how to design our experiment. Results and discussion section presents the evaluation of our experiment. Threats to validity section presents threats to the validity, including external and internal threats. To the end, we conclude our work in Conclusion section.

Related work

With the increasing popularity of multimedia applications like video streaming and computer games, it is necessary to develop software infrastructure in edge computing [1, 2]. Code search is a prevalent technique that adeptly addresses the diverse challenges encountered by modern interactive media apps. So, in this section, we provide the related work on code search and program representation learning. Program representation learning employs deep learning to extract semantic features for solving software engineering tasks, such as code search.

Code search

Code search/retrieval has become a common practice in software development, aiding engineers in enhancing productivity for software infrastructure in edge computing. Initially, researchers have utilized information retrieval (IR) technology to get the code snippets. For instance, Linstead et al. [7] developed a code search engine named *sourcerer* based on the code rank methodology. Mishne et al. [8] constructed the search index with a new method that statically mined code fragments and merged temporal API specifications. Lv et al. [16] developed a technique named *CodeHow* for locating potential API, and the experimental results demonstrated that it was effective. Ding et al. [17] creat a cloning search engine named *Kam1n0* by combining a new LST scheme and graph matching, that was accurate, efficient, and scalable for handling large amounts of code.

Following the program representing learning, researchers have recently proposed deep learning-based approaches for improving code search [3, 5]. It has been demonstrated that deep learning-based code search models, such as *DeepCS* [9], outperformed conventional code search techniques, such as *sorcerer and CodeHow*. These methods are also divided into the same three categories: token-based, tree-based, and graph-based. Taking the token-based method as an example, Sachdev et al. [10] build a search tool for large codebases called *neural code search (NCS)* to obtain a better result by adding a layer of supervision. Wang et al. [18] proposed a new deep learning tool called *COSEA*, which captured valuable code intrinsic structural logic for code search. Cheng et al. [19] proposed CSRS, consisting of an embedding module with n-gram embedding of queries and codes. Alternatively, the abstract syntax tree (AST) can provide more comprehensive semantic information. Thus, researchers also utilized AST to carry out program representation learning. Sun et al. [11] proposed *PSCS*, a path-based neural model for learning semantics and the structure code represented by an AST path, and their model showed a significant improvement in search performance compared to the current techniques. Researchers also explored the graph representation for the program. Ling et al. [12] proposed an end-to-end code search model named *DGMS*, that utilized graph neural network to represent program graphs generated from AST. Liu et al. [20] also constructed a code search model based on the graph representation generated from AST, that utilized a multi-head attention module to obtain local structure and global dependency information. Consequently, there is a dearth of research on code search for program graphs, such as control flow graph (CFG) and program dependency graph (PDG).

To improve the performance of code search models, researchers have begun experimenting with combining multiple representations. Gu et al. [9] invented a *CODE-NN* to embed code fragments, learning features from the method's name, API invocation sequence, and code tokens for the Java program language. Their result showed that *CODE-NN* went beyond the baseline, such as CodeHow. Following that, Shuai et al. [13] extended the work of Gu et al. by exploring additional neural networks on the same representations, naming their method as *CARLCS-CNN*. They also employed a co-attention mechanism to merge the information of *tokens, method name, and API sequence*. Du et al. [21] trained three different encoders that concentrate on structure, local variable, and API invocation separately, and then fused three models under the tactic of ensemble learning. Some researchers also have incorporated ASTs into their methods. Meng et al. [14] designed *At-CodeSM* that embeds *code, name, and ASTs*. Their three independent can handle lexical encoder, method name, and structural information. The experiment indicated that their model learned the lexical and syntactic features effectively. Mathew and Stolee [22] explored a model for code search, that can support dynamic and static information including the code tokens, generic ASTs, and IO relationship. Shi et al. [23] deployed *CoCoSoDa*, which consisted of pre-trained encoders (GraphCodeBERT) and momentum encoders to capture the high-quality sequence-level representation on several languages. Their work leveraged contrastive learning and soft data augmentation to promote the performance of code searches.

Although the works mentioned above are compatible with the Java programming language, no program graphs were incorporated to enhance their compatibility. In addition, researchers also tried to apply the program graph representation in their model on C programming languages. Specifically, Wan et al. [15]. proposed a deep model named *MMAN* for semantic code search, that represented source code with code token, AST, and CFG for C programs. However, they did not account for the Java programming language in their work, nor did they examine other graph representations, such as the program dependence graph.

Unlike the approaches stated above, in our work, we first make an empirical study to explore the contribution of program representations (especially program graphs) on code search for Java programming language. In our work, we employ a mixed graph by merging the control flow graph (CFG) and program dependence graph (PDG) as well as the code token and AST to enhance the effectiveness of our models.

Program representation learning

Nowadays, program representation learning has attracted the attention of researchers [11, 12]. Deep learning is being used in a greater variety of methods to extract semantic information for software engineering tasks. According to the program representation, such as program token, abstract syntax tree, and control flow or program dependence graph, these methods are divided into token-based, tree-based, and graph-based approaches.

The *n-gram* language model used in natural language processing can be traced back to the origins of the token-based approaches. Such approaches are referring to the analogy of code token and natural language word, that tokenism of the source code for representation learning using deep learning, and extending it to the software engineering tasks (e.g. code completion, vulnerability detection, etc.). Nguyen et al. [24] developed *SLAMC* on *n-gram* model for code recommendation task, encoding the code token by incorporating semantic information into the traditional encoding. Nguyen et al. developed the *MNIRE* [25] for method name generation by extracting features from code content, interface, and class name that contains. Hu et al. [26] used the LSTM to learn language models for making predictions on code element completion. Kang et al. [27] assessed the token embeddings on three downstream tasks including code comment generation, code authorship identification, and code clones detection.

