electronics

MDPI

Article

CogCol: Code Graph-Based Contrastive Learning Model for Code Summarization

Yucen Shi [1], Ying Yin [1,*], Mingqian Yu [1] and Liangyu Chu [2]

[1] School of Computer Science and Engineering, Northeastern University, No. 195 Chuangxin Road, Shenyang 110169, China; shiyucen@stumail.neu.edu.cn (Y.S.); 20226525@stu.neu.edu.cn (M.Y.)
[2] School of Medicine and Bioinformatics Engineering, Northeastern University, No. 195 Chuangxin Road, Shenyang 110016, China; 20227274@stu.neu.edu.cn
* Correspondence: yinying@cse.neu.edu.cn

Abstract: Summarizing source code by natural language aims to help developers better understand existing code, making software development more efficient. Since source code is highly structured, recent research uses code structure information like Abstract Semantic Tree (AST) to enhance the structure understanding rather than a normal translation task. However, AST can only represent the syntactic relationship of code snippets, which can not reflect high-level relationships like control and data dependency in the program dependency graph. Moreover, researchers treat the AST as the unique structure information of one code snippet corresponding to one summarization. It will be easily affected by simple perturbations as it lacks the understanding of code with similar structure. To handle the above problems, we build **CogCol**, a **Co**de **g**raph-based **Co**ntrastive **l**earning model. CogCol is a Transformer-based model that converts code graphs into unique sequences to enhance the model's structure learning. In detail, CogCol uses supervised contrastive learning by building several kinds of code graphs as positive samples to enhance the structural representation of code snippets and generalizability. Moreover, experiments on the widely used open-source dataset show that CogCol can significantly improve the state-of-the-art code summarization models under Meteor, BLEU, and ROUGE.

Keywords: code summarization; code graph representation; contrastive learning

Citation: Shi, Y.; Yin, Y.; Yu, M.; Chu, L. CogCol: Code Graph-Based Contrastive Learning Model for Code Summarization. *Electronics* **2024**, *13*, 1816. https://doi.org/10.3390/electronics13101816

Academic Editor: Arkaitz Zubiaga

Received: 6 April 2024
Revised: 26 April 2024
Accepted: 4 May 2024
Published: 8 May 2024

1. Introduction

In the software development circle, more than 50% time is spent on software maintenance. In this period, developers often need to repeatedly read existing code and work on code-related tasks such as code reuse and modification. Code snippets with summarization can enhance the code readability by providing high-quality natural language annotations, enabling developers to quickly understand code functionality and reduce misinterpretations. Moreover, this greatly improves software maintenance efficiency and saves software development costs.

Code summarization tasks can be categorized into statement-level [1], function-level [2–4], and file-level [5] summarization, corresponding to explaining specific statement meanings, function functionalities, and document purposes, respectively. In this paper, we focus on the function-level code summarization. Existing methods often treat code summarization as a translation task like natural language processing, where a code snippet (programming language) is input as the source language into a translation model to obtain the target natural language (usually English). However, traditional machine translation models struggle with long-distance dependencies and performance limitations [6–11]. Additionally, with the exponential growth of code files in open-source software repositories like GitHub, it has become easier to access code snippets with high-quality natural language summarization [12], which increases the demand for larger datasets and better performance in neural

machine translation models. More research has begun to use deep learning-based translation models for code summarization. The process of code summarization based on neural machine translation is illustrated in Figure 1. Taking the Java programming language as an example, the complete code snippet of the function convert() is input into the neural machine translation model in sequence form. The encoder learns the features(green and red boxes) of the code to generate high-dimensional representations, which are then decoded to translate into the target natural language sequence, such as "Converts an array of integers to string".

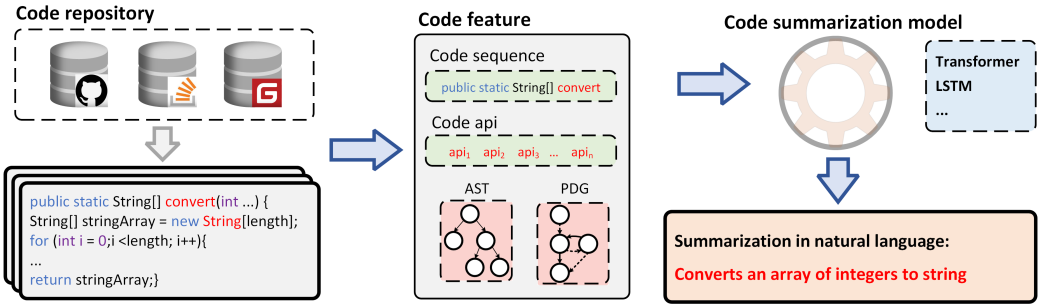

Figure 1. Deep learning-based code summary model.

Since code is highly structured, using sequence learning models like Transformer to treat code as a single sequence often results in the loss of structural information. As shown in Figure 1, current code summarization models that consider structural information typically choose Abstract Syntax Trees (ASTs) as the representation of code structure. For example, state-of-the-art AST-trans [13] extracts the AST of function-level code as the structural information of the code. However, Transformer cannot directly handle structural information (such as tree or graph information). Therefore, Tang et al. optimized the attention mechanism in the Transformer for learning AST and used a unique preorder traversal sequence of AST as input for feature learning. However, compared to Program Dependency Graphs (PDG), which contain advanced semantic information (control and data dependencies), AST only represents certain syntactic information in the code. Moreover, existing code representation methods only consider the transformed graph or code sequence as the sole representation information and still lack understanding of codes with similar structures. Thus, similar code structures and semantics may not be better understood during the code representation phase and they will be easily affected by simple perturbations.

To address these issues, we propose a code summarization model called CogCol (**Co**de **g**raph-based **Co**ntrastive **l**earning model) based on the contrastive learning of code graphs. CogCol leverages Transformer [14] models for learning the graph structure information of PDGs while improving the understanding of structurally similar codes through contrastive learning of PDG sequences. Firstly, CogCol designs a sequence traversal method based on G2SC [15] to obtain enhanced PDG (E-PDG) sequences as representations of code information. Then, considering that any missing node in the code graph may have a significant impact on code semantics, CogCol proposes a masking strategy based on four strategies for the edges in the code graph (relationships between code statements) to obtain positive samples similar to the enhanced E-PDG representation in structure. Finally, based on the obtained positive sample information, CogCol constructs a Transformer-based code graph contrastive learning model for code summarization. Experimental results on open-source datasets comparing six models demonstrate that CogCol achieves the best performance in code summarization tasks.

Our research makes significant contributions in the following key areas:

(1) We propose a code summarization model called CogCol based on code graph contrastive learning. CogCol can improve the understanding of code summaries for structurally similar code while using a sequence-learning model for learning the graph structure information through contrastive learning on graph sequences.

(2) Regarding the edges in the code graph (relationships between code statements), we introduce an edge-masking strategy based on four perspectives to obtain and enhance positive samples with similar structural information as code graphs. Furthermore, based on the positive sample information, we build a Transformer-based code graph contrastive learning model for code summarization tasks.

(3) Experiments on six models in open-source datasets show that CogCol achieves the best performance in code summarization tasks. Specifically, compared to the state-of-the-art code summarization model, AST-trans, CogCol shows an improvement of 4.1% in BLEU-1, 1.6% in Rouge-L, and 6.1% in Meteor.

2. Related Work

2.1. Code Summarization

Traditional code summarization approaches typically involve extracting textual information from code, such as keywords and APIs [6–10]. These methods, while saving time by not requiring training, face challenges in understanding both code and natural language semantics. These methods employ manually defined machine rules or search-based techniques to generate code summarizations. The former often results in comments that deviate significantly from human-generated comments, while the latter struggles to improve accuracy due to the use of alternative code descriptions. However, with the advancement of deep translation models, an increasing number of researchers are now designing code summarization methods based on deep learning models, achieving impressive results.

