How to Better Utilize Code Graphs in Semantic Code Search?

Yucen Shi
shiyucen@stumail.neu.edu.cn
School of Computer Science and Engineering, Northeastern University
Shenyang, China

Ying Yin
yinying@cse.neu.edu.cn
School of Computer Science and Engineering, Northeastern University
Shenyang, China

Zhengkui Wang
zhengkui.wang@singaporetech.edu.sg
InfoComm Technology Cluster, Singapore Institute of Technology
Singapore

David Lo
davidlo@smu.edu.sg
School of Information Systems, Singapore Management University
Singapore

Tao Zhang
tazhang@must.edu.mo
School of Computer Science and Engineering, Macau University of Science and Technology
Macau, China

Xin Xia
xin.xia@acm.org
Software Engineering Application Technology Lab, Huawei
Hangzhou, China

Yuhai Zhao*
zhaoyuhai@mail.neu.edu.cn
School of Computer Science and Engineering, Northeastern University
Shenyang, China

Bowen Xu
bowenxu.2017@phdcs.smu.edu.sg
School of Information Systems, Singapore Management University
Singapore

ABSTRACT

Semantic code search greatly facilitates software reuse, which enables users to find code snippets highly matching user-specified natural language queries. Due to the rich expressive power of code graphs (e.g., control-flow graph and program dependency graph), both of the two mainstream research works (i.e., multi-modal models and pre-trained models) have attempted to incorporate code graphs for code modelling. However, they still have some limitations: First, there is still much room for improvement in terms of search effectiveness. Second, they have not fully considered the unique features of code graphs.

In this paper, we propose a Graph-to-Sequence Converter, namely *G2SC*. Through converting the code graphs into lossless sequences, *G2SC* enables to address the problem of small graph learning using sequence feature learning and capture both the edges and nodes attribute information of code graphs. Thus, the effectiveness of code search can be greatly improved. In particular, *G2SC* first converts the code graph into a unique corresponding node sequence by a specific graph traversal strategy. Then, it gets a statement sequence by replacing each node with its corresponding statement. A set of carefully designed graph traversal strategies guarantee that the process is one-to-one and reversible. *G2SC* enables capturing rich semantic relationships (i.e., control flow, data flow, node/relationship properties) and provides learning model-friendly data transformation. It can be flexibly integrated with existing models to better utilize the code graphs. As a proof-of-concept application, we present two *G2SC* enabled models: *GSMM* (*G2SC* enabled multi-modal model) and *GSCodeBERT* (*G2SC* enabled *CodeBERT* model). Extensive experiment results on two real large-scale datasets demonstrate that *GSMM* and *GSCodeBERT* can greatly improve the state-of-the-art models *MMAN* and *GraphCodeBERT* by 92% and 22% on R@1, and 63% and 11.5% on MRR, respectively.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**.

KEYWORDS

Semantic Code Search, Neural Networks, Graph Embedding

ACM Reference Format:
Yucen Shi, Ying Yin, Zhengkui Wang, David Lo, Tao Zhang, Xin Xia, Yuhai Zhao, and Bowen Xu. 2022. How to Better Utilize Code Graphs in Semantic Code Search?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549087

*Yuhai Zhao is the corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549087