For these tree-based approaches, researchers parse the program into tree-based representations, such as abstract syntax trees (AST). Mou et al. [28] propose the novel tree-based convolutional neural network (*TBCNN*) to model programming languages for clone detection. A neural network model (*ASTNN*) was proposed by Zhang et al. [29], which divided the AST into a series of small corresponding statement trees, that can capture the lexical and syntactical knowledge of statements for source code classification and code clone detection. They discovered that their models were more excellent than state-of-the-art methods for the tasks of code classification and clone detection. Jayasundara et al. [30] employed a capsule network to learn the syntactic structure and semantic dependencies from AST, and vectorized the AST node with types (instead of specific tokens) as lexical words. Chakraborty[31] et al. propose a new tree-based neural network named *CODIT* to model source code changes, and their model was successful in suggesting program modifications.

Researchers also constructed different graph forms of source code to refine the effectiveness of representation learning. Wang et al. [32] employed heterogeneous graphs on learning from source code, which had a better ability than *ASTNN*. Xu et al. [33] proposed a graph-based approach to compute the similarity between two binary files based on the CFG, and implemented a prototype called *Gemini*. Nair et al. [34] proposed *funcGNN* on labeled CFGs, which also aimed to the similarity, and achieve the best effectiveness. Chen et al. [35] proposes a novel API recommendation method called *APIRecCST* by combining API usage with textual information in source code based on API context graph networks. Gao et al. [36] introduced *VulSeeker* for vulnerability detection, which first combined several graph representations of the program. David et al. [37] proposed a novel method for predicting the procedure name in stripped executables, which used static analysis to obtain augmented representations of call sites, encoded them with a control-flow graph (CFG), and then generated a target name. In this paper, we investigate the use of program graph representation to improve code search, as graph-based learning methods are becoming increasingly popular in program representation learning.

Methodology

In this section, we provide the architecture of our approach that employs deep learning technology for code search and also provide the details of building our models

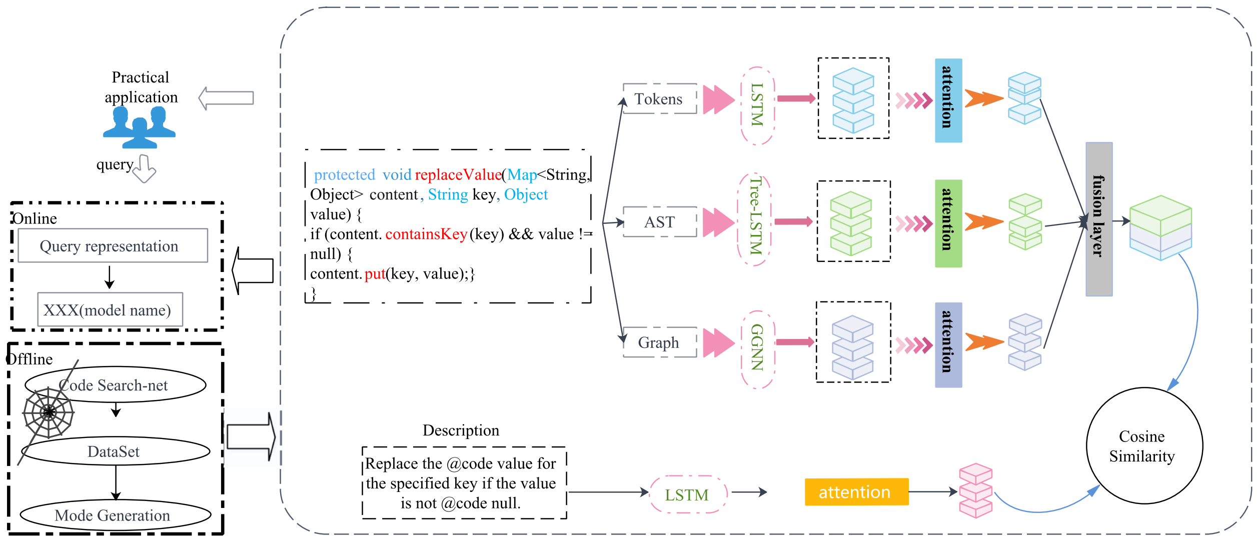

Practical application
query
Online
Query representation
XXX(model name)
Offline
Code Search-net
DataSet
Mode Generation
protected void replaceValue(Map<String, Object> content, String key, Object value) {
if (content. containsKey(key) && value != null) {
content. put(key, value);}
}
Tokens
AST
Graph
LSTM
Tree-LSTM
GGNN
attention
attention
attention
fusion layer
Description
Replace the @code value for the specified key if the value is not @code null.
LSTM
attention
Cosine Similarity

Fig. 2 The architecture on code search with multiple representations of the program

with multiple representations of the program. More especially, we provide three research questions as follows:

- Whether graph representation of java programs can have a positive effect on code search?
- Whether each of these representations of the program has a positive contribution, and which ones provide a more significant effect in code search?
- In practice, how do developers determine their representations, and also how to train their models to achieve optimal performance?

The architecture

Figure 2 represents the architecture of our approach based on deep learning with multiple representations of the program. It consists of two stages: the offline training stage and the online retrieving stage. In the offline training stage, we build and train our model that extracts and fuses the features from three representations of the program, including token, AST, and mixed graph. In the online retrieving stage, the developers provide a query with natural language, then our well-trained model will recommend the most relevant code snippets for developers.

More specifically, the offline training stage can be divided into three sub-parts, including program parser, representation learning, and fusion with attention. In the program parser, we employ the program analysis technology to parse the source code into different representations, including token, AST, and mixed graph. The mixed graph consists of two distinct program graphs: the control flow graph (CFG) and the program dependence graph (PDG). These three employed representations possess different levels of features, that play an appropriate role in code search. To extract these features, we employ deep learning to characterize the program representations automatically, and we call this modality representation learning. Here, we employ different neural networks as well as the attention mechanism to transfer program representations into vectors. In our model, we utilize (token, AST, and mixed graphs) to build our model. So, in the fusion with attention part, we fuse these three representations of the program, and we also embed the queries (description of the code snippet). Finally, we build and train our model with a large-scale dataset that generated ourselves with (*code*, *description*) pairs.