Iyer et al. [16] treated the code summarization task as a generation task and introduced the first deep learning-based code summarization generation model, Code-NN. Code-NN initially treats code as text, segments it based on semantics, forming a sequence of sequentially connected words, and represents the code as a linear order sequence. Subsequently, the linear order sequence is fed into a Long Short-Term Memory (LSTM) encoder to embed the code into a high-dimensional vector space. Finally, combining attention mechanisms, the decoder generates summarizations for the code. Hu et al. [2] build the semantic segmentation of code and represent the code by parsing the sequence of API calls from the perspective of the API call sequence. They recognize that both code text and APIs provide textual information for the code and summary code by using a sequence-to-sequence translation model. Considering the highly structured nature of code, Hu et al. [3] proposed a code summarization generation model based on AST. As natural language translation models cannot handle tree structures, Hu et al. introduced a structured traversal method to convert the code AST into a tree traversal sequence. This sequence is then input into a sequence-to-sequence model, with code summarization generation carried out based on attention mechanisms. Wan et al. [17] enhanced code summarization generation by introducing reinforcement learning on top of AST. Similarly, Tang et al. [18] transformed the code AST into a sequence and designed a model, AST-trans, based on a Transformer for learning structural information in tree-shaped structures. Cai et al. [19] also used code AST for code representation but avoided traversal-related losses by employing a Tree-LSTM for learning. Choi et al. [20] parsed code into an AST and improved it into a modified AST (m-AST) by connecting all child nodes of each node based on their occurrence order. Treating m-AST as a graph, Choi et al. employed graph convolutional networks (GNN) for convolutional learning. Wu et al. [21] further enhanced AST representation by adding relationship edges based on the code's conduct flow and variable flow. By constructing a graph based on the AST, they also use GNN for code representation and summarization. Table 1 shows the classification of current code summarization models and their applied techniques.

Table 1. Existing code summarization methods.

Features	Code Text (Sequence, API)	Abstract Syntax Tree (AST)
Machine rules	[6–10]	
Sequence learning	[2,16]	[3,18]
Tree learning		[17,19]
Graph learning		[20,21]

2.2. Deep Code Representation

In software development, numerous other endeavors focus on code representation [22–25], such as bug location [26], clone detection [27] and code search [12]. Similar to code summarization, most other traditional deep code representation models also treat code as textual information and utilize natural language processing models for processing [5].

Considering that code is highly structured, many researchers in different tasks are beginning to explore various methods to introduce code structural information. Code2vec [28] converted code snippets into ASTs, extracted path information of all leaf nodes, and learned code features through an attention mechanism to predict the function method names. Mou et al. [29] used a convolution neural network based on tree structure to capture the features of neighbor nodes in AST, and obtained the semantic information for program classification and source code similarity detection. MMAN [30] and GSMM [15] are both models that use structural information for code search tasks. Wan et al. use GNN to learn CFG and TreeLSTM to learn AST and create the code retrieval model MMAN. Shi et al. demonstrate that GNN is weak for code graph learning and convert the code graphs to unique graph sequences by the G2SC algorithm and use sequence learning models like LSTM and BERT for code structural learning. Allamanis et al. [31] took AST as the structure backbone, added data flow information and side information based on AST to transform AST into a graph containing more information, and applied GGNN to embed it for code variable misuse task. Liu et al. [32] proposed TAILOR, a graph-neural-network-based approach to detect functionally similar code snippets. They summarize the program's syntactic and semantic features into a code property graph to identify similar functionalities. Yadavally et al. [33] introduce NEURALPDA for the program dependence analysis of complete and partial code. They use intra-statement context learning and inter-statement context learning to achieve high accuracy in generating CFG/PDGs and obtain great performance on vulnerability detection for partial code snippets. GraphCodeBERT [34] designed a pre-trained model to learn the relationship between programming language and natural language. It can be applied to various tasks in code representation through different downstream tasks, such as the code search task and code clone detection task.

3. Proposed Algorithm

In order to ensure that code summarization tasks can effectively utilize the structural information of the code and leverage the efficient embedding capability of the Transformer, in this paper, we propose CogCol for accurate code Summarization. The overall framework of CogCol is shown in Figure 2.

As shown in Figure 2, firstly, for function-level code snippets (represented by the red block), we use TinyPDG to extract the PDG p_1 in the code snippet. CogCol then performs data augmentation on the extracted PDG to obtain various positive samples corresponding to the code PDG for subsequent contrastive learning. For all PDGs and their corresponding positive samples e_1, the enhanced Graph-to-Sequence Converter (E-G2SC) is utilized to obtain their respective graph sequences (E-PDG sequence). Subsequently, Transformer is employed to embed all graph sequences into feature matrices M_{p1}, and the embedding vectors of all tokens in the matrix are max-pooled to form the vector feature representation v_{p1} of the function, which is appended to the end of the embedding matrix. Contrastive learning losses are designed for the PDG vector v_{p1} and all augmented PDG vectors v_{E1} to optimize the embedding representation of the code. Finally, the corresponding code

is summarized in natural language using the decoder based on Transformer. In training phase, the model is jointly adjusted using decoding loss and contrastive learning loss.

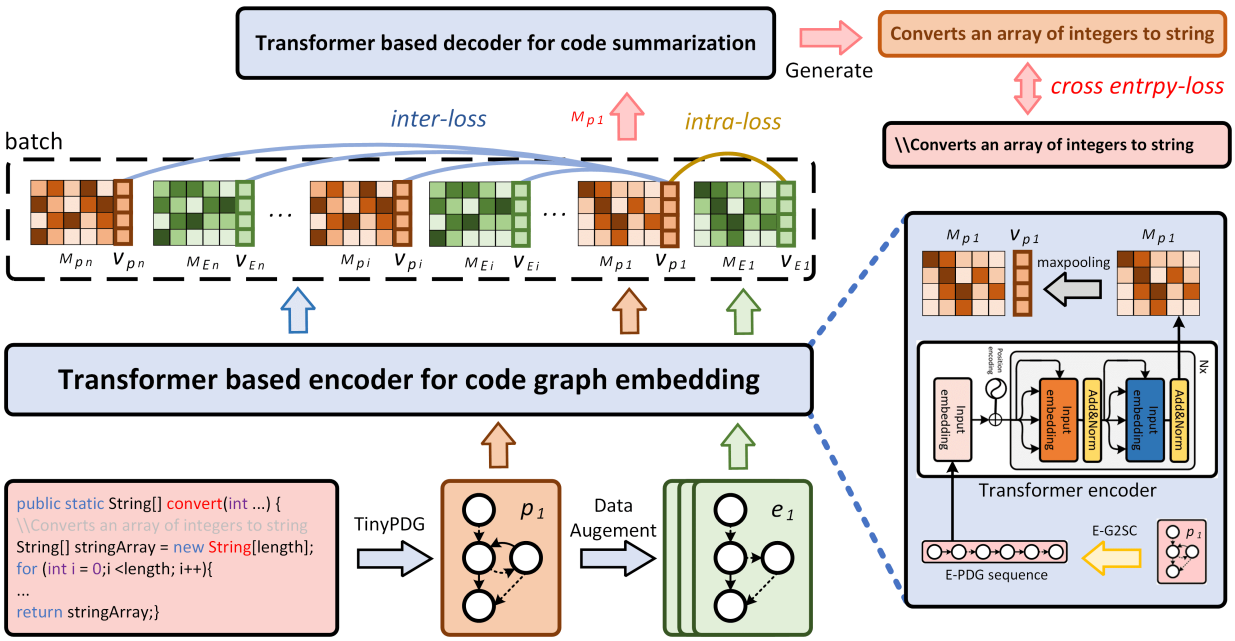

Figure 2. Overall framework of CogCol.

3.1. Background and Basics Knowledge

3.1.1. Code Graph and Code Sequence

Program Dependence Graph (PDG) [35] is a graphical representation method used to represent dependencies between code statements. PDG effectively captures two types of dependencies in the program, including control dependency and data dependency. PDG can help developers analyze, understand, and optimize programs.

As shown in Figure 3, on the left is a function code snippet getCharacter(), and on the right is its corresponding PDG. The nodes in the PDG represent basic blocks or statements in the program. Each node contains a basic operation performed during program execution, such as assignment statements, conditional statements, etc. Two types of edges in the PDG represent different types of dependencies. (1) data dependency edges: If the execution of one node requires data produced by another node, there exists a data dependency edge. Solid arrows in Figure 3 represent data dependency edges, such as the statement **String** key having a data dependency edge to the statement **Object val == getValue(key)**. (2) control dependency edges: If the execution of one node is influenced by the control flow of conditional statements, there exists a control dependency edge. The dotted arrows in Figure 3 represent control dependency edges, such as the statement **return null** being control-dependent on the statement **val == null**.