1 INTRODUCTION

Semantic code search plays a vital role for software developers to serve various purposes, especially code reuse, which enables users to find code snippets highly matching user-specified natural language queries. For example, to the query *"How to read an object from an xml?"*, code search engine returns a code snippet candidate like Figure 1 from a billion-token-scale code base (e.g., Github). The reuse of existing code avoids the necessity for developers to reinvent the wheel and save the time and resource cost during software development.

```
public static < S > S deserialize(Class c, File xml) {
    try {JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

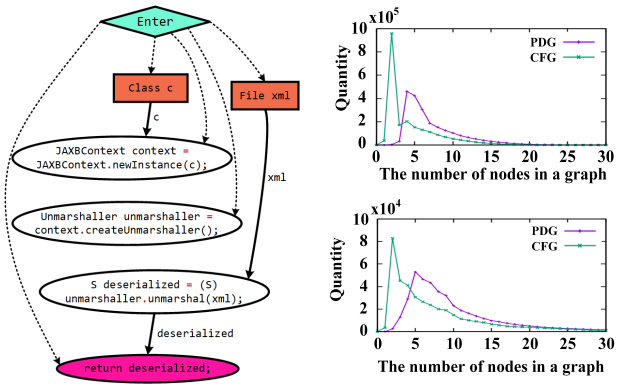
Figure 1: Code snippet for query "*How to read an object from an xml*?"

The task is non-trivial as there is often a semantic gap between the high-level intent in the natural language queries and the low-level implementation details in the source code [13]. For example, the code snippet in Figure 1 contains none of the key words in the query such as *read* and *object* or other similar texts. Thus, those unimodal approaches that are based solely on keyword matching or text similarity, such as information retrieval (IR)-based code search approaches [25, 26, 29, 36, 40], cannot address the challenge well.

Recent research works have proposed various multi-modal models based on deep learning techniques. As the lexical gap between queries and source code is bridged by their semantic representation learning in high-dimensional space, they succeed in improving search effectiveness. These works can be categorized into two main streams: 1.) the multi-modal models with pre-training, which are designed with Transformer-based architecture and guided by several pre-training tasks to learn the generic representation of multi-modal data and then fine-tuned for a set of code-related downstream tasks such as code search, code summarization, etc.; 2.) the multi-modal models without pre-training, which are specifically trained for code search in an end-to-end way. For simplicity, we refer to the models in the above-mentioned first and second main streams of work as pre-trained models and multi-modal models respectively. *CodeBERT* [10] and *DCS* [13] are the examples of pre-trained models and multi-modal models, respectively.

Though these models are more capable to capture more semantics, they still suffer from using simple features of code and ignore the code structural information. Code graphs (e.g., abstract syntax trees (ASTs) or control-flow graphs (CFGs)) have been proven effective and expressive to capture the rich structural semantics of the source code[20, 45]. The most recent research effort is made to leverage the code graphs in both of the two research streams. For example, in the multi-modal model stream, Wan et al. [45] proposed a multi-modal attention network *MMAN* by integrating CFGs in the model and adopted the Graph Neural Networks (*GNNs*) to learn the semantic information hidden in the code graphs followed by an attention layer to integrate the unstructured and structured features. In the pre-trained model stream, *GraphCodeBERT* [16] introduced a graph-guided masked attention function to incorporate the code structure that captures data flow. However, through an in-depth study of the two main streams of work, we have the following two findings that motivate our work in this paper.

1.) *Both of them still have much room for improvement in the search effectiveness.* For example, according to the original results reported in *MMAN* [45] and *GraphCodeBERT* [16], the two models have achieved the state-of-the-art in their respective tasks. However, on a widely used evaluation metric Mean Reciprocal Rank (MRR) with



(a) PDG of code in Figure 1 (b) Node size of code graphs

Figure 2: Code graph (PDG) and its node distribution in two data sets *JAVA-2M* and CodeSearchNet

a maximum value of 1, *MMAN* and *GraphCodeBERT* can only reach about 0.452 and 0.713, respectively. Compared to their respective counterpart *DCS* (0.377) and *CodeBERT* (0.693), MRR is improved by only 0.075 and 0.02, respectively.

2.) *They both ignore the potential difficulties and challenges arising from the unique features of code graphs, which contain diverse information and are usually small.* Figure 2(a) shows the program dependency graph (PDG) [11] of the code snippet in Figure 1. Observably, a PDG contains two kinds of structural information, i.e., control dependency (the dashed edge for control flow) and data dependency (the solid edge for data flow), as well as two kinds of attribute information, i.e., the node attribute (the statement in each node) and the edge attribute (the parameter variable on a data dependency edge). The multi-modal model *MMAN* and the pre-trained model *GraphcodeBERT* incorporated the code structure that captures control flow or data flow into their learning task respectively. However, they still may potentially lose much valuable information, since the control flow or the data flow alone can only capture the statement control relationship or the variable relationship. More importantly, the code graphs are often small in size. As Figures 2(b) shows, most of the code graphs (CFGs and PDGs) extracted from two real data sets for code search (CodeSearchNet [22] and the data set we collected JAVA-2M) have only 2-5 nodes and very few of them have more than 20 nodes. The multi-modal model *MMAN* utilized GNNs to embed CFGs as the structural features. However, existing research [19] has shown that GNNs often achieve better performance on the large graphs with 2, 000 to 20, 000 nodes.

The two findings motivate us that the effectiveness of code search could be substantially improved if those unique features of code graph are more fully utilized in the semantic learning model. In this paper, to tackle the problem, we propose a Graph-to-Sequence Converter (*G2SC*). By converting the code graphs to lossless sequences that retain complete graph structure information, *G2SC* enables to address the problem of using *GNN* for small graph learning through sequence feature learning and capture both the edges and nodes attribute information of code graphs. In particular, *G2SC* first converts the code graph into a unique corresponding node sequence by a specific graph traversal strategy. Then, it obtains a

statement sequence by replacing each node with its corresponding statement. A set of carefully designed graph traversal strategies guarantee that the process is one-to-one and reversible. *G2SC* enables capturing rich semantic relationships (i.e., control flow, data flow, node/relationship properties), and provides learning model-friendly data transformation. Further, to show that *G2SC* can be flexibly integrated with the state-of-the-art models in the two research streams, i.e. the multi-modal model and the pre-trained model, and greatly improve their effectiveness, we present two *G2SC* enabled models: *GSMM* (*G2SC* enabled *MMAN* model) and *GSCodeBERT* (*G2SC* enabled *CodeBERT* model). Specifically, *GSMM* adopts the bi-directional Long Short-Term Memory (*BiLSTM* [43]) to learn the structured feature of code graphs from *G2SC* converted sequences followed by an attention layer to integrate the unstructured and structured features together. *GSCodeBERT* replaces the code sequence in the code-description sequence with the *G2SC*-converted sequence to further fine-tune *CodeBERT* model. The replication package of our work has been released at Github[1].

The main contributions of our work are summarized as follows.

1.) We present a Graph-to-Sequence converter (*G2SC*), which transforms the code graphs into lossless sequences. Through *G2SC*, the structural information of code graphs can be effectively learned by using sequence feature learning. To the best of our knowledge, we are the first to introduce the idea into the semantic code search.

2.) We propose two *G2SC* enabled semantic code search models: *G2SC* enabled multi-modal model (*GSMM*) and *G2SC* enabled *CodeBERT* (*GSCodeBERT*). They both demonstrate the generality and applicability of integrating *G2SC* into existing models to improve their capability of utilizing code graphs for semantic code search.

3.) We have conducted extensive experiments to evaluate the proposed *G2SC* enabled models over various real datasets. The results show that *G2SC* improves the state-of-the-art models in the two streams substantially. For example, in terms of MRR, *GSMM* improves *DCS* and *MMAN* by 100% and 63% respectively, while *GSCodeBERT* improves *CodeBERT* and *GraphCodeBERT* by 13% and 11.5% respectively.