Program parser part

Different representations of the program have different characteristics, such as syntax, semantics, program behaviors and etc, that imply different features of the program. We regard these representations of a program as multiple modalities, and they are semantically consistent to some extent (from one program). Researchers have employed lots of representations, such as token [9, 15], AST [11, 15, 38], graphs [12, 39, 40], method name [9], and API [9, 16, 41]. In our experiment, we investigate the corresponding significance of each program representation in code search. The results indicate that neither the method name nor the API accurately reflects the positive contribution. Consequently, our model retains the token, AST, and graphs. The following are all these program representations:

- Token: the sequence of the original code tokens that compose the method itself.
- AST: abstract syntax tree of the method, that contains rich and well-defined structured information.
- Graphs: the graph representations of the program, including the control flow graph (CFG) and program dependence graph (PDG). Such graphs contain the control and dependency relationships.
- Method name: the name of the given method. Developers consciously christen the method closer to its functionality.
- API: the sequence of API calls in the method. Developers call API to implement a function, and the sequence of the API calls contains some released information.

These representations can facilitate the neural networks to extract the syntax and semantic features. We take corresponding measures to parse the source code into different representations under the guidance of program analysis technology. To get the *token*, we split the source code into a sequence of tokens via $\{(space), \backslash n\}$. We employ *ANTLR4*, a cross-language parser, to generate the AST that obtains richer information. Most of the code from the data is piratical program, that are not compiled by any tool when generating the program graphs. Therefore, we employ *Jcoffee* to complete the partial program for compilation and apply *soot* to generate the needed graphs. In our work, we generate two kinds of program graphs, including CFG and PDG, and merge these two graphs to the mixed graphs. The details can be found in Data collection section. For the method name and API, we apply *JAVALANG*, a pure Python library, to obtain the related properties that can be transformed into a sequence of tokens.

Representation learning

In this section, we focus on feature extraction with deep learning technology from the program representation (Program parser part section). We regard each representation as *the program modality*, which reveals some special characteristics of the program. We first employ a single neural network unit for each modality and then fuse more than one modality to enhance the ability of our model. To be specific, we apply LSTM with attention mechanism to characterizing token, tree-based LSTM with attention mechanism for AST, graph neural network (GGNN) with attention mechanism for the mixed graphs, and RNN for the method name and API.

Given a code snippet is denoted as c. The code snippet is parsed into $[tok; ast; g]$, which represent token, AST, and mixed graphs respectively.

Token representation learning

We employ LSTM as our neural network unit for extracting features from code tokens, as follows:

$$\mathbf{h}_i^{tok} = LSTM\left(\mathbf{h}_{i-1}^{tok}, w(tok_i)\right)$$

where $i \in [1, n]$, and n is the number of the tokens in this code snippet c. The w is the word embedding layer, and the last state $\mathbf{h}_n^{tok}$ is the final token. We also employ an attention mechanism to identify the significant information from all tokens:

$$\alpha_i^{tok} = \frac{\exp(\mathrm{g}^{\mathrm{tok}}(\mathrm{f}^{\mathrm{tok}}(\mathbf{h}_\mathrm{i}^{\mathrm{tok}}), \mathbf{u}^{\mathrm{tok}}))}{\sum_j \exp(\mathrm{g}^{\mathrm{tok}}(\mathrm{f}^{\mathrm{tok}}(\mathbf{h}_\mathrm{j}^{\mathrm{tok}}), \mathbf{u}^{\mathrm{tok}}))}$$

where $\mathbf{h}_i^{tok}$ represent the i^{th} hidden state in tokens, f^{tok} denote a linear layer and g^{tok} is the inner project. The $\mathbf{u}^{tok}$ denotes the context vector of token modality, which is a high-level representation. The word context vector $\mathbf{u}^{tok}$ is randomly initialized and jointly learned during training. Then, the final token representation of this code snippet c can be present as a vector as:

$$\mathbf{Tok} = w\left[\sum_i a_i^{tok}\mathbf{h}_i^{tok}\right]$$

where w is the attention weight.

AST representation learning

The AST representation of the code snippet is a binary tree, and its left children denote as $(\mathbf{h}_L, \mathbf{c}_L)$ and right as $(\mathbf{h}_R, \mathbf{c}_R)$. We employ the tree-based LSTM to transform AST into vectors. As follows,

$$(\mathbf{h}_i^{ast}, \mathbf{c}_i^{ast}) = LSTM\left(([\mathbf{h}_{iL}^{ast}; \mathbf{h}_{iR}^{ast}], [\mathbf{c}_{iL}^{ast}; \mathbf{c}_{iR}^{ast}]), w(ast_i)\right)$$

where $i \in [1, n]$, and operation $[;]$ denotes the concatenation operation of two vectors. Analogously, we also use the attention mechanism to score the nodes of the AST, and get the final vector of AST representation:

$$\mathbf{AST} = w\left[\sum_i a_i^{ast}\mathbf{h}_i^{ast}\right].$$

Graph representation learning

According to Program parser part section, we possess two kinds of graph representations in our work, which are the control flow graph and the program dependence graph. Taking these two graphs, we also obtain a third mixed graph that merges CFG and PDG into a hybrid graph, denoted as *Mix graph*. For each graph, we employ the graph neural network (GGNN) with self-node attention to embed it into a vector.

Given a directed graph, we use the GGNN to learn features from the graph. We first get a graph with $\{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V}$ is a set of vertices (v, l_v), on behalf of all nodes of the graph. $\mathcal{E}$ is a set of edges (v_i, v_j, l_e), on behalf of each relationship of code. l_v and l_e are labels of vertex and edge.