A Control Flow Graph (CFG) is used to represent the sequential control flow relationship during program execution. As shown in the middle of Figure 3, the nodes in the CFG represent basic blocks in the program, which are groups of statements executed sequentially. CFG has only one type of edge representing control flow transitions. For example, based on the control flow of the **if(val == null)** statement, there are two executable blocks **return null** and **return Java.Types.CHARACTER.convert(val)** afterward. CFG is mainly used to analyze the control flow structure of programs, such as detecting loops, identifying unreachable code, etc.

The graphical representations of PDG and CFG provide powerful tools for software engineers and researchers to understand the structure and behavior of programs. They also serve as effective tools for program analysis, optimization, and debugging. In practical applications, the graphical structures of different languages can be generated and visualized

using open-source tools. In this paper, our research primarily relies on the JAVA PDG generated by the TinyPDG (https://github.com/YoshikiHigo/TinyPDG accessed on 5 April 2024) tool for analysis. An overview of the pseudocode for extracting CFG and PDG is outlined in Algorithm 1.

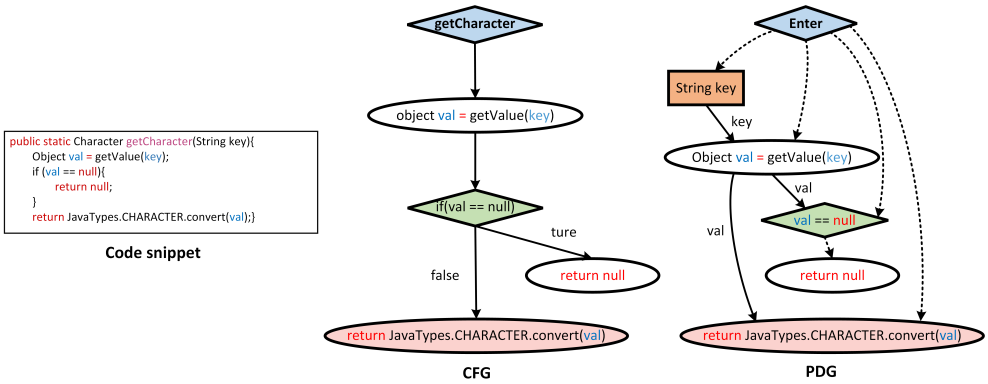

Figure 3. Function-level code getCharacter() and its corresponding CFG and PDG.

Algorithm 1: Extraction for CFG and PDG

Input: Code snippet c
Output: CFG g_c, PDG g_p

initialize $g_c = \{\}$ $g_p = \{\}$;
g_c = `Build_CFG();`
g_p = `Build_PDG();`

Function: `Build_CFG(`c`)`

```
cfg = empty_graph();
current_block = None;
while line in code is not empty do
    if line is branch then
        cfg.add_node(line);
        if current_block is not None then
            cfg.add_edge(current_block, line);
        current_block = line;
    if current_block is None then
        cfg.add_node(line)
    else
        cfg.add_node(line);
        cfg.add_edge(current_block, line);
        current_block = None;
return cfg
```

Function: `Build_PDG(`c`)`

```
pdg = empty_graph();
while line in code is not empty do
    pdg.add_node(line);
    if line is branch then
        while dependent_line in get_dependent(line) is not empty do
            pdg.add_edge(dependent_line, line);
    while data_dependency in get_data_dependencies(line) is not empty do
        pdg.add_edge(data_dependency, line);
return pdg
```

3.1.2. Transformer and Contrastive Learning

Transformer is a deep learning architecture model used for sequence-to-sequence learning tasks, which is powerful for code-related tasks [36]. In this paper, we focus on its embedding layer, which maps each element in the input sequence to a high-dimensional vector representation. In the Transformer model, each element of the input sequence (such as text sentences or time series) is first embedded into a high-dimensional space. This embedding process is performed by the input embedding matrix. To preserve the order of the input sequence, the Transformer introduces position encoding, which encodes the position information as vectors and adds them to the word embeddings. The word embeddings and position encodings in the input embedding matrix are combined by addition, incorporating both semantic and positional information into the embedding representation. The input embedding matrix is obtained by adding word embeddings and position encodings, with dimensions [sequence length, embedding dimension]. The matrix obtained by Transformer has dimensions of $n \times d$, where n is the length of the sequence and d is the corresponding dimension of the vector matrix. Through the embedding matrix, Transformer maps each element of the input sequence into a high-dimensional vector space, providing abstract representations of the input. This representation helps the model capture complex relationships between inputs and is used in the subsequent decoding of neural network layers.

Contrastive learning [37] is a widely used deep learning algorithm that learns by comparing the similarity between two or more samples. As shown in Figure 4, contrastive learning typically involves three main concepts: anchor, positive sample, and negative sample. The goal of contrastive learning is to optimize the model to increase the similarity between positive samples and anchors in high-dimensional space while decreasing the similarity between negative samples and anchors. For example, in Figure 4, the anchor code snippet performs a sorting algorithm(function sort() with summarization "sort an array"), so it should have higher similarity with positive samples(green box) implementing reverse sorting in high-dimensional space and lower similarity with negative samples(red box) implementing Fibonacci sequence generation. The loss function of contrastive learning can be formulated as follows:

$$L(A, P, N) = max(0, margin - similarity(A, P) + similarity(A, N)) \quad (1)$$

where A represents the anchor vector, P and N represent the corresponding positive and negative samples, respectively. The function $similarity()$ measures the similarity between the anchor and the sample, typically calculated using cosine similarity. Additionally, there is always a margin term, denoted as margin, which ensures that the similarity between positive samples and the anchor is greater than the similarity between negative samples and the anchor. Contrastive learning compares the similarity and dissimilarity between samples, effectively enhancing the generalization of the model.

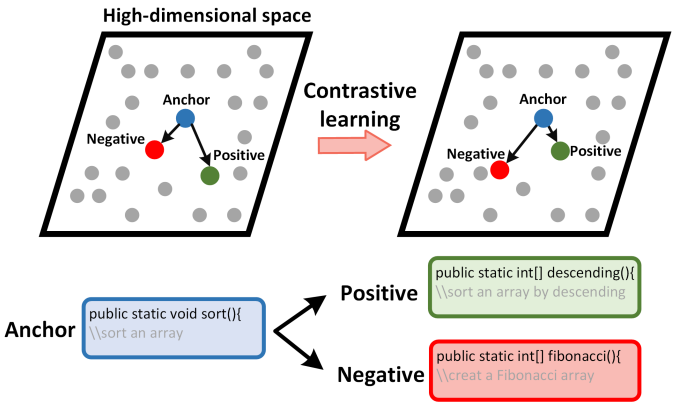

Figure 4. Process of contrastive learning.

3.2. Code Graph-Based Contrastive Learning Model

3.2.1. Contrastive Learning Sample Construction Based on PDG

In this section, we will first introduce the method of enhancing PDG to obtain E-PDG and the traversal method to obtain the sequence of E-PDG. Then, we will detail how this paper obtains the four types of PDG-related contrastive learning positive samples.

Enhanced Program Dependence Graph (E-PDG) Construction Based on Code Control Flow Graph: As shown in Figure 5, the left two graphs represent the CFG and PDG corresponding to a functional code snippet "getCharacter()". PDG contains two types of edges between statements: control dependency edges and data dependency edges. In the PDG of Figure 5, the function body contains only one statement "return null", which has a control dependency relationship with "val == null". This control dependency relationship expresses that the execution of the target statement must pass through the execution of the source statement. The remaining statements have control dependencies on the program entry point "Enter", meaning there are no control dependency relationships between the other statements. During program execution, although the execution of some statements is independent of others, the overall execution of the code still follows the order in which the user wrote it. Therefore, in this paper, we replace the dependency relationships between statements inside the function body in PDG with control flow relationships in the CFG. The replaced E-PDG is shown on the right side of Figure 5 and follows three steps: (1) all data dependency relationships are retained in E-PDG (solid directed edges with attributes, such as $\xrightarrow{key}$, $\xrightarrow{value}$); (2) all control dependency relationships between statements inside the function body (solid directed edges without attributes) are retained; (3) the original directed dashed lines starting from the entry of code will be removed, and these weak executions dependent on the program entry point will be replaced by the control flow in the CFG. For example, as shown in Figure 5, according to step (3), the dashed edge between node <0>→<4> in PDG will be removed and according to CFG a dashed edge between node <2>→<4> will be added to E-PDG, and according to step (2), the dashed edge starting from <2>→<3> will be saved as a solid edge and another dashed edge will further be added according to CFG. Finally, according to step (1), all data dependency edges will be retained. Following the above steps, E-PDG preserves the dependency information of the code while further enhancing the understanding of the code's execution flow, enabling a richer expression of the code's structural information.