Outline. The remainder of this paper is organized as follows. Section 2 introduces the research background of this paper. In Section 3, we present an overview of *G2SC*. Section 4 presents the two *G2SC*-enabled models. In Section 5, we provide experimental evaluations to address our research questions. Finally, Section 6 and Section 7 provide the related work and conclusion respectively.

2 BACKGROUND

In this section, we introduce the overall process of deep learning-based semantic code search and four representative models from the two mainstream research, i.e. *DCS*, *MMAN*, *CodeBERT* and *GraphCodeBERT*, which are selected for comparison in our experiments.

2.1 Deep Learning-based Semantic Code Search

As shown in Figure 3, the overall process of deep learning-based semantic code search includes three main phases: offline training, offline code embedding, and online code search. First, a deep neural network takes a large-scale corpus of $< code, description >$ pairs as input and maps them into a unified high-dimensional vector

[1]https://github.com/G2SMM/G2SC

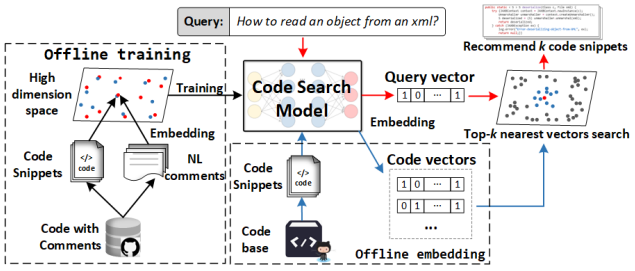


Figure 3: Deep learning based code search process

space that code snippets and descriptions with similar semantics are embedded into nearby vectors of the space (the training phase). Then, the trained model is used to compute a vector for each code snippet in a given codebase that users would like to search (the embedding phase). Finally, an online query (e.g. *How to read an object from xml?*) is also embedded into a vector when it arrives. By computing vector similarity, code snippets whose vectors are similar to the query vector, such as the code snippet in Figure 1, are returned and recommended to users (the online code search phase).

Intuitively, the quality of vector representation learned in the training phase determines the effectiveness of the solutions in performing code search. To return code snippets S that highly match a query Q in semantics, the code search model needs to learn a correlation f between Q and S, namely,

$$f : Q \rightarrow S \tag{1}$$

For this purpose, the model training is expected to produce close representation of code snippets and their corresponding descriptions. Given a set of code snippets S and their natural language descriptions D, the task in this phase can be formulated as:

$$D \xrightarrow{\phi} V_D \longrightarrow J(V_D, V_S) \longleftarrow V_S \xleftarrow{\varphi} S \tag{2}$$

where $\phi : D \rightarrow R^d$ is an embedding function to map a code snippet description D into a d-dimensional vector space V as a vector V_D, $\varphi : S \rightarrow R^d$ is an embedding function to map a code snippet S into the same vector space as V_S. $J(\cdot, \cdot)$ is a measure function (e.g. cosine similarity) to score the similarity between V_D and V_S. During the training, the model is guided by the spirit of producing higher similarity between a code snippet S and its correct description D^+ and lower similarities between S and its incorrect descriptions D^- simultaneously. Once the model is trained, the correlation f between Q and S can be easily set up through their vectors embedded by functions ϕ and φ, respectively.

2.2 Mainstream Models for Code Search

Our *G2SC* aims to improve both multi-modal and pre-trained models. Four representative works from two mainstream solution families are considered in this work, including two multi-modal models *DCS* [13] and *MMAN* [45], and two pre-trained models *CodeBERT* [10] and *GraphCodeBERT* [16].

A. Multi-modal Models

1.) **Deep code search** (DCS): *DCS* is the first deep learning based code search model proposed by Gu et al. It learns the code representation by extracting and fusing three aspects of information, i.e. multi-layer perceptron (*MLP*) for the tokens contained in the

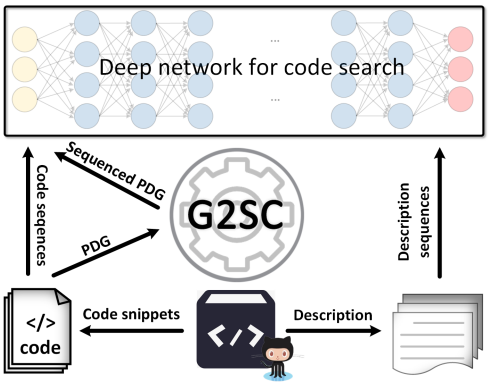


Figure 4: Overall framework of using G2SC for code search

source code (body tokens) and Recurrent Neural Networks (*RNN*) [33] for the method names and the API sequences. However, *DCS* ignores the structure information that the code may contain.

2.) **Multi-modal attention network** (MMAN): *MMAN* is the state-of-the-art multi-modal model proposed by Wan et al, which introduces the code structure information into the code search task [45]. It embeds ASTs by *TreeLSTM*, CFGs by *GGNN* (gated graph neural network) and code sequences by *LSTM* followed by an attention fusion layer to integrate them into a single vector representing the entire code. However, *MMAN* ignores the fact that code graphs are usually small and contain diverse information (e.g. control dependency and data dependency in PDG).

B. Pre-trained Models

1.) **CodeBERT**: *CodeBERT* is the first pre-trained model that can be used in code search proposed by Feng et al. Inspired by BERT [7], it adopts the Transformer architecture and uses two pre-training tasks, *MLM* (Masked Language Model) and *RTD* (Replaced Token Detection), to learn the general model parameters for a set of code-related downstream tasks such as code search, code summarization, etc. Since it is not designed for the code search task alone, the user needs to perform additional fine-tuning to return the query result. However, *CodeBERT* still ignores the code structure information.

2.) **GraphCodeBERT**: *GraphCodeBERT* is the state-of-the-art pre-trained model that can be used in code search proposed by Guo et al. It is an upgraded version of *CodeBERT* introducing the variable graph structure that captures the data flow[16]. Besides the *MLM* task in *CodeBERT*, it uses two more variable graph-specific pre-training tasks, edge prediction and node alignment, as well as a graph-guided masked attention function. However, *GraphCodeBERT* ignores rich semantic information other than the variable relationship, such as the control relationship in Figure 2(a).

3 THE GRAPH-TO-SEQUENCE CONVERTER

The unique characteristics of the code graphs bring great challenges for the existing semantic code search models. To improve their effectiveness, it may be a promising way to ensure that the data can be supported and fit into the learning model nicely. Unfortunately, as mentioned in Section 1, the capability of existing learning models, even the *GNN*, is not so friendly to learn the code graphs.

To tackle this issue, we propose a Graph-to-Sequence Converter (*G2SC*) based solution. Figure 4 presents the overall framework of using *G2SC* in the code search model, which is the core of the entire code search process. Once an effective code search model is trained, we only need to apply *G2SC* in the later two phases in Figure 3, i.e., offline code embedding and online code search, to find code snippets highly matching user-specified natural language queries.

As shown in Figure 4, our solution comprises two main components: 1.) *G2SC module*. It transforms the code graphs into a type of the model-friendly data format, code sequences that can losslessly retain the code graph structure information as well as the key attribute information related to the edges and nodes; 2.) *G2SC-enabled model*. By learning the sequences converted by *G2SC* and other code information, such as code textual information and natural language description, through different carefully-designed strategies, we are able to develop various *G2SC* enabled models.

In this section, we focus on the *G2SC* module. It first converts the code graph into a unique corresponding node sequence by a specific graph traverse. Then, it gets a statement sequence by replacing each node with its corresponding statement. A set of carefully designed graph traversal rules guarantee that the process is one-to-one and reversible. Thus, the graph structure information is retained in the converted sequence in a lossless manner. For ease of demonstration, we will explain the *G2SC* by using PDGs. One reason is that PDG is a more complex code graph with both control dependency and data dependency than others like CFG [11]. Another reason is that PDG is used as the graph feature in our proposed models. Note that *G2SC* can be easily extended to handle different kinds of the code graphs like CFG, etc.