We use GGNN to learn the vector. First, we initialize the hidden state for each vertex $v \in \mathcal{V}$ as $\mathbf{h}_{v,0}^{graph} = w(l_v)$, w denotes the one-hot embedding. For every round t, each vertex $v \in \mathcal{V}$ gets the vector $\mathbf{m}_{v,t+1}$, which on behalf of the message, converged from its neighbors. The vector $\mathbf{m}_{v,t+1}$ is aggregated as following:

$$\mathbf{m}_{v,t+1} = \sum_{v' \in \mathcal{N}(v)} \mathbf{W}_{l_e} \mathbf{h}_{v',t}$$

where $\mathcal{N}(v)$ denotes the neighbours of vertex v. For round t, $\mathbf{W}_{l_e}$ is the weight matrix to map messages from each neighbor into a shared space. Then, GGNN uses GRU(Gated Recurrent Unit) to update the hidden state of each vertex, as follows:

$$\mathbf{h}_{v,t+1}^{graph} = GRU\left(\mathbf{h}_{v,t}^{graph}, \mathbf{m}_{v,t+1}\right).$$

In the end, after this t round of iterations, we gather all hidden states of vertices to obtain the embedding. Therefore, we adopt each graph node with the weight α^{graph} as:

$$\alpha_i^{graph} = \mathrm{sigmoid}\left(\mathrm{g}^{\mathrm{graph}}(\mathrm{f}^{\mathrm{graph}}(\mathbf{h}_\mathrm{i}^{\mathrm{graph}}), \mathbf{u}^{\mathrm{graph}})\right)$$

where $\mathbf{h}_i^{graph}$ represent the i^{th} hidden state in Graph nodes. f^{graph} denote a linear layer and g^{graph} is the inner project. $\mathbf{u}^{graph}$ denotes the context vector of Graph modality, a high-level representation of the whole Graph nodes. Finally, we get the final representation of the graph:

$$\mathbf{G} = w\left[\sum_i a_i^{graph} \mathbf{h}_i^{graph}\right]$$

where $\mathbf{G}$ is the final semantic representation of *graph* and w is the attention weight.

To enhance the effectiveness of our approach, we naturally can select more than one representations to build our model. We believe that such multiple modalities play a positive contribution to the code search. In this work, we finally select *token, AST, and graphs* these three modalities. To do that, we fuse these modalities of the program representations with concatenation along every specific dimension. As follows,

$$\mathbf{C} = tanh([Tok; AST; G])$$

where $\mathbf{C}$ is the final representation of code snippet, the [;] is the concatenation.

Besides, given a description d for a code snippet, that corresponds to a code snippet c. We also employ the *LSTM* with attention mechanism to represent natural language description **Des**.

$$\mathbf{h}_i^{des} = LSTM(\mathbf{h}_{i-1}^{des}, w(d_i))$$

where $i = 1,...,n$, w is word embedding layer. And the last state $\mathbf{h}_n^{tok}$ is the final sample d in whole dataset. We apply description attention layer to calculate the attention score $\alpha^{des}(i)$:

$$\alpha_i^{des} = \frac{\exp(\mathrm{g}^{\mathrm{des}}(\mathrm{f}^{\mathrm{des}}(\mathbf{h}_\mathrm{i}^{\mathrm{des}}), \mathbf{u}^{\mathrm{des}}))}{\sum_j \exp(g^{des}(f^{des}(\mathbf{h}_j^{des}), \mathbf{u}^{des}))}$$

where $\mathbf{h}_i^{des}$ represent the i-th hidden state in description. f^{des} denote a linear layer and g^{des} is the inner project. $\mathbf{u}^{des}$ denotes the context vector of description modality, which is a high level representation of the whole descriptions. The word context vector $\mathbf{u}^{des}$ is randomly initialized and jointly learned during training. Finally, we get the final representation of description **Des**:

$$\mathbf{Des} = w\left[\sum_i a_i^{des} \mathbf{h}_i^{des}\right]$$

where **Des** is the final semantic representation of *Des* and w is the attention weight.

Model training

In our approach, we build and train our model that embeds code and description into a unified vector space. The goal of our model is that if a code snippet c and a description d represent consistent semantics, then their embedded vectors should be similar to each other.

When training our model, we construct each training instance as a triple $\langle c, d^+, d^- \rangle$. For each code snippet c, there is a positive description d^+ (correct description) as well as a negative description d^- (incorrect description), that is randomly chosen from our dataset D^*. Then, we employ the two couples $\langle c, d^+ \rangle$ and $\langle c, d^- \rangle$ from $\langle c, d^+, d^- \rangle$ to train our model with the following loss function [9][15]:

$$\mathcal{L} = \sum_{\langle c, d^+, d^- \rangle} max(0, \epsilon - cos(\mathbf{C}, \mathbf{D}^+) + cos(\mathbf{C}, \mathbf{D}^-))$$

where ϵ denotes the constant margin. $(\mathbf{C}, \mathbf{D}^+, \mathbf{D}^-)$ are the embedded vectors for c, d^+, d^-. In our experiment, we set the ϵ to 0.6.

Experimental methodology

In this section, we provide the details of the dataset for building our model and describe our experimental methodology, including experiment steps and evaluation metrics.

Data collection

A large-scale data corpus is essential for training our models, which contain code fragments and the corresponding descriptions. We choose the dataset from *codeSearchNet*[1] as our original dataset, that it has been widely adopted by researchers in this field [5, 13, 23, 42–44]. This dataset covers the code snippets in python, JavaScript, Ruby, go, java, and PHP programming languages. In our work, we focus on the Java programming language. The original dataset contains numerous website-collected partial programs, but there are no appropriate representations of the program that can be used in our work. We, therefore, parse these code snippets according to Program parser part section to generate our dataset. Our model employs multiple representations of the code fragment, including token, AST, and mixed graphs (CFG and PDG). The generation details are as follows:

- Code token: we spilled the source code with $\{(space), ; , . \backslash\}$.
- AST: we employ *ANTLR4*[2] to build the AST of the code, and transform it to binary trees following the *leftmost-child-next-right-sibling* rule. Specifically, a) the root node of tree is directly used as the root node of the new binary tree; b) take the first child node of the root node of the tree as the left son of the root node, and if the child node has a sibling node, the first sibling node (direction from left to right) of the child node is the right son of the child node; c) add the remaining nodes in the tree to the binary tree in order as in the previous step, until all the nodes in the tree are in the binary tree.
- Graph representation: we employ *Soot*[3] to generate the graph representations. Before the soot works, we use *Jcoffee*[4] to complete the partial code snippet. Although this tool only can help us handle 26% of them, it is enough for our model build (71865). In our work, we explore three kinds of graphs, including CFG, PDG, and mix graph (CFG and PDG).
- Method name and API: we employ *JAVALANG*[5] to parser the method name and MethodInvocation of the code snippet, and split them into token sequence by the camel case. For example, the method name *clearCache* can be split into tokens *clear* and *Cache*. These two representations are used in our baselines.