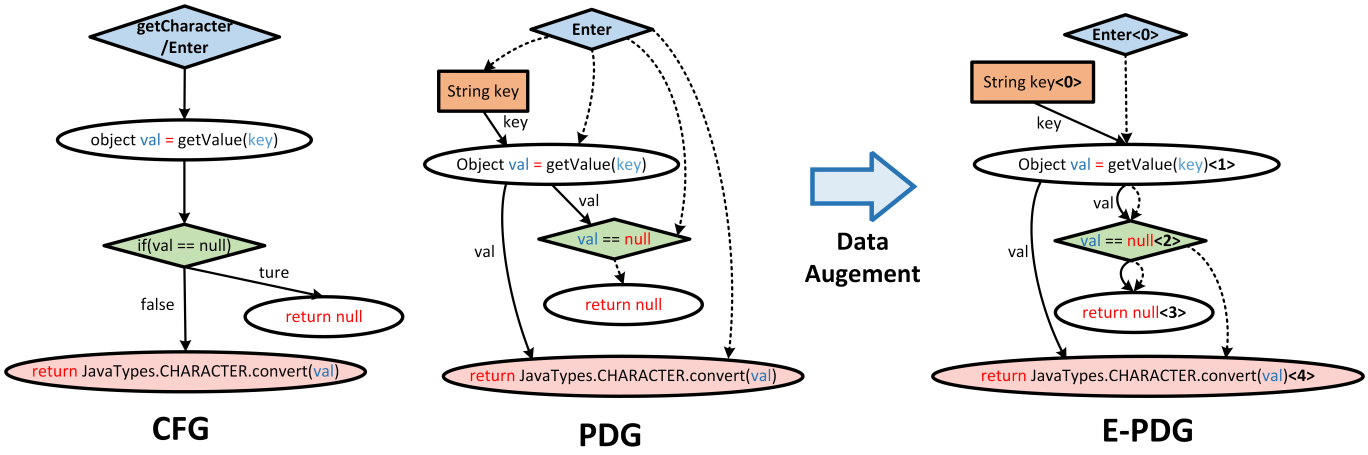

Figure 5. Process of constructing E-PDG by CFG and PDG.

E-PDG Traversal Algorithm: Based on the code graph transformation algorithm G2SC proposed by Shi et al. [15] to convert code graphs into unique sequences, to preserve the structural dependency information of the graph, we incorporate the traversal of execution dependency edges into the original G2SC. Specifically, the improved E-G2SC execution for E-PDG follows the following rules: (1) back edges are prioritized over forward edges;

(2) when multiple back edges and forward edges appear simultaneously, control dependency edges are prioritized over data dependency edges, and execution dependency edges have the lowest priority; (3) when both conditions (1) and (2) are satisfied, select the node with the smaller numeric identifier of the target node as the highest priority. Overall, it follows the traversal rules of G2SC and incorporates execution dependency edges into the traversal process to obtain a unique corresponding E-PDG sequence.

Construction of Contrastive Learning Samples Based on E-PDG: As shown in Figure 6, to facilitate the explanation of the process of generating E-PDG, we assume that the left side of Figure 6 represents an E-PDG corresponding to a function-level code snippet. Unlike constructing contrastive learning samples in fields such as images, masking any statement (node in PDG) in the code will have a significant impact on the semantics of the code. Therefore, when constructing contrastive learning samples, we do not consider node masking in E-PDG. We design the following four strategies (a)–(d) for generating positive samples for contrastive learning based on the structural information in E-PDG:

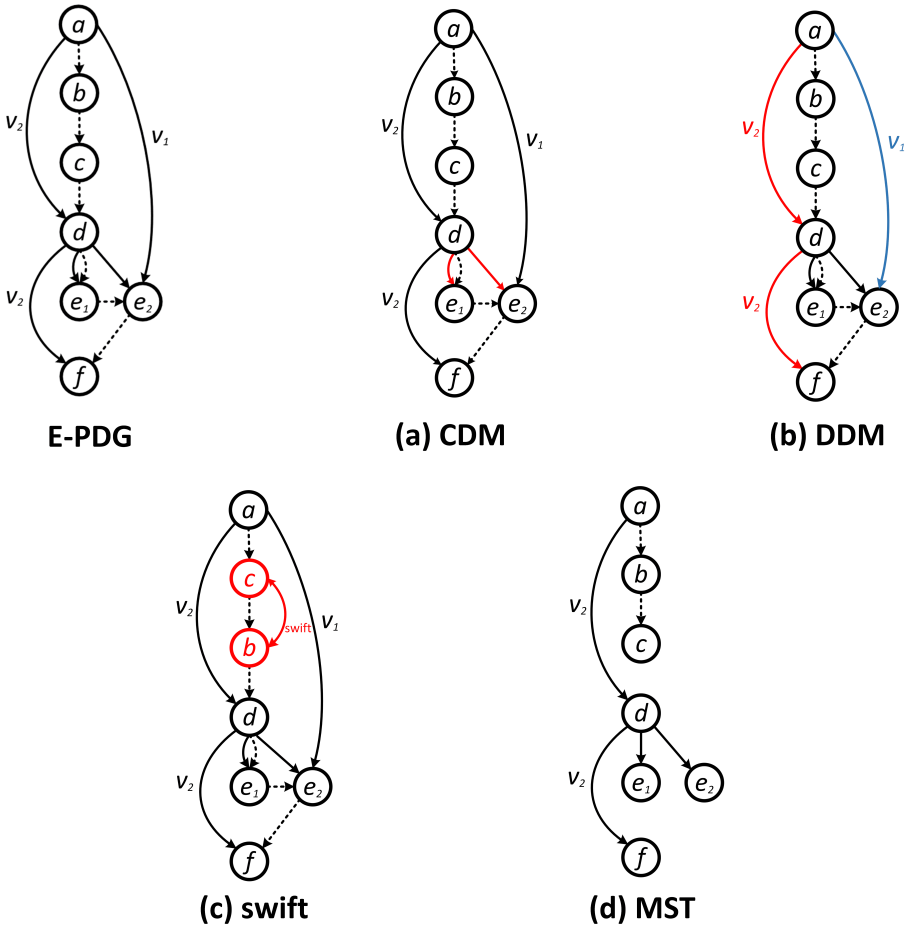

Figure 6. E-PDG and its corresponding contrastive learning samples.

(a) Control Dependency Masking (CDM): As shown in Figure 6a, there are two control dependency edges $d \rightarrow e_1$ and $d \rightarrow e_2$ in the PDG. Considering different coding habits of users for implementing the same method, users may achieve different control dependency relationships using different conditions and loop statements, resulting in functions implemented with the same statements possibly having different control dependency relationships. Therefore, when masking control dependencies, one of all control dependency edges is randomly selected for masking. As shown in Figure 6a, one of the two solid red directed edges $d \rightarrow e_1$ and $d \rightarrow e_2$ will be randomly deleted to serve as a positive sample for CDM in contrastive learning. CDM is designed to enhance the model's general

understanding of control dependencies, rather than being limited to the format of control dependencies provided by the training data.

(b) Data Dependency Masking (DDM): As shown in Figure 6b, there are three data dependency edges $a \xrightarrow{v_1} e_2$, $a \xrightarrow{v_2} d$, and $d \xrightarrow{v_2} f$ in the E-PDG. To perform data dependency masking, all variables in the code are standardized first. For example, variables x, y, i used by users will be standardized as v_1, v_2, v_3 according to the order of variables in the program. Considering the different usage habits of variables by users when implementing the same function, in data dependency masking, the variable with the highest frequency of occurrence is selected, and all data dependency edges related to that variable are masked as positive samples for DDM in contrastive learning. The most frequent variable path reflects the user's variable usage habits to the greatest extent, thus masking all paths related to that variable can reduce the bias of the model towards different users' variable usage habits. As shown in Figure 6b, two solid red directed edges with attribute v_2, $a \xrightarrow{v_2} d$ and $d \xrightarrow{v_2} f$, will be deleted to serve as positive samples for DDM in contrastive learning. DDM is designed to reduce the impact of users' variable usage habits in the training set on the model's learning of code features.