Algorithm 1 provides the details of our graph-to-sequence converter algorithm. In the first step, given a PDG, we need to identify the sequence entry node for the PDG sequence we want to generate. Since PDG only has one root node, the root node v_0 of a PDG can be directly selected as the entry s_0 of the entire PDG sequence (Algorithm 1 line 1). After that, starting from node s_0, we recursively select an edge for expansion according to the following rules in order: 1.) The backtracking edge (the edge whose endpoint has been traversed) is selected before the forward edge (the edge whose endpoint has not been traversed); 2.) When there are multiple backtracking edges or only multiple forward edges, we can either choose control dependency edge or data dependency edge for the traversal after the first step. Either priority can produce a unique result. In our graph-to-sequence algorithm, we set control dependency edges to have higher priority (Algorithm 1 line 4-8); 3.) If the edge still cannot be determined after the second step, we select the edge whose endpoint is of the highest priority as the preferentially expanded edge. In our method, the priority of nodes is determined by the order of their corresponding statements in the code snippet (Algorithm 1 Function Edge_Check); 4.) In particular, if there are no connected edges, the traversal continues back to the nearest ancestor node with at least one edge that has not been traversed (Algorithm 1 line 9-11). During traversing the PDG, we can follow the following two steps to generate our proposed PDG sequence (Algorithm 1 Function Sequence_Expand): ① If the traversed edge e is a control dependency edge, we only add the attribute of the node pointed by e to the resultant PDG sequence S; ② If the traversed edge e is a data dependency edge, we add the attribute of e and the attribute of the node pointed by e to S in turn.

Algorithm 1: Graph-to-sequence Algorithm

Input: Graph $G = \{V, E\}$ $V = \{v_0, \ldots, v_n\}$ $E = \{e_0, \ldots, e_m\}$
Output: Minmum PDG sequence $S = \{s_0, s_1, \ldots\}$

initialize $S = \{\}$ $i = 0$, append v_0 as s_0 into S;
while *E is not NULL* **do**
  select edge e connect with s_i;
  if *e contains backtracking edge* **then**
    select the backtracking edge as e;
    `Sequence_Expand(Edge_Check(e),S);`
  else if *e contains forward edge* **then**
    `Sequence_Expand(Edge_Check(e),S);`
  else if *e is NULL* **then**
    select the nearest father of s_i which has forward edge e' as e;
    `Sequence_Expand(Edge_Check(e),S);`

Function: `Sequence_Expand(e, S)`
if *e is control dependency edge* **then**
  $S \leftarrow \{e \rightarrow v\}$ and remove e from E;
  ($e \rightarrow v$ means the node $v \in V$ that e points to)
  i=i+1;
else if *e is data dependency edge* **then**
  $S \leftarrow \{e, e \rightarrow v\}$ and remove e from E;
  i=i+2;

Function: `Edge_Check(e)`
if *there is only one edge* **then**
  select the edge as e;
else if *there are multiple edges* **then**
  if *there exists multiple control dependency edges* **then**
    compare all control dependency edge $e_x \rightarrow v_x$ and select the minmum condition e_x as e;
  else
    compare all $e_x \rightarrow v_x$ and select the minmum condition e_x as e;

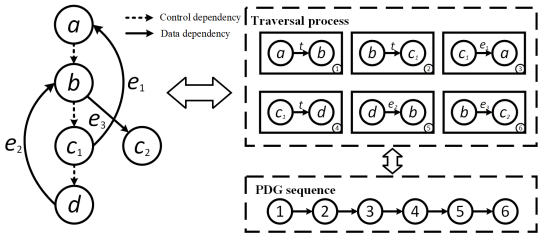


Figure 5: Process of converting PDG to sequence

Here we give an example in Figure 5 to illustrate how to convert a PDG to its unique corresponding sequence. Given a PDG in Figure 5, the order of nodes is the order in which they are executed in the program, the dashed line represents the control dependency edge, and the solid line represents the data dependency edge with its attribute e_i. The first traversal is $a \dashrightarrow b$ since root node a is the program entry. The second traversal is $b \dashrightarrow c_1$ because the control dependency edge needs to be traversed first according to our proposed edge expansion rule. The third traversal is $c_1 \xrightarrow{e_1} a$, where the attribute of the edge is e_1, because we define the backtracking edge has the highest priority. In the fourth traversal, we go back to c_1, so the traversal is $c_1 \dashrightarrow d$. In the fifth traversal, we traverse the backtracking edge from d, which is $d \xrightarrow{e_2} b$, where the attribute of the edge is e_2. After that, since d and c_1 have no subsequent node, we return b for the last traversal of $b \xrightarrow{e_3} c_2$, where the attribute of the edge is e_3. So far, we can obtain the sequence of edges of PDG by the traversal process shown in Figure 5. In the PDG sequence, the control dependency edge has no attributes since it only means the control dependency between two nodes. Then, we use the number in the lower right corner of each box to represent the sequence it corresponds to (e.g. ① represents sequence $\{a, b\}$ and ③ represents sequence$\{c_1, e_1, a\}$). Finally, we can obtain the PDG sequence $< abbc_1c_1e_1ac_1dde_2be_3c_2 >$ by connecting all circles in digital order. Note: the graph-to-sequence algorithm is also reversible. Given the PDG sequence obtained above, we can restore the traversal process by picking two adjacent nodes at a time from scratch (e.g. first $\{a, b\} \rightarrow$ ①, second $\{b, c_1\} \rightarrow$ ②, third $\{c_1, e_1, a\} \rightarrow$ ③ and so on). After that, we only need to reconnect the edges in the order of the numbers in the circle to regain the PDG in Figure 5. Since the process is reversible, no information will be lost when converting the sequence through our graph-to-sequence algorithm. It should be noted that the graph-to-sequence algorithm only requires linear time consumption $O(|E|)$, which means it does not need much time for PDG preprocessing.

4 *G2SC* ENABLED MODELS

The *G2SC* enabled model is the second component of our solution. *G2SC* is designed as a general converter, which can empower various learning models to integrate the code graphs for effective code search. Thus, it can be easily incorporated into other models. In this section, we take the two mainstream models as examples to present two *G2SC* enabled models, namely *GSMM* (*G2SC* enabled multi-modal model) and *GSCodeBERT* (*G2SC* enabled *CodeBERT* model).

4.1 Data Pre-processing

The pre-processing is performed based on a large-scale training corpus. For multi-modal models, we first extract a collection of pairs of code snippets and their corresponding descriptions. For the description part, we extract all the words and sort them by JAVA word split API[2] to get a token sequence. For the code part, we adopt the same method for extracting Body Token (tokens in function body) and Method name (function name) from code snippet. Similar to *DCS*, we treat tokens as unordered ones in the source code. In addition, we extract the PDGs from code snippets through an extraction tool TinyPDG[3]. Then, all PDGs are converted to PDG sequences using our *G2SC* proposed in Section 3. For the pre-trained models, CodeBERT and GraphCodeBERT both provide pre-processing source programs in their downstream tasks. Therefore, we can directly utilize them in our GSCodeBERT.

[2]https://docs.oracle.com/javase/6/docs/api/
[3]https://github.com/YoshikiHigo/TinyPDG

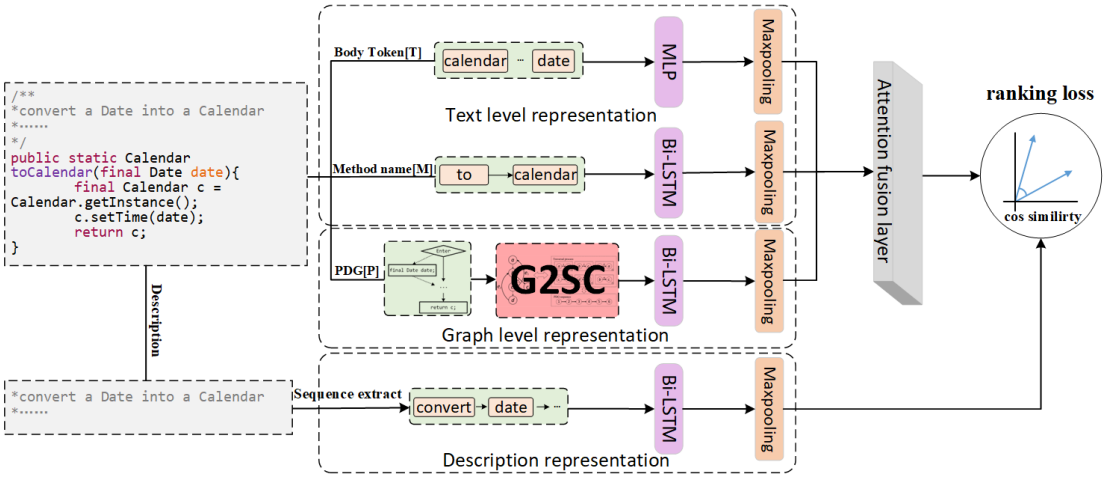


Figure 6: The pipeline of *GSMM*. The red box is our converter *G2SC*.

4.2 *G2SC* Enabled Multi-modal Model

First, we present *G2SC* enabled multi-modal model, *GSMM*. In particular, *GSMM* utilizes *MLP* to learn Body Tokens, *BiLSTM* to learn Method Name and PDG sequence, and adopts an attention mechanism to combine the three code features. The main difference between *GSMM* and other multi-modal models is that *GSMM* uses *G2SC* converted PDG sequence for code graph structure learning.

The overall framework of *GSMM* is shown in Figure 6. The input code snippets is represented as $C = \{c_1, c_2, \ldots, c_n\}$, where c_i is the i^{th} code snippet. For each code snippet c_i, it is denoted as the combination of its method names, Body Tokens and PDG:

$$c_i =< M_i, T_i, P_i > \tag{3}$$

where M_i represents the method name in an ordered sequence of tokens of c_i, T_i is the Body Tokens that is a token set and $P_i = \{V_i, E_i\}$ is the PDG. *GSMM* first embeds the code snippets through three components: *Text level representation*, *Graph level representation* and *Attention-based fusion*, and then collects the embedding of natural language description by the *description representation* component. Finally, the correlation between code snippets and their natural language descriptions is inferred by *model training*. Next, we present the five steps in detail respectively.

Text level representation: The text level representation model of *GSMM* is similar to that of *DCS*. It treats the method name and the token in the code snippet body as the text level. For the method name $M =<m_1, m_2, \ldots, m_{N_M}>$ decomposed as a sequence of tokens, *GSMM* embeds the sequence of camel split tokens using a BiLSTM with maxpooling:

$$\begin{aligned} \overrightarrow{h_t} &= LSTM(\overrightarrow{h_{t-1}}, \omega(m_t)) \quad \forall t = 1, 2, \cdots, N_M \\ \overleftarrow{h_t} &= LSTM(\overleftarrow{h_{t-1}}, \omega(m_{N_M - t+1})) \quad \forall t = 1, 2, \cdots, N_M \\ m &= maxpooling([\overrightarrow{h_1}; \overleftarrow{h_{N_M}}], \cdots, [\overrightarrow{h_{N_M}}; \overleftarrow{h_1}]) \end{aligned} \tag{4}$$