[1] https://github.com/github/CodeSearchNet/data-details
[2] https://github.com/antlr/grammars-v4
[3] https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot
[4] https://github.com/piyush69/JCoffee
[5] https://github.com/c2nes/javalang

Table 1 The information of our experimental data set

Dataset	Total	Train	Test	Valid
CodeSearchNet	496,688	454,451	26,909	15,328
Our Dataset	71,865	67,865	2,000	2,000

For each code snippet in our dataset, we also extract the description in natural language, that can be obtained directly from the original dataset. Table 1 is the information of the original dataset and our own dataset. From this table, we can see that there are still a large number of data from the original dataset remaining in our dataset (71, 865). We divide our dataset into three parts, including the training data (67, 865), the validation data (2, 000), and the test data (2, 000).

Experimental steps

To assess the effectiveness of our approach, we conduct our experiments from three perspectives, including the effectiveness experiment, the modal experiment, and the performance experiment.

- The effectiveness experiment: It assesses the best ability of our model in searching the intent code from our corpus, that building and training with three modalities including token, AST, and the mixed graphs.
- The modality experiment: It assesses the ability of each modality in code search, especially the program graph representation. It will help developers make their decision on how to choose these representations in practice.
- The performance experiment: It assesses the effect of the parameter in our model training process, and also assesses the scope of the performance with the training epochs.

Metric

We employ two kinds of metrics for evaluating our well-trained model, including *R@{k}* that recalls at top *k* successful recommendations and MRR that the mean reciprocal ranking. These two metrics are as follows,

- **R@{k}** is the percentage of the correct result in the top *k* results in a set of queries. It is calculated by

Table 2 Results for the effectiveness experiment

Method	R@1	R@5	R@10	MRR
DeepCS	0.294	0.495	0.589	0.393
MMAN	0.319	0.532	0.622	0.422
Token+AST+CFG	0.369	0.583	0.657	0.469
Token+AST+PDG	0.351	0.569	0.651	0.456
Token+AST+Mix	0.370	0.594	0.671	0.474

$$R@\{k\} = \frac{1}{|Q|}\sum_{q=1}^{|Q|} f(q,k)$$

where Q is the set of queries, $f(q, k)$ returns 1 if the correct result exists in the top k result or returns 0 otherwise. So, a higher metric value represents the better performance of our approach. In this work, we select the value of k in (1, 5, 10).

- **MRR**(Mean Reciprocal Rank) is the average of reciprocal of ranks of queries, which is calculated by

$$MRR = \left(\frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{rank_i}\right).$$

We also choose two state-of-the-art works as our baselines, and that are *DeepCS* [9] *and MMAN* [15]. We implement these baselines to adapt our dataset, and the parameters for these approaches in our experiment are the same, including word embedding size with 300, the hidden size of LSTM and tree-based LSTM with 512, the and 5 rounds of iteration in GGNN. We set the margin ϵ to 0.6, and the learning rate to 0.001, dropout with 0.1. The models in this work if not specified are trained with 300 epochs and 256 batch size. We implement these models using PyTorch 1.8.0 with Python 3.8.3 on a machine with a NVIDIA RTX 3090 graphics card as well as 24 GB memory. Our implementation can be found on the GitHub[6], and the dataset is available in the Google Drive[7].

Results and discussion

In this section, we provide the results of our experiment and the discussions for our approach.

[6] https://github.com/metacodeteam/codeSearch

[7] https://drive.google.com/drive/folders/1BTdsQNMwXMFEZ4bUHFVywnmhPTEjZ7-i?usp=sharing

The effectiveness experiment

RQ1 *Whether graph representation of Java programs can have a positive effect on code search?*

To evaluate the effectiveness of our approach, we conduct this *effectiveness experiment*. The results are shown in Table 2. Column 1 depicts all the methods, and columns 2-4 list the metrics employed in this experiment. Rows 2 and 3 are the results of baselines (*DeepCS and MMAN*), and the other Rows depict the effectiveness of our improvement with the graph representations. *MMAN* uses tokens, AST, and CFG to build the model, while *DeepCS* uses tokens, API, and method name. For our models, we employ token, AST, and the mixed graphs, that abbreviated as "Mix".

Compared with the baselines, the effectiveness of our models is *more effective*. Particularly, every single metric is higher than the other two baselines. From the perspectives of graph representation, we can see that those models that employ different graph have effective effectiveness. In particular, the metrics of these models are ranging from 35.1 to 37.0% for R@1, 58.3% to 59.4% for R@5, 65.1% to 67.1% for R@10, and 45.6% to 47.4% for MRR. Meanwhile, the model built with the mixed graphs possesses the best effectiveness. Taking R@1 and R@5 as examples, the R@1 reaches up to 37% and R@10 to 59.4%. Furthermore, there are no notable differences among these models with various graphs.

Therefore, we can answer our research question 1, the graph representation plays the positive effect in code search, and we recommend that developers prefer mixed graph representation to achieve the best ability.

The modality experiment

RQ2 *Whether each of these representations of the program has a positive contribution, and which ones provide a more significant effect in code search?*

In our work, we parse the program with several representations, and we call each representation as program modality. To assess the significance of each modality, we conduct this modal experiment. Table 3 displays the effectiveness of our modal experiment. We explore the single and double modality to find out their contribution. For the single modality (Rows 2-9), we investigate the models that only employ method name (abbreviated "MN"), API, token, and graph representation (CFG, PDG, and the Mix respectively. For the double modality (Rows 10-16), we investigate the models with three kinds

Table 3 Results for the modal experiment

Method	R@1	R@5	R@10	MRR
MN	0.001	0.002	0.004	0.004
API	0.003	0.004	0.007	0.006
CFG	0.052	0.118	0.177	0.095
PDG	0.061	0.154	0.201	0.112
Mix	0.106	0.241	0.310	0.177
AST	0.232	0.403	0.481	0.317
Token	0.235	0.433	0.522	0.333
AST+CFG	0.202	0.398	0.487	0.296
AST+PDG	0.216	0.388	0.477	0.301
AST+Mix	0.200	0.385	0.476	0.291
Token+AST	0.326	0.565	0.657	0.435
Token+CFG	0.326	0.565	0.656	0.435
Token+PDG	0.332	0.538	0.639	0.443
Token+Mix	0.321	0.549	0.639	0.427
Token+AST+Mix	0.370	0.594	0.671	0.474

of combination, including AST and graphs, token and AST, token and graphs.