(c) Statement Position Exchange (Swift): As shown in Figure 6c, for two nodes in the E-PDG only connected by execution dependency but not connected by other nodes according to control and data dependencies, swapping the positions of statements in Swift does not affect the two types of dependencies in the original PDG. Considering that the positioning of certain elements such as API declarations or variable definitions in code writing does not affect the function semantics, Swift can mitigate the impact of user coding order habits on the model during training.

(d) Minimum Spanning Tree (MST): As illustrated in Figure 6d, MST ensures a tree-like structure with the minimum number of edges while preserving all nodes in the graph. Therefore, we believe that the MST of the E-PDG represents the minimum number of relationships required to construct functions using the same set of statements among different users. In this paper, the E-PDG traversal principle preserves the edges in the graph based on the priority order of data dependency, control dependency, and execution dependency edges. MST enhances the model's ability to represent different structural implementations of the same code functionality using the same set of statements.

3.2.2. Contrastive Learning Model for Code Structure Sequence Samples

Through Section 3.2.1, we can obtain processed E-PDG and its corresponding sequences. To learn the graph features of E-PDG, we use the Transformer encoder to transform the graph sequences into high-level embedding. During the training phase, the symbol corresponding to the E-PDG sequence is denoted as $P = \{p_1, p_2, \ldots, p_n\}$, n represents the batch size. And the positive sample graph sequences for contrastive learning corresponding to E-PDG are denoted as $E = \{e_{11}, e_{12}, e_{13}, e_{14}, \ldots, e_{n3}, e_{n4}\}$. E is four times larger than P because each p contains four positive samples. The embedding of p and e will be v_p and v_e

To embed all graph sequences, p and e, for any given sequence $t \in \{p, e\}$, they are embedded according to the following formula:

$$M_t = Transformer_{en}(t) = softmax(\frac{QK^T}{\sqrt{d_k}}) \tag{2}$$

$$v_t = maxpooling(M_t) \tag{3}$$

where M_t represents the attention matrix after $Transformer_{en}()$ embedding, which represents the sequence features. $softmax()$ is the normalization function, d_k is the dimension of queries Q and keys K, and Q, K, V are linear mapping matrices related to t. v_t represents the vector feature representation of the graph sequence, which will be used for subsequent contrastive learning loss calculation. $maxpooling()$ is the maximum pooling function used to transform the vector matrix M_t into vector form.

After obtaining the unique graph sequences corresponding to all graphs using the above method, we propose the following loss function to train the contrastive learning module based on the inter-loss and intra-loss between samples:

$$L_{ctr} = -\frac{1}{K}\sum_{i=1}^{K} log(\frac{e^{\frac{sim(v_p, v_{e_i})}{\tau}}}{e^{\frac{sim(v_p, v_{e_i})}{\tau}} + \sum_{j=1}^{N} e^{\frac{sim(v_p, v_{n_j})}{\tau}}}) \quad (4)$$

where $sim(\cdot)$ represents the similarity calculation between two vectors, with cosine similarity used in this paper. τ is the temperature parameter used to scale the model output. v_{n_j} represents the negative sample vector from the negative candidate set N, which is composed of other E-PDGs within the same batch and their positive samples. The number of negative samples selected is usually the same as the number of positive samples. K represents the batch size.

In addition to the contrastive loss, the decoding loss formula incorporates the joint representation of matrices from the code and vectors from contrastive learning as the attention matrix required for Transformer decoding. Therefore, the decoding loss formula is as follows:

$$L_{com} = Transformer_{de}(concat(M_t, v_t)) \quad (5)$$

where $concat()$ represents the function for concatenating matrices and vectors. $Transformer_{de}()$ represents the decoding module of the Transformer. In this paper, we use the decoding module of Transformer. By constructing the aforementioned loss functions, the model is trained using the following overall loss function:

$$L = \alpha L_{ctr} + (1 - \alpha) L_{com} \quad (6)$$

where α is the joint training parameter between the loss functions to balance the weights of the two losses in the model. Through the training losses designed in CogCol, we aim to use contrastive learning to make the model more suitable for learning structured code language features and to perform more accurate code summarization tasks using more accurately expressed code features.

4. Experimental Results and Discussion

In this section, we will introduce the experimental dataset, experimental metric, experimental method, and experimental results.

4.1. Experimental Dataset

In this paper, we utilize the newest dataset provided by Hu et al. in their TL-CodeSum model research [2], which is widely used in code summarization studies. Hu et al. processed and segmented high-quality open-source projects from GitHub in the years 2015–2016 to obtain various code snippets along with their corresponding natural language summaries. The dataset underwent PDG extraction using TinyPDG and filtering for code without identifiable graph structures. Subsequently, all the code was tokenized, and all tokens were standardized to lowercase format. Table 2 shows the information of the processed dataset, with the original dataset quantities indicated in parentheses.

Table 2. Dataset in this paper.

	Total Set	**Training Set**	**Validation Set**	**Test Set**
Number	64,825 (69,708)	51,861	6482	6482

Data are available at https://github.com/xing-hu/TL-CodeSum (accessed on 5 April 2024).

As shown in Table 2, we filtered out 4883 (7%) code-summarization pairs. This includes instances where certain graph structures such as CFG and PDG were unidentifiable, as well

as code snippets that were either too long or too short to their corresponding summaries. During the model training and evaluation partitioning, we split the data by 8:1:1, providing 80% of the data for training, while the remainder was used for model validation and testing. Furthermore, we conducted the token features of code and their corresponding natural language summaries, the statistical results are shown in Table 3.

Table 3. Statistical results of dataset.

	Average Token	Unique Token	Length > 20	Length > 100	Mid.Length
Code	120.16	66,650	-	33.18%	68
Summarization	17.73	46,895	23.01%	-	13

As shown in Table 3, the average token count (length) of code is 120.16, with a median of 68 tokens. Additionally, 33.18% of the code snippets have a length exceeding 100 tokens, indicating generally high code quality in this dataset and possibly containing rich semantic and structural information. The average token count of summarization is 17.73, with a median of 13 tokens, and 23.01% of comments have a token count exceeding 20. These statistics suggest that natural language summaries typically require a significant amount of vocabulary to describe code functionality.

4.2. Experimental Metric

In this paper, we employ three widely used evaluation metrics in code summarization: BLEU, Rouge-L, and Meteor. The detailed explanations of each metric are as follows:

BLEU (Bilingual Evaluation Understudy) [38] is a widely used metric for automatically evaluating the quality of machine translation. BLEU calculates scores by comparing the overlap of n-grams between the model-generated translation and the reference translation. BLEU considers both phrase matching and accuracy, and a higher score indicates better model performance. The calculation formula for BLEU is as follows:

$$\text{BLEU} = BP \times exp(\sum_{n=1}^{N} \omega_n \times log(p_n)) \tag{7}$$

where N is the maximum order of n-grams, p_n is the precision of n-gram exact matches between system output and reference translation, ω_n is the weight of n-grams, and BP is the brevity penalty. In this paper, in addition to the widely used value of 4 for n-grams, we also include values of 1 and 2 as references.

Rouge (Recall-Oriented Understudy for Gisting Evaluation)-L [39] is used to evaluate the quality of text summaries, particularly focusing on the overlap between the summaries generated by the model and the reference summaries. ROUGE-L is based on the Longest Common Subsequence (LCS), measuring the similarity between long sequences in the generated summary and the reference summary. A higher ROUGE-L score indicates a stronger summarization capability of the model. The calculation formula for ROUGE-L is as follows:

$$\text{Rouge-L} = \frac{\text{LCS}(C, R)}{len(R)} \tag{8}$$

where $\text{LCS}(C, R)$ represents the length of the longest common subsequence between the model output C and the reference summary R, and $len(R)$ is the length of the reference summary.