where $\omega()$ is the word embedding layer to embed each method name token m_t into a d-dimensional vector; $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ represent the hidden states of the forward and backward LSTM in BiLSTM, respectively; $[a; b] \in R^{2d}$ represents the concatenation of two vectors. The final hidden state h_t of BiLSTM is jointly represented by the corresponding $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$. A method name is thus embedded as a d-dimensional vector m. As for the Body Tokens $T = \{t_1, t_2, \ldots, t_{N_T}\}$, considering that *DCS* considers them to have no strict order in the source code, we also embed them via a *MLP* layer:

$$\begin{aligned} h_i &= tanh(W^T \omega(t_i)) \quad \forall i = 1, 2, ..., N_T \\ t &= maxpooling([h_1, \cdots, h_{N_T}]) \end{aligned} \tag{5}$$

where $\omega()$ embeds each token t_i into a d-dimensional vector, and W^T is the matrix of trainable parameters. All hidden states h_i are summarized to a single vector t for the Body Tokens by maxpooling.

Graph level representation: Given a PDG sequence $P=<p_1, p_2, \cdots, p_{N_P}>$, which is already processed by our *G2SC*, we adopt another *BiLSTM* with maxpooling to embed it:

$$\begin{aligned} \overrightarrow{h_t} &= LSTM(\overrightarrow{h_{t-1}}, \omega(p_t)) \quad \forall t = 1, 2, \cdots, N_P \\ \overleftarrow{h_t} &= LSTM(\overleftarrow{h_{t-1}}, \omega(p_{N_P - t+1})) \quad \forall t = 1, 2, \cdots, N_P \\ p &= maxpooling([\overrightarrow{h_1}; \overleftarrow{h_{N_P}}], \cdots, [\overrightarrow{h_{N_P}}; \overleftarrow{h_1}]) \end{aligned} \tag{6}$$

where $\omega()$ is the word embedding layer to embed each token p_t in PDG sequence into a d-dimensional vector. In addition, *GSMM* gets the final PDG sequence representation p through maxpooling similar to the method name.

Attention-based fusion layer: Finally, the vectors of three aspects are fused into one vector through an attention fusion layer:

$$\alpha_i = \frac{e^i}{e^m + e^t + e^p} \quad i \in \{m, t, p\} \tag{7}$$

$$c = \alpha_m \times m + \alpha_t \times t + \alpha_p \times p \tag{8}$$

where α_i represents the attention weight of each vector. The output vector c represents the final embedding of the code snippet.

Description representation: For the natural language description $D =< d_1, d_2, \ldots, d_{N_D} >$, *GSMM* embeds the sequence of camel split tokens using a *BiLSTM* with maxpooling:

$$\begin{aligned} \overrightarrow{h_t} &= LSTM(\overrightarrow{h_{t-1}}, \omega(d_t)) \quad \forall t = 1, 2, \cdots, N_D \\ \overleftarrow{h_t} &= LSTM(\overleftarrow{h_{t-1}}, \omega(d_{N_D - t+1})) \quad \forall t = 1, 2, \cdots, N_D \\ d &= maxpooling([\overrightarrow{h_1}; \overleftarrow{h_{N_D}}], \cdots, [\overrightarrow{h_{N_D}}; \overleftarrow{h_1}]) \end{aligned} \tag{9}$$

where $\omega()$ is the word embedding layer to embed each description word d_t into a d-dimensional vector. In addition, *GSMM* gets the final description representation d through maxpooling.

Model training: We follow Gu[13] and Wan[45] to train the *GSMM* model to jointly embed code snippet and description into the intermediate semantic space with similar coordination. The goal of joint representation is that if a code snippet and a description have similar semantics, their embedding vectors are expected to be close to each other. In other words, given an arbitrary code snippet C and an arbitrary description D, *GSMM* predicts the similarity of C and D. During the training, we feed a triple $<C, D+, D->$ into the model. For each code snippet C, there is a positive description $D+$ (a correct description of C) as well as a negative description (an incorrect description of C) $D-$ randomly chosen from the pool of all D's. For any given a set of $<C,D+,D->$ triples, the *GSMM* predicts the cosine similarities of both $<C, D+>$ and $< C,D- >$ pairs and minimizes the hinge range loss:

$$L(\theta) = \sum_{<C,D+,D->\in P} max(0, \epsilon - cos(c, d+) + cos(c, d-)) \quad (10)$$

where θ denotes the model parameters; P denotes the training dataset, ϵ is a slack variable commonly used in machine learning and often set as 0.05; c, $d+$ and $d-$ are the embedded vectors of C, $D+$ and $D-$. The intuition behind the ranking loss is that pushing the model to predict a high similarity between a code snippet and its correct description and a low similarities between a code snippet and incorrect descriptions.

4.3 *G2SC* Enabled *CodeBERT* Model

Next, we present *G2SC* enabled *CodeBERT* model, *GSCodeBERT*. Like other pre-trained models, *CodeBERT* consists of two stages: pre-training and fine-tuning. As the source code of the pre-training *CodeBERT* is not released, we only use *G2SC* in the downstream task of *CodeBERT*.

Figure 7 provides the overall pipeline of *GSCodeBERT*. Each input code snippet is decomposed into two pairs (D, S) and (D, P). D=$<d_1, \cdots, d_n>$ is the sequence of tokens d_i, $i \in \{1, 2, \cdots, n\}$, which are extracted from the natural language description D of code snippet. S=$<s_1, \cdots, s_m>$ is the sequence of tokens s_j, $j \in \{1, 2, \cdots, m\}$, which are extracted from the program language code sequence S. And P=$<p_1, \cdots, p_k>$ is the sequence of tokens p_t, $t \in \{1, 2, \cdots, k\}$, extracted from our *G2SC* converted PDG sequence P. All (D, S) pairs and (D, P) pairs are respectively fed into the pre-trained *CodeBERT* model with a special token $[CLS]$ indicating the semantic relevance between D and S (or P) as input to the downstream fine-tuning task. This enables our *GSCodeBERT* model to increase its understanding of code structure information while retaining *CodeBERT*'s understanding of natural language and code sequence.

Model fine-tuning: we fine-tune the model with a binary classification loss function, where a softmax layer is used to map the output vector *ClassLabel* to a score between $[0, 1]$. A higher score indicates the greater similarity between the code Sequence/PDG Sequence and natural language. Our fine-tuning task consists of two parts. One is for the similarity between natural language description D and the PDG sequence P, and another is between D and the code sequence S. Both of them utilize a similar loss function formulated as follows:

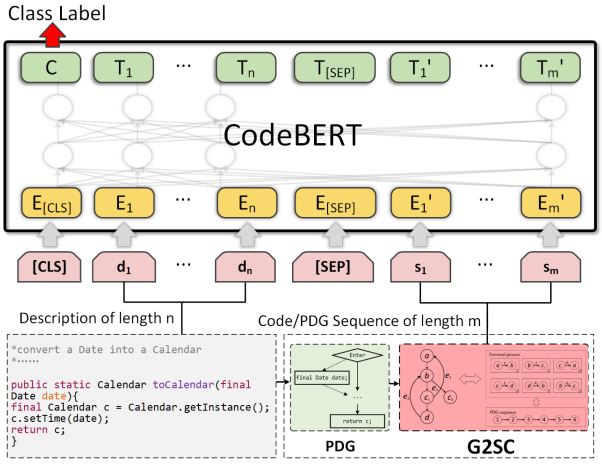


Figure 7: The pipeline of *GSCodeBERT*.

$$L(\theta) = \sum_{<D,X>\in E} max(0, \varepsilon - score([CLS](D, X)) + score([CLS](D, X-)) + score([CLS](D-, X))) \quad (11)$$

where θ denotes the model parameters; E denotes the training set; D denotes natural language description and X can be code sequence S or PDG sequence P; ϵ is a slack variable similar to that in Equation (10); $[CLS]$ is a special token indicating if (D, X) is a positive or a negative sample. All (D, X) pairs are taken as the positive samples. Both $(D, X-)$ and $(D-, X)$ are negative samples, which have the balanced number of instances and are created by randomly replacing D or X in (D, X) with $D-$ or $X-$, respectively. The function $score()$ is used to calculate the similarity. Our loss functions aim to increase (resp. decrease) the similarity score between a sequence X and its correct (resp. incorrect) description D.

4.4 Code Search

After the model is trained, for multi-modal models with a given code base C and a given query q, the target is to rank all these code snippets by their similarities with query q. We first feed the code snippet c into *GSMM* and feed the query q as the description to obtain their corresponding representations, denoted as c and q. Then we calculate the ranking score as follows:

$$sim(c, q) = cos(c, q) = \frac{c^T q}{||c||\ ||q||} \quad (12)$$

where c and q are the vectors of code and query respectively. The higher the similarity, the more semantic-related the code to the query. For each query, *GSMM* returns top-10 most related results.

For pre-trained models, as described in Section 4.3, for each query q, we feed q paired with all code sequence $c \in C$ to the model. The input is formalized as $([CLS]q[SEP]c)$. After CodeBERT processing, the tag embedding corresponding to $[CLS]$ is returned. Further, we calculate the similarity through the trained downstream classifier shown below:

$$sim(c, q) = DT(CodeBERT([CLS](c, q))) \quad (13)$$

where c and q are the sequences of code and a query respectively. $[CLS]$ is a special token corresponding to each (c, q) pair. $DT()$ represents the Downstream Task that scores the Class Label processed by CodeBERT. In addition, the higher similarity indicates a stronger match between the code and the query.

5 EXPERIMENT

5.1 Research Questions

Our experiment is guided by answering the following research questions:

- **RQ1: *H*ow much can *G2SC* improve the state-of-the-art multi-modal model for semantic code search?** We compare our *GSMM* with *DCS* and *MMAN* in code search on two real data sets to prove G2SC can effectively improve the effectiveness of multi-modal models.