For the single modality, models are built with only one modality, and the results are shown in *Rows 2-9*. For the method name and API, CFG, PDG, and mixed graph, the effectiveness of them is incredibly unacceptable, and they have no ability in finding target source code for the developers. Specifically, the numbers of the R@1 metric are all below 11.0%, and even R@10 below 31%. For the modality of the token, we built two models that applied different neural networks with LSTM and MLP. For AST and Token (MLP), the effectiveness of these models is acceptable, and have the general ability on code search. In particular, the metric of R@1 is around 23.0%, R@5 is around 42%, R@10 is around 50%, and MRR is around 32%. Nonetheless, the model that builds with token (LSTM) possesses the positive effectiveness, that has the stronger ability on code search. Specifically, the metric of R@1 reaches 32.8%, R@5 to 55.0%, R@10 to 65.7%, and MRR to 43.4%. Therefore, we can conclude that the models built with single modality have limited ability, we do not advise developers to select this; and if that's the only option, we prefer to build the model with AST or token, especially token (LSTM).

For the double modality, the models are built with two modalities, and the results are shown in Rows 10-16. According to that whether employing token modality or not, these models are kindly divided into two categories, the models without token (Rows 10-12) and the models with token (Rows 12-16). For these models without token, the effectiveness of these models is acceptable, that have the general ability on code search. Specifically, the metric of R@1 is around 23.0%, R@5 is around 42%, R@10 is around 50%, and MRR is around 32%. For these models with token, they possess the positive effectiveness, that have the stronger ability on code search. Specifically, the metric of R@1 reaches 32.8%, R@5 with 55.0%, R@10 with 65.7%, and MRR with 43.4%. Therefore, we consulate that these models that are built with dual modalities have positive effects, and the token plays a significant role in code search.

In summary, we can answer our research question 2, each representation plays a positive effect in code search. When building the models, we strongly recommend developers choose the appropriate double modality while retaining the token in the model to achieve better ability of effectiveness.

The performance experiment

To figure out the contribution of the parameter in our model training, we conduct the performance experiment. We build the models with the modalities of token, AST, and mixed graph, and adjust three parameters, including dropout, batch size, and learning rate. The performance of our models is shown in Fig. 3. We set a default value for each parameter (dropout=0.1, learning rate=0.001, and batch size=256). For each model in the figures, the x-axis and the y-axis is the number of epochs and the score of our metrics respectively. These figures illustrate the results of the experiment evaluated by different metrics, including R@1, R@5, R@10, and MRR. We display our models for each parameter with colored curves as the number of epochs increases (from 1 to 300).

We can observe that the effectiveness of our models achieves its maximum results around 200 epochs. In the early stage of training (from 1 to 100), the performance of our models improves rapidly. From 100 to 200 epochs, although the improvement of the models slows down, it is still improving. When training more than 200 epochs, the performance reaches stability. So, we recommend developers train the models with 200 epochs to reach the best ability.

Now, we set different values for other parameters.

First, we change dropout to 0.5. It can be concluded that the dropout parameter only plays a positive effect on the effectiveness of our models. As the dropout increases to 0.5, the performance improvement is quite slight on the effectiveness of our models. Specifically, the value of the R@1 metric reaches up to the best, which is equal to 37.15%. Nonetheless, setting the value of dropout to 0.1 is still a pretty option.

After that, we adjust batch size to 128. It can be derived that it does not have a significant effect on the effectiveness of our models. When reducing the value from 256 to 128, there are no significant differences between the two

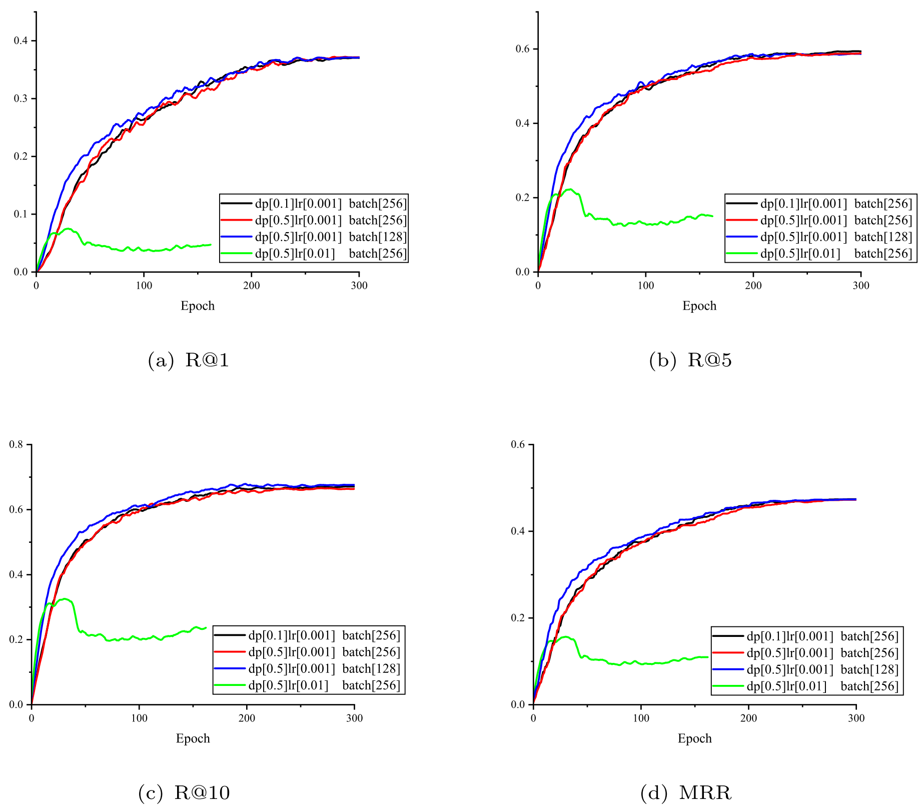


Fig. 3 The effectiveness of the performance experiment

models. Nevertheless, this parameter still serves different effects on metrics, which give slight improvement for a certain metric, such as R@1 can be more accurate. So, developers can make their own decisions for their special preferences.