Meteor (Metric for Evaluation of Translation with Explicit Ordering) [40] is a comprehensive metric for evaluating translation quality. Meteor combines various aspects of evaluation, considering exact matches, stem matches, synonym matches, and other criteria. Meteor can overcome the limitations of BLEU evaluation and provide a more

comprehensive reflection of translation quality. The calculation formula for Meteor is as follows:

$$\text{Meteor} = (1 - \beta) \times precision + \beta \times Recall \tag{9}$$

where β is a parameter balancing precision and recall. The calculation of precision and recall includes factors such as exact matches, stem matches, and others. In this paper, the value of β is 0.5.

4.3. Experimental Setup

We select seven models for experimental comparison, including Code-NN [16], Tree2Seq [41], RL + Hybrid2Seq [17], DeepCom [3], API + Code [2], SIT3 [19], mAST + GCN [20] and AST-trans [13]. Considering the potential impact of experimental preprocessing on the results, to ensure fairness, for the open-source models DeepCom, SIT3, and AST-trans were used with the parameters provided in the respective articles, we have tried our best to fine-tune these models to achieve optimal performance.

To train the deep models, we set the batch size to 32. The Adam optimizer was used for parameter updates during training. To enhance the model's generalization capability, we employ a dropout rate of 0.25. We set the learning rate to 1×10^{-3}, the training time was approximately 20 h, and the maximum number of epochs for training was set to 400. We set the maximum input length to 512, sequences longer than this length will be truncated. We use the early stopping mechanism to avoid overfitting, following AST-trans, we set the patience to 20. Our model is based on Transformer 4.17.0. All experiments were implemented using PyTorch 1.2 and Python 3.7 and conducted on a GPU server equipped with two Nvidia RTX 2080Ti GPUs. For the models reproduced in this paper, initial parameters provided by each model were used based on adhering to the proposed data processing procedure and fine-tuned accordingly.

4.4. Experimental Results

In order to evaluate the effectiveness of the code summary model CogCol based on code graph contrastive learning, our experiments were guided by answering the following research questions:

- **RQ1: Can CogCol achieve the best performance compared to state-of-the-art code summarization models?**
- **RQ2: Can each module in CogCol effectively enhance the performance of code summarization?**
- **RQ3: What is the impact of the different parameter settings on CogCol?**

4.4.1. Answer to RQ1

To verify whether CogCol can achieve the best performance in current research on code summarization, we compared CogCol with eight mainstream code summarization models. To ensure the fairness of the experiments, we reproduced the results of open-source models on the processed dataset. The comparison results are shown in Table 4.

When compared with non-structural information-based models such as Code-NN, RL + Hybrid2Seq, and API + Code, CogCol demonstrates significant improvement. For instance, compared to the best-performing model API + Code, CogCol achieves an increase of 10.12%, 6.79%, and 22.33% in BLEU-4, Rouge-L, and Meteor scores, respectively. Compared to RL+Hybrid2Seq, CogCol shows improvements of 33.79%, 25.58%, 20.28%, 7.49%, and 27.6% in BLEU-1, BLEU-2, BLEU-4, Rouge-L, and Meteor scores, respectively. Non-structural models overlook the potential semantics contained within code structure. CogCol effectively represents code features by enhancing code graphs and transforming them into graph sequences for learning. Additionally, the introduction of contrastive learning further enhances model generalization, leading to better summarization results.

In comparison with models based on code structural information (AST) such as Tree2Seq, SIT3, and AST-trans, CogCol still shows significant improvements. For example,

compared to the state-of-the-art model AST-trans, CogCol outperforms it by 4.1%, 2.3%, 1.0%, 1.6%, and 6.1% across five metrics. Moreover, as reported in various code summarization studies [42–44], BLEU alone is insufficient to reflect the semantic alignment between machine translation results and actual results. Therefore, a comprehensive evaluation with metrics like Rouge-L and Meteor is necessary to assess translation effectiveness. In this regard, CogCol's improvements over AST-trans are more pronounced, with a 1.6% increase in Rouge-L and a 6.1% increase in Meteor scores. Overall, CogCol effectively enhances the effectiveness of AST-trans in code summarization tasks. AST primarily expresses the syntactic information of the code, thereby having limited capability in conveying code semantics. To overcome the limitation of AST, CogCol utilizes PDG to represent relationships between statements, such as control and data dependencies. Furthermore, the PDG enhancement algorithm proposed in this paper helps the model to further learn knowledge similar to the original code's structural information. Furthermore, contrastive learning further strengthens the model's generalization to provide better code summarization.

Table 4. Comparing CogCol with code summarization models (The parentheses indicate the improvement compared to the state-of-the-art code summarization model, AST-trans.).

Models	BLEU-1	BLEU-2	BLEU-4	Rouge-L	Meteor
Code-NN	30.06	29.77	27.60	41.10	12.61
Tree2Seq	40.49	38.62	37.88	51.50	22.55
RL + Hybrid2Seq	40.33	39.13	38.22	51.91	22.75
DeepCom	43.65	41.27	39.78	51.92	24.01
API + Code	46.95	42.91	41.31	52.25	23.73
SIT3	50.77	46.81	45.03	54.42	27.13
mAST + GCN	50.90	46.85	45.17	54.71	27.17
AST-trans	51.82	48.04	45.49	54.91	27.37
CogCol	**53.96 (4.1% ↑)**	**49.14 (2.3% ↑)**	**45.97 (1.0% ↑)**	**55.80 (1.6% ↑)**	**29.03 (6.1% ↑)**

4.4.2. Answer to RQ2

To answer RQ2, we conducted experiments by respectively removing the contrastive learning module from CogCol and replacing the graph learning module with the original code sequence learning module (proposed by G2SC). Furthermore, to investigate the impact of different positive samples on the contrastive learning module, experiments were conducted separately on four types of positive contrastive learning samples. In the positive sample ablation experiment, to ensure balance in contrastive learning, the number of negative samples during training was reduced to 25% of the original. Table 5 shows the results of CogCol using only each module.

In Table 5, "Transformer-base" represents the code summarization performance using only the Transformer model. "w/o Graph" indicates the performance of CogCol without using PDG and contrastive learning, after fine-tuning based on the Transformer. "w/o CL" represents the performance on top of the Transformer where the code sequence is replaced with E-PDG sequences. For all three models mentioned above in Table 5, the performance drops significantly compared to CogCol. For instance, in BLEU-4, the performance of the aforementioned three models decreased by 4.4%, 3.8%, and 3.4%, respectively. Similarly, in Meteor, they dropped by 4.7%, 3.9%, and 2.9%. This indicates that both the contrastive learning module and graph structural information are essential components of the model and significantly enhance the effectiveness of code summarization. "CL only CDM" represents the contrastive learning model in CogCol, using only control flow masking. The subsequent three models represent using only data flow masking, only statement swift, and only the minimum spanning tree for contrastive learning, respectively. The four models using only one sample for contrastive learning show some improvement

compared to CogCol without contrastive learning (CogCol-w/o CL). Among them, the model "CL only Swift" using only statement swapping shows the best improvement, with enhancements across all five metrics compared to the CogCol model without contrastive learning. Although "CL only MST" shows a relatively small improvement compared to the CogCol model without contrastive learning, combining the four types of positive samples in the model "CogCol" yields the best performance regardless of which module is used. In conclusion, each module in CogCol effectively enhances code summarization, and the combination of multiple modules achieves the best summarization results.

Table 5. Ablation study for CogCol.

Models	BLEU-1	BLEU-2	BLEU-4	Rouge-L	Meteor
Transformer-base	49.92	46.27	43.91	53.16	27.66
–w/o Graph	50.46	46.31	44.18	53.93	27.91
–w/o CL	50.87	46.73	44.37	54.42	28.20
–CL only CDM	50.94	47.03	45.12	54.97	28.33
–CL only DDM	51.06	47.12	44.54	55.18	28.29
–CL only swift	51.16	47.26	45.19	55.28	28.47
–CL only MST	50.95	46.98	44.49	54.62	28.37
CogCol	51.82	47.79	45.94	55.80	29.03

4.4.3. Answer to RQ3

To explore the impact of the temperature parameter, the ratio of positive to negative samples in contrastive learning, the weight parameter in contrastive learning and joint loss with decoding, and the CogColg's performance under different epochs, we conducted experiments with different parameter settings while keeping other parameters fixed. Each experiment indicator was represented using different colored lines and coordinate systems. Figures 7–9 show the experimental results under different parameters.