- **RQ2: *H*ow much can *G2SC* improve the state-of-the-art CodeBERT-based model for semantic code search?** We boost *CodeBERT* by integrating our PDG sequence and compare our *GSCodeBERT* with the state-of-the-art pre-trained models *CodeBERT* and *GraphCodeBERT*.
- **RQ3: *H*ow do our proposed models behave using other features converted by *G2SC*?** Since there are some other code features that can be used by G2SC, we design a series of experiments to show the benefit of choosing text information and PDG for both the multi-modal model and the pre-trained model.

5.2 Experiment Setup

5.2.1 Data Set. In our experiment, we have used two different datasets including CodeSearchNet and JAVA-2M. Following [13] and [45], we construct our first dataset based on the source codes from CodeSearchNet [22] as our first data set. In particular, the original dataset contains 542,991 java code snippets. After removing code snippets for which their AST, API, CFG, or PDG cannot be extracted, we retain 390,504 snippets as the final training dataset.

To further investigate the models' capability of handling large-scale dataset, following [13, 22], we construct our second larger data set JAVA-2M by following three steps: 1.) We download the Github repositories published from August 2016 to September 2020 with the programming language labeled as Java through Github API. On the basis of [13], we set the minimum number of stars of the repositories as 10 to ensure the code is of substantial quality. 2.) To provide models code snippets with natural language description, we retain those Java methods extracted from the repositories with English annotation before the code snippet. And we remove the method without annotation or annotated by other languages. As a result, each method consists of code snippet and its corresponding natural language annotation. 3.) We extract the API, AST, CFG, and PDG from all the extracted methods, and drop those without these information. In the end, JAVA-2M contains 2,141,921 Java methods.

For the codebase, we use the same code base as CodeBERT provided by Husain et al.[22] and retain the methods with all the API, AST, CFG, and PDG information. These methods are different from the training corpus, as they are considered in isolation and contain codes without descriptions. In the end, we get 1,569,525 method candidates as our code search base.

5.2.2 Experiment Setting. To train GSMM, we set the batch size as 64. We set the size of vocabulary to 10,000 to ensure the word coverage rate of the corpus reaches more than 95%. Inspired by [13], we set the maximum length of method name, natural language and token to 6, 20 and 50 respectively. We truncate the sequence whose length is beyond the maximum length. The sequences below the maximum length are padded with a special token <PAD> to the maximum length. For LSTM unit, we set the hidden size to be 256. Similarly, for *MLP*, the embedding vector dimension is set to 512. We update the parameters by utilizing Adam [24] optimizer. To make the model with better generalization ability, we set the dropout rate as 0.25. All the experiments are implemented using the PyTorch 1.2 with Python 3.7, and the experiments were conducted on a server with 2 Nvidia RTX 2080Ti. We follow the same parameters released by the original authors and apply them to all the considered algorithms for a fair comparison (DCS[4],CodeBERT[5] and GraphCodeBERT[5]).

5.2.3 Evaluation Metric. To measure the effectiveness of the approaches, we employ three widely used metrics including FRank, SuccessRate@k, and MRR [13, 45].

FRank, also known as best hit rank, is the rank of the first hit result in the result list. It is practical measurement as it considers the pattern that users often scan the results from top to bottom. A smaller FRank implies lower inspection effort for finding the desired result.

SuccessRate@k(R@k), also known as success percentage at k, measures the percentage of queries for which more than one correct result could exist in the top-k ranked results. The higher the SuccessRate@k is, the better the code search performance is.

MRR denotes the mean of the reciprocal ranks of results for a given set of queries Q. The reciprocal rank of a query is the inverse of the rank of the first hit result. And the higher the MRR value is, the better is the code search performance.

5.2.4 Evaluation Methodology. For RQ1, when comparing multi-modal models, we follow *DCS* and use human judgement. For human judgement, we use the 50 Java questions provided by *DCS*. The manual analysis was performed independencyly by 3 graduate students with 3-5 years of experience in Java and the developers performed an open discussion to resolve conflict grades for the 50 questions. To test the statistical significance, we also apply Wilcoxon signed-rank test ($p < 0.05$) for the comparison of FRank for all the queries. We conservatively treat the FRank as 11 for queries that fail to obtain relevant results within the top 10 returned results. The p-values for the comparisons are all lower than 0.05, indicating the improvement of *GSMM* over the related approaches is significant.

For RQ2, note that *CodeBERT* was pre-trained on CodeSearchNet, we also use the CodeSearchNet to train other pre-trained models to facilitate a fair comparison. Then we employ the same automatic evaluation method that was used to evaluate original *CodeBERT*.

For RQ3, since both multi-modal models and pre-trained models are considered, we use human judgement and automatic evaluation simultaneously.

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

5.3 Answer to RQ1

To answer RQ1, we measure the effectiveness difference between *GSMM*, *GSMM*-w/o.A(*GSMM* without attention)), *DCS* and *MMAN* on two real data sets CodeSearchNet and JAVA-2M.

Table 1: *GSMM, DCS* and *MMAN* search effectiveness on CodeSearchNet

Model	R@1	R@5	R@10	MRR
DCS	0.20	0.36	0.48	0.26
MMAN	0.22	0.40	0.50	0.29
GSMM-w/o.A	0.24	0.40	0.52	0.33
GSMM	**0.34**	**0.58**	**0.62**	**0.47**

Table 2: *GSMM, DCS* and *MMAN* search effectiveness on JAVA-2M

Model	R@1	R@5	R@10	MRR
DCS	0.20	0.42	0.66	0.31
MMAN	0.26	0.56	0.74	0.38
GSMM-w/o.A	0.34	0.74	0.78	0.49
GSMM	**0.50**	**0.80**	**0.86**	**0.62**

Table 1 shows the performance of *GSMM*, *DCS* and *MMAN* on CodeSearchNet. The results demonstrate that *GSMM* consistently outperforms *MMAN* and *DCS* on all the evaluation metrics. For example, *GSMM* performs better than *MMAN* by 62%, 55%, 45% and 24% in terms of MRR, R@1, R@5 and R@10, respectively. *GSMM* outperforms *DCS* by 81%, 70%, 61% and 29% respectively in terms of MRR, R@1, R@5 and R@10.

We conduct the experiment on JAVA-2M by following the same setting. As shown in Table 2, *GSMM* shows significant improvement compared with *DCS* and *MMAN* on all the metrics. For example, *GSMM* improves *MMAN* by 92%, 43%, 16% and 63% on R@1, R@5, R@10 and MRR respectively while it outperforms *DCS* by 150%, 90%, 30% and 100% respectively in terms of R@1, R@5, R@10 and MRR. Further, the FRank statistics in Table 3 shows that *GSMM* has less NF (not found) than the two baselines. This means *GSMM* can answer more questions, and the correct code snippets recommended by *GSMM* are ranked higher in the recommended list.

To further investigate the performance improvement of *GSMM*, we compare *GSMM*-w/o.A, the version for *GSMM* without attention mechanism, with the baselines shown in Table 1 and Table 2. From the results, we can see that even without attention mechanism, *GSMM* still outperforms other baselines like *DCS*, *MMAN*. For example, *GSMM*-w/o.A is 27% and 14% higher than *DCS* and *MMAN* respectively in terms of MRR on the CodeSearchNet dataset, and 29% and 58% higher than *DCS* and *MMAN* in terms of MRR on the JAVA2M dataset. This further verifies the improvement of *GSMM*'s search effectiveness is not only from attention mechanism but also the features learned from code graphs.

Moreover, the results show that GSMM outperforms the two baselines significantly. The potential reason for *GSMM* outperforming DCS could be that *GSMM* learns richer semantic information from the code graph using *G2SC* while *DCS* cannot utilize code graph information. The reason why *GSMM* is better than *MMAN* is twofold. First, the size of code graphs are usually small. As demonstrated in Section 1, *GNN*s generally perform better on large-scale graphs. Benefiting from *G2SC*, *GSMM* can address this problem by performing a specific traversal of the code graphs and convert them into sequences in a lossless manner that can be easily learned by *BiLSTM*. Another reason is *MMAN* does not utilize node attribute information in the code graph by *GNN*. The attribute of each node in the code graph is the statement corresponding to the node, the information of which is exactly what attribute graph embedding needs. *G2SC* can not only keep the structure information between nodes, but also the order of tokens inside the node. Thus, the information of code statements can also be retained intact.