For the learning rate, it can be observed that our model that has a learning rate of 0.001 possesses extremely negative effectiveness. Particularly, the metrics of the model are below 7.35% for R@1, 22.65% for R@5, 32.8% for R@10, and 15.68% for MRR, which is much worse than the previous results. Such models cannot recommend source code to developers. Therefore, we select 0.01 in our experiment.

In summary, when building and training our models, the parameters play a decisive role in the performance. We provide our suggestion on the parameter values with our default setting, which can acquire a good performance.

Discussion

RQ3 *In practice, how do developers determine their representations, and also how to train their models to achieve optimal performance?*

According to the results, we give our recommendation for developers to build their models, and answer our last research problem.

Our models built with three modalities (*token, AST, and mixed graph*) possess the *best ability* on code search according to the effectiveness experiment (The effectiveness experiment section). We strongly suggest developers employ such three modalities when performing a search. Furthermore, the *graph representation has a positive effect.* The developers should obtain the graph representation as much as possible to enhance the ability of the model.

From the perspective of *the modality selection*, the modality of *token, AST, and mixed graphs* play a more *significant impact* than the others, especially the token. In a situation where these three modalities are not available at the same time, we strongly recommend that the developers *employ the modality of AST or graph while retaining at least the modality of token.* The models trained by this combination still achieve an *effective ability.*

Considering the parameters, developers should be more careful in choosing the values. According to our results of the performance experiment (The performance experiment section), we recommend that developers follow our *default value.* In the case that developers have further special requirements, they may adjust parameters, while should ensure that the model's capabilities are not compromised. When training the models, it is a good option that *sets the number of epochs to 200 epochs* to reach the best performance.

Threats to validity

There are two threats to the validity of our work, including internal threat and construct threat. The internal threat to the validity is the construction of our used dataset. To generate our dataset, we employ Jcoffee and Soot to parse the program for graph representation. Some semantic information from the program may have been discarded. When we employ *Jcoffee* to complete the partial code, the complemented part may not possess valid semantics. Our defense is that the effect of the invalid semantics is quite small. Our model still has strong capabilities. The second threat to the validity is the construction of our models. We employ several neural networks to characterize the program representations. Therefore, there is a potential improvement in the selection of neural networks for each representation of the program. For instance, we can employ the MLP and LSTM to encode the token of the program, however, the effectiveness of these two networks has significant differences. We recommend the LSTM to the developers because of the better effectiveness. Analogous matters can also occur for the other representations of the program. Meanwhile, for the specific neural network, the effectiveness of the models is not necessarily optimal, and it is possible to enhance the predictive ability by adjusting specific parameters. Nonetheless, we conclude that our models with the current configuration already have a reasonable capacity from our experiment. We make the selection of the neural networks as well as the parameter adjusting as our future work.

Conclusion

In response to the increasing expansion of multimedia applications, we provide an approach for code search to enhance productivity in the development of software infrastructure in edge computing. This study conducted an empirical investigation to assess the efficacy of representations of the program on code search, including token, AST, graph, method name, and API. Our model that consists token, AST, and mixed graphs possessed the best capability. Specifically, we construed a mixed program graph for java language containing the control flow and the program dependence graph and employed different neural networks to characterize this mixed graph as well as the other representations. We constructed an experiment to answer three research problems. The result showed that our mixed graph has a positive effect, and the token and AST have significant contributions to our models. When building and training models, we strongly recommend developers retain these three representations, or keep AST or graphs while retaining at least one token. Such selection helps developers to obtain an effective ability model. The selection for parameters is also provided to the developer who can optimize the performance to the best.

In the future, we plan to further enhance the capability of our model for multimedia application development. We only handle the Java programming language in this work, so intend to explore more programming languages, such as Python, C, JavaScript, etc. In this work, the features captured from the program are simply fused by concatenating together. We will also consider taking a new fusion strategy on code search, that will be more productive in the future.

Acknowledgements
We would like to express our sincere thanks to the editors and reviewers for their insightful comments and suggestions on improving this paper.

Authors' contributions
F.Z and M.L wrote the main manuscript text and conducted the experiment. All authors reviewed the manuscript.

Funding
This work was partially supported by the National Natural Science Foundation of China under Grant No. U20A6003, National Natural Science Foundation of China for the Key Program under Grant No. 62237001, the Guangdong Natural Science Fund Project under Grant No.2024A1515011502, the Key Area Research and Development Program of Guangdong Province under Grant No. 2018B010109007, and Guangdong Joint Fund of the National Natural Science Foundation of China under Grant Nos. U1801263 and U1701262, the Self-selected research projects in the field of national defense science and technology of the Chinese People's Armed Police Force under Grant No. ZZKY20233115.

Availability of data and materials
No datasets were generated or analysed during the current study.

Declarations

Ethics approval and consent to participate
This article does not contain any studies with human participants or animals performed by any of the authors.

Competing interests
The authors declare no competing interests.