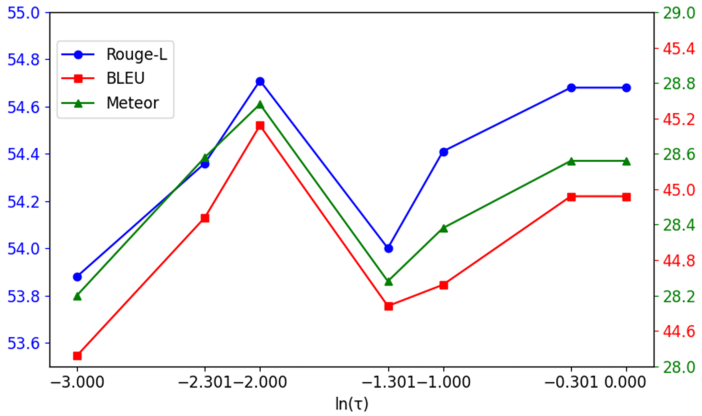

Figure 7. The influence of temperature parameter parameter τ.

Figure 7 shows the experimental results of the influence of the temperature parameter τ on the model performance in CogCol. In the contrastive loss, the temperature parameter τ amplifies the similarity comparison proportionally. The magnitude of similarity between positive and negative examples serves as the exponent of the natural logarithm e in the calculation of contrastive loss. Therefore, during contrastive learning, the model prioritizes learning the differences between negative examples. As shown in Figure 7, if the temperature parameter gradually decreases, the performance of the model slightly decreases. However, when the temperature parameter τ decreases to 0.01, the model's

performance increases, and subsequently, as the temperature parameter decreases further, the model's performance does not improve. Although smaller temperature parameters can increase the distance between anchor points and negative samples to improve the quality of representation, setting the temperature parameter too small may cause the model to overly focus on negative samples [45], resulting in poorer generalization ability. Therefore, in this paper, we set the temperature parameter to 0.01.

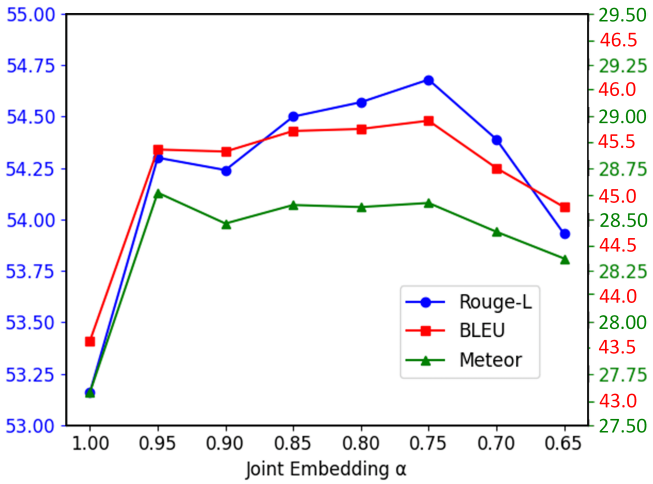

Figure 8. The influence of joint learning parameter α.

Figure 8 shows the influence of joint training parameters of contrastive learning and decoding loss on the model in CogCol. In model training, it is necessary to balance the weights of different modules to achieve the best fit for the current task. For the experiment on balancing joint training parameters, specific values of α (between 0 and 1 with an interval of 0.05) should be first selected, and we chose the range between 0.65 and 1 as the reference range. As shown in Figure 8, when α is set to 0.75, CogCol can achieve the best balance between contrastive learning loss and decoding loss, resulting in the best code summarization effect.

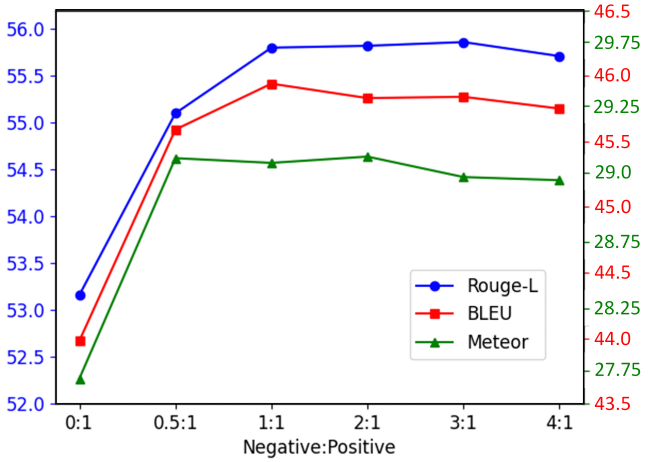

Figure 9. The influence of the ratio of negative samples to positive samples in contrastive learning.

Figure 9 shows the influence of the selection of positive and negative samples in contrastive learning for CogCol. In this paper, for the collection of negative samples in contrastive learning, sequences outside the batch (a batch contains 32 samples) of E-PDG or their corresponding contrastive learning positive samples were selected. Increasing the number of negative samples can increase the distance of unrelated representations in high-dimensional space. However, too many negative samples will cause the model to

focus more on the training related to negative samples and ignore the relationship between positive samples and anchor points. As shown in Figure 9, when no negative samples were selected, the model's performance was poor. As the proportion of negative samples increased, all indicators reached higher points when balanced with the number of positive samples, and with further increase in the proportion of negative samples, three indicators showed slight increases and decreases. Since a too large proportion of negative samples will increase the burden of model training, as well as the risk of overfitting and excessive emphasis on negative sample learning. So, in this paper, we adopt a balanced selection of positive and negative samples in a 1:1 ratio.

As shown in Figure 10, CogCol employed an early stopping strategy, thus stopping training when the epoch reached 135. For selecting the best model, to ensure fairness in the experiments, we followed AST-trans and chose the checkpoint with the best BLEU performance as the final model (in epoch 101), then evaluated it on the test set to obtain the results shown in Table 4 of RQ1.

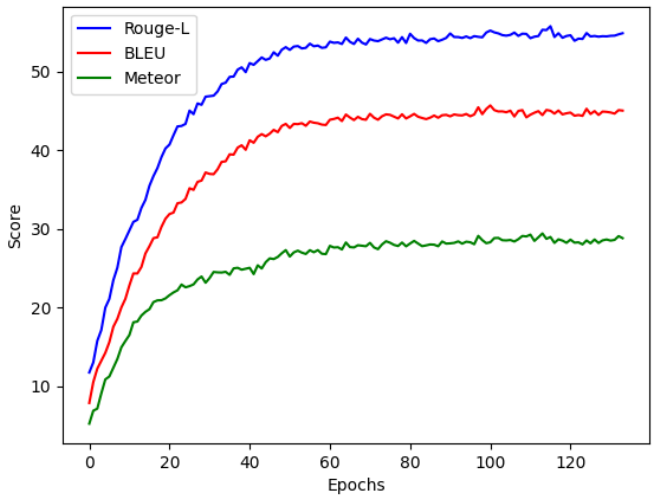

Figure 10. The evaluation of three metrics under different training epochs.

In this paper, we conducted parameter analysis to select reasonable parameter configurations to ensure that the model gradually converges to a better performance. Moreover, by employing appropriate parameter configurations, the model can achieve better results while ensuring both its generalization capability and training efficiency.

5. Conclusions

Natural language code summarization plays a crucial role in aiding software developers to swiftly and accurately comprehend existing code, thereby enhancing software development efficiency. In this paper, we introduce CogCol, a code summarization model based on contrastive learning utilizing code graphs, addressing the limitations of current code summarization models in leveraging structural information and understanding codes with similar structures. CogCol proposes a PDG enhancement method to derive an E-PDG incorporating multiple graph features. Subsequently, CogCol develops four operations on the E-PDG to construct positive samples for contrastive learning. By leveraging this proposed contrastive learning approach, CogCol improves the comprehension of code structural information for code summarization. Additionally, by employing a contrastive learning model based on the Transformer, CogCol effectively captures code graph information and achieves significant advancements in code summarization. Experimental results on large-scale open-source datasets demonstrate that CogCol outperforms existing state-of-the-art code summarization models in terms of BLEU, Rouge-L, and Meteor.