Answer to RQ1: GSMM outperforms the state-of-the-art model *MMAN* and *DCS* significantly in terms of all the evaluation metrics for code search.

Table 3: FRank list of first 50 Java questions to *GSMM, MMAN* and *DCS* on *JAVA-2M*

Query	DCS	MMAN	GSMM
1. convert an inputstream to string	6	1	1
2. create arraylist from array	1	1	1
3. iterate through a hashmap	NF	6	7
4. generate random integers in a specific range	1	4	4
5. converting string to int in java	NF	NF	NF
6. initialization of an array in one line	7	6	2
7. how can I test if an array contains certain value	6	8	5
8. lookup enum by string value	NF	NF	1
9. breaking out of nested loops in java	NF	NF	NF
10. how to declare an array	6	5	4
11. how to generate a random alpha-numeric string	2	3	2
12. what is simplest way to print a java array	NF	NF	NF
13. sort a map by values	1	1	1
14. fastest way to determine if an integer's square root is an integer	NF	NF	NF
15. how can I concatenate two arrays in java	4	5	5
16. how do I create a java string from the contents of a file	1	1	1
17. how can I convert a stack trace to a string	1	1	5
18. how do I compare strings in java	4	2	1
19. how to split a string in java	NF	4	2
20. how to create a file and write to a file in java	4	3	1
21. how can I initialize a static map	5	5	5
22. iterating through a collection, avoiding concurrent modification exception when removing in loop	NF	NF	4
23. how can I generate an md5 hash	1	1	1
24. get current stack trace in java	2	1	1
25. sort arraylist of custom objects by property	2	1	1
26. how to round a number to n decimal places in java	NF	5	1
27. how can I pad an integers with zeros on the left	NF	NF	NF
28. how to create a generic array in java	7	6	4
29. reading a plain text file in java	1	2	1
30. a for loop to iterate over enum in java	NF	NF	NF
31. check if at least two out of three booleans are true	NF	NF	NF
32. how do I convert int to string	NF	NF	1
33. how to convert a char to a string in java	NF	NF	1
34. how do I check if a file exists in java	6	3	1
35. java string to date conversion	1	1	1
36. convert inputstream to byte array in java	2	1	2
37. how to check if a string is numeric in java	NF	8	1
38. how do I copy an object in java	1	1	1
39. how do I time a method's execution in java	5	3	3
40. how to read a large text file line by line using java	6	4	1
41. how to make a new list in java	NF	NF	1
42. how to append text to an existing file in java	4	1	1
43. converting iso 8601-compliant string to date	1	1	1
44. what is the best way to filter a java collection	9	8	3
45. removing whitespace from strings in java	10	7	1
46. how do I split a string with any whitespace chars as delimiters	7	5	5
47. in java, what is the best way to determine the size of an object	5	2	1
48. how do I invoke a java method when given the method name as a string	7	7	2
49. how do I get a platform dependency new line character	9	6	1
50. how to convert a map to list in java	NF	NF	2

5.4 Answer to RQ2

To explore if our *G2SC* can improve pre-trained models, we choose two recent and well-known models *CodeBERT* and *GraphCodeBERT* for comparison. As described in Section 4.3, *GSCodeBERT* is fine-tuned based on both code sequences and the PDG sequences converted by *G2SC*.

Table 4: *CodeBERT*, *GraphCodeBERT* and *GSCodeBERT* results on CodeSearchNet

Model	R@1	R@5	R@10	MRR
CodeBERT	0.630	0.878	0.917	0.741
GraphCodeBERT	0.649	0.875	0.916	0.751
GSCodeBERT	**0.792**	**0.890**	**0.920**	**0.837**

Table 4 presents the performance of *CodeBERT*, *GraphCodeBERT*, and our approach *GSCodeBERT*. The results show *GSCodeBERT* achieves a significant better performance than *CodeBERT*, e.g., by 13% and 25.7% in terms of MRR and R@1. The potential reason may be *CodeBERT* can further leverage the structure information in code snippet to learn a better mapping between code snippets and natural language with the support of *G2SC*. And compared with *GraphCodeBERT*, *GSCodeBERT* achieves better performance on all the metrics consistently, e.g., 11.5% in MRR and 22% in R@1. The reason may be *GraphCodeBERT* uses a self-created variable graph that only contains the structure information between variables. Differently, our PDG sequence carries richer information such as control dependency and data dependency of the code snippet.

> *Answer to RQ2:* By integrating *G2SC* into *CodeBERT*, the performance can be significantly improved for code search, even better than *GraphCodeBERT* by large margin in terms of R@1 and MRR.

5.5 Answer to RQ3

To explore the potential best features combination in our models, we design an experiment of comparing commonly used code features with multiple combinations. In particular, we consider five types of code features, text, API, AST, CFG, and PDG. We use *G2SC* to convert code structural representation into code sequences and use the BiLSTM for feature embedding. For AST, we follow [20] and preserve the properties of each node to obtain the sequence properties of the AST rather than the original sequence of the code. For AST and CFG, since none of their edges has attributes, we ignore the related judgments of edges attributes in *G2SC*.

Table 5 presents the comparison between five features and four combinations in *GSMM*. The results show that *GSMM* achieves the best performance with PDG among all. Moreover, we find that both CFG and PDG perform better than code Text information. This means that graphs contain richer semantic information. However, we can still observe that Text contribute its own to performance. In particular, the combination "Text+PDG" performs the best among all the considered combinations.

In Table 6, *GSCodeBERT* (Text) denotes the original *CodeBERT*. We observe the similar result with *GSMM*, the combination "Text+PDG" achieves the best performance while "Text+CFG" performs the second best. In particular, the combination "Text+PDG" outperforms "Text" by 13% and 25.7% in terms of MRR and R@1.

Table 5: Comparison of features and their combination in *GSMM*

Features	R@1	R@5	R@10	MRR
Text(MethodName+Token)	0.16	0.40	0.50	0.26
API	0.10	0.26	0.40	0.17
AST	0.06	0.20	0.28	0.14
CFG	0.14	0.40	0.52	0.27
PDG	**0.16**	**0.42**	**0.58**	**0.29**
Text+API	0.20	0.42	0.66	0.31
Text+AST	0.18	0.38	0.60	0.29
Text+CFG	0.32	0.70	0.74	0.47
Text+PDG	**0.34**	**0.74**	**0.78**	**0.49**

Table 6: Comparison of features and their combination in *GSCodeBERT*

Features	R@1	R@5	R@10	MRR
GSCodeBERT(Text)	0.630	0.878	0.917	0.741
GSCodeBERT(Text+API)	0.704	0.877	0.907	0.783
GSCodeBERT(Text+AST)	0.644	0.862	0.902	0.742
GSCodeBERT(Text+CFG)	0.776	0.884	0.913	0.826
GSCodeBERT(Text+PDG)	**0.792**	**0.890**	**0.920**	**0.837**

> *Answer to RQ3:* Adopting *G2SC* algorithm to convert PDG is more effective than other code features in performing semantic code search. Moreover, the combination of feature Text and PDG can achieve promising results in both multi-modal models and pre-trained models.

5.6 Threats to Validity

In this paper, we analyse the performance of all models on CodeSearchNet and multi-modal models on *JAVA-2M*. The two real data sets both are collected from Github. The training sets contains the source codes with the corresponding description, while the search code base provided by [22] contains all the source codes (including codes without description). According to the report in [13], we believe the threat of overfitting for this overlap is not significant. The 50 queries collected from Stack Overflow can further meet the real-world search, and those are not descriptions of code snippets used for training.

In our experiments, the human evaluation of code snippets is manually done and could suffer from human bias. To mitigate this threat, we have taken the below two prevention measures: 1.) we randomly divide the code snippets into three parts on average and distribute to three developers; 2.) the developers hold an open discussion to mitigate the manual bias on 50 queries.

6 RELATED WORK

Code search. In the code search task, many studies focus on API recommendation [14, 15, 31, 55]. Li [28], Van [44] and Nguyen