Received: 14 December 2023 Accepted: 9 March 2024
Published online: 01 April 2024

References
1. Bilal K, Erbad A (2017) Edge computing for interactive media and video streaming. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC). IEEE p 68–73
2. Long C, Cao Y, Jiang T, Zhang Q (2017) Edge computing framework for cooperative video processing in multimedia iot systems. IEEE Trans Multimedia 20(5):1126–1139

3. Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv (CSUR) 51(4):1–37
4. Perkusich M, Silva LC, Costa A, Ramos F, Saraiva R, Freire A, Dilorenzo E, Dantas E, Santos D, Gorgônio K et al (2020) Intelligent software engineering in the context of agile software development: A systematic literature review. Inf Softw Technol 119:106241
5. Di Grazia L, Pradel M (2023) Code search: A survey of techniques for finding code. ACM Comput Surv 55(11):1–31
6. Liu C, Xia X, Lo D, Gao C, Yang X, Grundy J (2021) Opportunities and challenges in code search tools. ACM Comput Surv (CSUR) 54(9):1–40
7. Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories. Data Min Knowl Discov 18(2):300–336
8. Mishne A, Shoham S, Yahav E (2012) Typestate-based semantic code search over partial programs. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. p 997–1016
9. Gu X, Zhang H, Kim S (2018) Deep code search. In: Proceedings of the 40th International Conference on Software Engineering. p 933–944
10. Sachdev S, Li H, Luan S, Kim S, Sen K, Chandra S (2018) Retrieval on source code: A neural code search. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. p 31–41
11. Sun Z, Liu Y, Yang C, Qian Y (2020) PSCS: A path-based neural model for semantic code search. arXiv preprint arXiv:2008.03042.
12. Ling X, Wu L, Wang S, Pan G, Ma T, Xu F, Liu AX, Wu C, Ji S (2021) Deep graph matching and searching for semantic code retrieval. ACM Trans Knowl Discov Data (TKDD) 15(5):1–21
13. Shuai J, Xu L, Liu C, Yan M, Xia X, Lei Y (2020) Improving code search with co-attentive representation learning. In: Proceedings of the 28th International Conference on Program Comprehension. p 196–207
14. Meng Y (2021) An intelligent code search approach using hybrid encoders. Wirel Commun Mob Com 2021:1–6
15. Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu P (2019) Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, p 13–25
16. Lv F, Zhang H, Lou JG, Wang S, Zhang D, Zhao J (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, p 260–270
17. Ding SH, Fung BC, Charland P (2016) Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. p 461-470
18. Wang H, Zhang J, Xia Y, Bian J, Zhang C, Liu TY (2020) Cosea: Convolutional code search with layer-wise attention. arXiv preprint arXiv:2010.09520.
19. Cheng Y, Kuang L (2022) CSRS: code search with relevance matching and semantic matching. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. p 533–542
20. Liu S, Xie X, Siow J, Ma L, Meng G, Liu (2023) Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. IEEE Transactions on Software Engineering. p 1–6
21. Du L, Shi X, Wang Y, Shi E, Han S, Zhang D (2021) Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management. p 2994–98
22. Mathew G, Stolee KT (2021) Cross-language code search using static and dynamic analyses. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p 205–217
23. Shi E, Gub W, Wang Y, Du L, Zhang H, Han S, Zhang D, Sun H (2022) Enhancing semantic code search with multimodal contrastive learning and soft data augmentation. arXiv preprint arXiv:2204.03293
24. Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN (2023) A statistical semantic language model for source code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. p 532–542
25. Nguyen S, Phan H, Le T, Nguyen TN (2020) Suggesting natural method names to check name consistencies. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering. p 1372–1384
26. Hu X, Men R, Li G, Jin Z (2019) Deep-autocoder: Learning to complete code precisely with induced code tokens. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). IEEE, 1:159–168
27. Kang HJ, Bissyandé TF, Lo D (2019) Assessing the generalizability of code2vec token embeddings. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, p 1–12
28. Mou L, Li G, Jin Z, Zhang L, Wang T (2014) TBCNN: A tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718
29. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, p 783–794
30. Jayasundara V, Bui ND, Jiang L, Lo D (2019) TreeCaps: Tree-structured capsule networks for program source code processing. arXiv preprint arXiv:1910.12306
31. Chakraborty S, Ding Y, Allamanis M, Ray B (2020) Codit: Code editing with tree-based neural models. IEEE Trans Softw Eng. 31;48(4):1385–99
32. Wang W, Li G, Ma B, Xia X, Jin Z (2020) Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. arXiv preprint arXiv:2002.08653
33. Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security p 363–376
34. Nair A, Roy A, Meinke K (2020) funcgnn: A graph neural network approach to program similarity. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). p 1–11
35. Chen C, Peng X, Xing Z, Sun J, Wang X, Zhao Y, Zhao (2021) Holistic combination of structural and textual code information for context based API recommendation. IEEE Trans Softw Eng. 48(8):2987–3009
36. Gao J, Yang X, Fu Y, Jiang Y, Sun J (2018) Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p 896–899
37. David Y, Alon U, Yahav E (2020) Neural reverse engineering of stripped binaries using augmented control flow graphs. Proc ACM Program Lang 4(OOPSLA):1–28
38. LeClair A, Haque S, Wu L, McMillan C (2020) Improved code summarization via a graph neural network. In: Proceedings of the 28th international conference on program comprehension. p 184–195
39. Cummins C, Fisches ZV, Ben-Nun T, Hoefler T, Leather H (2020) Programl: Graph-based deep learning for program optimization and analysis. arXiv preprint arXiv:2003.10536
40. Zeng C, Yu Y, Li S, Xia X, Wang Z, Geng M, Bai L, Dong W, Liao X (2023) degraphcs: Embedding variable-based flow graph for neural code search. ACM Transactions on Software Engineering and Methodology. 32(2):1–27
41. Gu X, Zhang H, Zhang D, Kim S (2016) Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. p 631–642
42. Zhu Q, Sun Z, Liang X, Xiong Y, Zhang L (2020) Ocor: An overlapping-aware code retriever. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p 883–894
43. Wang C, Nong Z, Gao C, Li Z, Zeng J, Xing Z, Liu Y (2022) Enriching query semantics for code search with reinforcement learning. Neural Netw 145:22–32
44. Ishtiaq AA, Hasan M, Haque MM, Mehrab KS, Muttaqueen T, Hasan T, Iqbal A, Shahriyar R (2021) Bert2code: Can pretrained language models be leveraged for code search?. arXiv preprint arXiv:2104.08017