Author Contributions: Methodology, Y.S.; software, Y.S. and M.Y.; validation, Y.S., Y.Y. and L.C.; data curation, Y.S.; writing—original draft preparation, Y.S.; writing—review and editing, Y.Y., M.Y. and L.C.; supervision, Y.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in the article.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CogCol	Code graph-based Contrastive learning model
AST	Abstract Syntax Tree
CFG	Control Flow Graph
PDG	Program Dependency Graph
MST	Minimum Spanning Tree

References

1. Shi, C.; Cai, B.; Zhao, Y.; Gao, L.; Sood, K.; Xiang, Y. CoSS: Leveraging statement semantics for code summarization. *IEEE Trans. Softw. Eng.* **2023**, *49*, 3472–3486. [CrossRef]
2. Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; Jin, Z. Summarizing source code with transferred api knowledge. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelli-Gence (IJCAI 2018), Stockholm, Sweden, 13–19 July 2018; pp. 2269–2275.
3. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 28–29 May 2018; pp. 200–210.
4. Ferretti, C.; Saletta, M. Naturalness in Source Code Summarization. How Significant is it? In Proceedings of the IEEE/ACM 31st International Conference on Program Comprehension, Melbourne, VIC, Australia, 15–16 May 2023; pp. 125–134.
5. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics, Online, 16–20 November 2020; pp. 1536–1547.
6. Ying, A.T.T.; Robillard, M.P. Code fragment summarization. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 655–658.
7. Ying, A.T.T.; Robillard, M.P. Selection and presentation practices for code example summarization. In Proceedings of the 22nd ACM Sigsoft International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 460–471.
8. Rodeghero, P.; McMillan, C.; McBurney, P.W.; Bosch, N.; D'Mello, S. Improving automated source code summarization via an eye-tracking study of programmers. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 390–401.
9. McBurney, P.W.; McMillan, C. Automatic source code summarization of context for java methods. *IEEE Trans. Softw. Eng.* **2015**, *42*, 103–119. [CrossRef]
10. Haiduc, S.; Aponte, J.; Moreno, L.; Marcus, A. On the use of automated text summarization techniques for summarizing source code. In Proceedings of the 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 35–44.
11. Eddy, B.P.; Robinson, J.A.; Kraft, N.A.; Carver, J.C. Evaluating source code summarization techniques: Replication and expansion. In Proceedings of the 21st International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013; pp. 13–22.
12. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944.
13. Tang, Z.; Shen, X.; Li, C.; Ge, J.; Huang, L.; Zhu, Z.; Luo, B. AST-trans: Code summarization with efficient tree-structured attention. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 150–162.
14. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Volume 30.
15. Shi, Y.; Yin, Y.; Wang, Z.; Lo, D.; Zhang, T.; Xia, X.; Zhao, Y.; Xu, B. 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, Singapore, 14–18 November 2022; pp. 722–733.

16. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 2073–2083.
17. Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; Yu, P.S. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 397–407.
18. Tang, Z.; Li, C.; Ge, J.; Shen, X.; Zhu, Z.; Luo, B. AST-transformer: Encoding abstract syntax trees efficiently for code summarization. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, VIC, Australia, 15–19 November 2021; pp. 1193–1195.
19. Cai, R.; Liang, Z.; Xu, B.; Li, Z.; Hao, Y.; Chen, Y. TAG: Type auxiliary guiding for code comment generation. In Proceedings of the Findings of the Association for Computational Linguistics, Online, 16–20 November 2020; pp. 291–301.
20. Choi, Y.; Bak, J.; Na, C.; Lee, J.H. Learning sequential and structural information for source code summarization. In Proceedings of the Findings of the Association for Computational Linguistics, Online, 1–6 August 2021; pp. 2842–2851.
21. Wu, H.; Zhao, H.; Zhang, M. Code summarization with structure-induced transformer. In Proceedings of the Findings of the Association for Computational Linguistics, Online, 1–6 August 2021; pp. 1078–1090.
22. Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; Jin, Z. Building program vector representations for deep learning. In Proceedings of the International Conference on Knowledge Science, Engineering and Management, Chongqing, China, 28–30 October 2015; pp. 547–553.
23. Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; Guibas, L. Learning program embeddings to propagate feedback on student code. In Proceedings of the International Conference on Machine Learning, Lille, France, 6–11 July 2015; pp. 1093–1102.
24. Dam, H.K.; Tran, T.; Pham, T. A deep language model for software code. *arXiv* **2016**, arXiv:1608.02715.
25. Xin, X.; Lo, D. An effective change recommendation approach for supplementary bug fixes. *Autom. Softw. Eng.* **2017**, *24*, 455–498.
26. Lam, A.N.; Nguyen, A.T.; Nguyen, H.A. Combining deep learning with information retrieval to localize buggy files for bug reports. In Proceedings of the International Conference on Automated Software Engineering, Lincoln, NE, USA, 9–13 November 2015; pp. 476–481.
27. White, M.; Tufano, M.; Vendome, C. Deep learning code fragments for code clone detection. In Proceedings of the International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 87–98.
28. Alon, U.; Zilberstein, M.; Levy, O. code2vec: Learning distributed representations of code. In Proceedings of the ACM on Programming Languages, Phoenix, AZ, USA, 22–26 June 2019; Volume 3, pp. 1–29.
29. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Association for the Advancement of Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; pp. 1287–1293.
30. Wan, Y.; Shu, J.; Sui, Y.; Xu, G.; Zhao, Z.; Wu, J.; Yu, P. Multi-modal attention network learning for semantic source code retrieval. In Proceedings of the International Conference on Automated Software Engineering, San Diego, CA, USA, 11–15 November 2019; pp. 13–25.
31. Allamanis, M.; Tarlow, D.; Gordon, A.; Wei, Y. Bimodal modeling of source code and natural language. In Proceedings of the International Conference on Machine Learning, Lille, France, 6–11 July 2015.
32. Liu, J.; Zeng, J.; Wang, X.; Liang, Z. Learning graph-based code representations for source-level functional similarity detection. In Proceedings of the 45th International Conference on Software Engineering, Melbourne, VIC, Australia, 14–20 May 2023; pp. 345–357.
33. Yadavally, A.; Nguyen, T.N.; Wang, W.; Wang, S. (Partial) Program Dependence Learning. In Proceedings of the 45th International Conference on Software Engineering, Melbourne, VIC, Australia, 14–20 May 2023; pp. 2501–2513.
34. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the 9th International Conference on Learning Representations, Virtual, 3–7 May 2021.
35. Ferrante, J.; Ottenstein, K.J.; Warren, J.D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* **1987**, *9*, 319–349. [CrossRef]
36. Gad, W.; Alokla, A.; Nazih, W.; Aref, M.; Salem, A.B. DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code. *Comput. Mater. Contin.* **2022**, *70*, 3117–3132. [CrossRef]
37. Jaiswal, A.; Babu, A.R.; Zadeh, M.Z.; Banerjee, D.; Makedon, F. A survey on contrastive self-supervised learning. *Technologies* **2020**, *9*, 2. [CrossRef]
38. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. Bleu: A method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 6–12 July 2002; pp. 311–318.
39. Lin, C.Y. Rouge: A package for automatic evaluation of summaries. In Proceedings of the Text Summarization Branches out, Barcelona, Spain, 25–26 July 2004; pp. 74–81.
40. Banerjee, S.; Lavie, A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, Ann Arbor, MI, USA, 29 June 2005; pp. 65–72.

41. Eriguchi, A.; Hashimoto, K.; Tsuruoka, Y. Tree-to-sequence attentional neural machine translation. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 823–833.
42. Nie, P.; Zhang, J.; Li, J.J.; Mooney, R.J.; Gligoric, M. Impact of evaluation methodologies on code summarization. In Proceedings of the 40th International Conference on Software Engineering, Olympic Valley, CA, USA, 23–26 October 2022; pp. 4936–4960.
43. Shi, E.; Wang, Y.; Du, L.; Chen, J.; Han, S.; Zhang, H.; Zhang, D.; Sun, H. On the evaluation of neural code summarization. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1597–1608.
44. Haque, S.; Eberhart, Z.; Bansal, A.; McMillan, C. Semantic similarity metrics for evaluating source code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 36–47.
45. Wang, F.; Liu, H. Understanding the Behaviour of Contrastive Loss. In Proceedings of the Computer Vision and Pattern Recognition Conference, Nashville, TN, USA, 20–25 June 2021; pp. 2495–2504.