[35] represented the code snippet as an API set, and represented the API features based on Word2vec model to recommend the appropriate API. Chan et al. Recently, with the growth of open-source code repositories such as GitHub [13], it becomes quite common for developers to search relevant code snippets based on their own requirements. Some code retrieval tools have appeared [4, 25, 26, 36, 54]. Traditional code searching tools are based on keyword matching. In particular, code snippets and natural language queries are regarded as token sets, and appropriate code snippets are recommended according to the similarity of keywords [5, 18, 32, 42]. Lu [30] et al extended the query semantically through WordNet. Gvero [17] et al. allow mixed input in English and Java, and constructed a probabilistic context-free grammar for Java constructs and library invocation. Keivanloo [23] et al. used a code cloning detection model to support spotting working code examples. With the successful development of deep learning, Gu et al. [13] first applied deep learning technology in code search and proposed *DCS* model. Wan et al. [45] proposed *MMAN* to further introduce the graph information of the code into the code retrieval task. In addition to code features (API[8], AST[12, 51] and graph[45]) and attention [9], researchers also improved the code search model from other directions. Yao et al. [52] introduced reinforcement learning mechanism and improved the search effect of existing code search models by combining with them, the performance of reinforcement learning model alone is not as good as *DCS*. Ye et al. [53] connected code summary task with code retrieval task, and learned the two tasks simultaneously through dual learning method to improve the effect of code summary and code search.

Deep code representation. In software engineering, there are also other code related works [6, 21, 37, 38, 50], such as bug location [27, 27], code summary [1, 46, 48], clone detection [49] and code completion [41]. The introduction of code structure information is also applied in these deep code representation tasks. Code2vec [3] converted code snippets into ASTs, extracted path information of all leaf nodes, and learned code features through attention mechanism to predict the function method names. Mou et al. [34] used convolution neural network based on tree structure to capture the features of neighbour nodes in AST, and obtained the semantic information for program classification and source code similarity detection. Allamanis et al. [2] took AST as the structure backbone, added data flow information and side information on the basis of AST to transform AST into a graph containing more information, and applied GGNN to embed it for code variable misuse task. Wang et al. present CodeT5[47] for code related generation task, which is a pre-trained model based on T5[39]. In addition, CodeBERT [10] and GraphCodeBERT [16] both designed a pre-trained model to learn the relationship between programming language and natural language. They can be applied to various tasks in code representation through different downstream tasks, such as the code search task described in this paper.

7 CONCLUSION AND FUTURE WORK

In this paper, to better learn the code graphs, we proposed *G2SC*, an algorithm that can convert graphs to lossless sequences for semantic code search. *G2SC* transforms the graph into a special sequence that is able to retain the graph information for representation learning. To the best of our knowledge, this is the first time to introduce such a graph to sequence algorithm into the semantic code search task, which can be effectively learnt using the deep neural network to capture the structure information on the data representation. Our experimental results show that by inducing *G2SC* into multi-modal model, *GSMM* outperforms the state-of-the-art multi-modal models significantly in terms of code search effectiveness. Additionally, we added *G2SC* to the downstream task of the pre-trained model and achieved a great improvement in code search performance.

In the future, we will further try to apply our *G2SC* to other kinds of code search models. And we will try to investigate how well our proposed G2SC can cooperate with those models.

ACKNOWLEDGMENTS

This research / project is supported by National Natural Science Foundation of China (62032013), Singapore's National Research Foundation (NRF's) National Cybersecurity R&D Grant (GC2018-NCR-0008), and Science and Technology Development Fund of Macau (0047/2020/A1), and the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. We also thank the reviewers for their helpful comments. Yuhai Zhao is the corresponding author.

REFERENCES

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.
[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International conference on machine learning*. 2123–2132.
[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[4] 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.
[5] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empirical Software Engineering* 17, 4-5 (2012), 424–466.
[6] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).
[7] 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).
[8] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning for Semantic Code Search. *arXiv preprint arXiv:2107.04773* (2021).
[9] Sen Fang, You-Shuai Tan, Tao Zhang, and Yepang Liu. 2021. Self-Attention Networks for Code Search. *Information and Software Technology* (2021), 106542.
[10] 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 *(Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.
[11] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[12] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 483–494.
[13] 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.

[14] 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.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734* (2017).

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[17] Tihomir Gvero and Viktor Kuncak. 2015. Interactive synthesis using free-form queries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 689–692.

[18] 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.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[20] 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.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

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

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Koders 2020. https://koders.co/.

[26] Krugle 2020. https://www.krugle.com//.

[27] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 476–481.

[28] Xiaochen Li, He Jiang, Yasutaka Kamei, and Xin Chen. 2018. Bridging semantic gaps between natural languages and APIs with word embedding. *IEEE Transactions on Software Engineering* (2018).

[29] Jinting Lu, Ying Wei, Xiaobing Sun, Bin Li, Wanzhi Wen, and Cheng Zhou. 2018. Interactive query reformulation for source-code search with word relations. *IEEE Access* 6 (2018), 75660–75668.

[30] 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.

[31] 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.

[32] 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.

[33] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. Makuhari, 1045–1048.

[34] 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).

[35] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[36] ohloh 2020. https://www.openhub.net/.

[37] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 547–553.

[38] 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).

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[40] Mohammad Masudur Rahman and Chanchal K Roy. 2016. QUICKAR: Automatic query reformulation for concept location using crowdsourced knowledge. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 220–225.

[41] 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.

[42] Steven P Reiss. 2009. Semantics-based code search. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 243–253.

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215* (2014).

[44] Thanh Van Nguyen, Anh Tuan Nguyen, Hung Dang Phan, Trong Duc Nguyen, and Tien N Nguyen. 2017. Combining word2vec with revised vector space model for better code retrieval. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 183–185.

[45] 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.

[46] 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.

[47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[48] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems*. 6563–6573.

[49] 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.

[50] Xin Xia and David Lo. 2017. An effective change recommendation approach for supplementary bug fixes. *automated software engineering* 24, 2 (2017), 455–498.

[51] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 342–353.

[52] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*. 2203–2214.

[53] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*. 2309–2319.

[54] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 956–961.

[55] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 266–277.