CodeGRAG: Extracting Composed Syntax Graphs for Retrieval Augmented Cross-Lingual Code Generation

Anonymous ACL submission

Abstract

Utilizing large language models to generate codes has shown promising meaning in software development revolution. Despite the intelligence shown by the general large language models, their specificity in code generation can still be improved due to the syntactic gap and mismatched vocabulary existing among natural language and different programming languages. In addition, programming languages are inherently logical and complex, making them hard to be correctly generated. Existing methods rely on multiple prompts to the large language model to explore better solutions, which is expensive. In this paper, we propose Syntax Graph Retrieval Augmented Code Generation (CodeGRAG) to enhance the performance of LLMs in single-round code generation tasks. CodeGRAG extracts and summarizes the control flow and data flow of code blocks to fill the gap between programming languages and natural language. The extracted external structural knowledge models the inherent flows of code blocks, which can facilitate LLMs for better understanding of code syntax and serve as a bridge among different programming languages. CodeGRAG significantly improves the code generation ability of LLMs and can even offer performance gain for cross-lingual code generation, e.g., C++ for Python.

1 Introduction

In recent years, large language models (LLMs) (Achiam et al., 2023; Touvron et al., 2023a) have shown great impact in various domains. Automated code generation emerges as a captivating frontier (Zheng et al., 2023; Roziere et al., 2023; Shen et al., 2023), promising to revolutionize software development by enabling machines to write and optimize code with minimal human intervention.

Despite the success in natural language understanding and generation, there is still a gap between the natural language and programming language

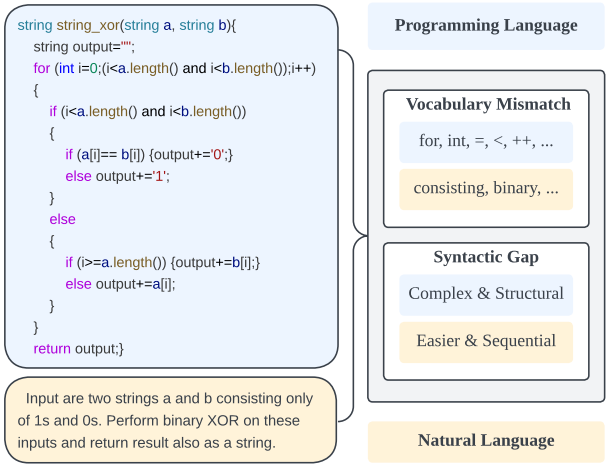

Figure 1: Illustration of the gap between the programming language and the natural language.

as illustrated in Figure 1. Code, unlike natural language, possesses a well-defined structure that governs its syntax and semantics. This structure provides valuable information about the relationships between different parts of the code, the flow of execution, and the overall organization of the functions (Jiang et al., 2021; Guo et al., 2020). General-purpose LLMs regard a code block as a sequence of tokens. By ignoring the inherent structure of codes, they miss out on essential cues that could help them better understand and generate code. In addition, the multi-lingual code generation abilities of LLMs remain a challenge due to the gap between different programming languages. Moreover, the training data of different programming languages can be unbalanced, which leads to a biased ability of multi-lingual code generation.

Previous methods on code representation learning seek to interpret code blocks based on the abstract syntax tree, data flow graph (Aho et al., 2006), control flow graph (Allen, 1970), and the intermediate states (Lattner and Adve, 2004), where the semantic level and logical level information of codes are better captured. These methods gain

the performance improvement in code classification (Long et al., 2022), code search (Zeng et al., 2023), etc, showcasing the benefits of capturing the structures of codes. In this paper, we aim to inject the structural knowledge of codes into the general-purpose large language models for a better understanding of codes. By understanding the underlying structure of code, the LLMs can 1) extract the inherent semantic and logic flow of codes for higher quality generation and 2) transfer knowledge across different programming languages, which enables cross-lingual code generation.

However, tuning the base model for structural code understanding is expensive and heavy. Recently, retrieval augmented generation (RAG) has shown great performance gains in facilitating LLMs on complex and temporal-related tasks, well utilizing the in-context-learning ability to instruct LLMs for enhanced generation. In this paper, we propose to develop a code RAG framework to retrieve relevant structural code representations to fill the gap between natural language and programming language, provide hints for enhanced code generation, and utilize the structure information of codes to offer cross-lingual code generation augmentation.

The main challenges of designing an effective framework of RAG for code generation can be summarized in threefolds. 1) How to design effective structural views of code blocks to represent the inherent syntax of codes, which could fill the gap between natural language and programming languages, inform LLMs of external syntactic knowledge, and model the underlying cross-lingual knowledge. 2) How to measure the structural similarities between codes and how to align the natural language and programming language for joint similarity computation. 3) How to obtain an informative query format for efficient knowledge querying, which can offer consistent useful information extraction under different sets of problems that are diversified in syntax and styles.

Concretely, we propose to 1) represent the code blocks with the composed syntax graph based on the data-flow-graph and control-flow-graph, which extracts both the semantic level and the logical level information of code blocks, 2) use hybrid-GNN and pretrained cross-lingual code search models to measure the distances, and 3) obtain the problem description and the function declaration as the query content.

The main contributions of the paper can be summarized as follows:

- We propose an efficient and effective method named CodeGRAG that bridges the gap among natural language and programming languages, transfers knowledge among different programming languages, and utilizes the in-context learning ability of LLMs for enhanced code generation. The generation process requires only one calling of LLMs and can offer cross-lingual enhancement.

- We propose an effective graphical view to purify the semantic and logic knowledge from the code space, which offers more useful information than the raw code block and can summarize the cross-lingual knowledge.

- We propose an effective retrieval method to retrieve relevant code knowledge for generation enhancement, where the cross-modality alignment and structure preserving are achieved.

2 Related Work

LLMs for NL2Code. The evolution of the Natural Language to Code translation (NL2Code) task has been significantly influenced by the development of large language models (LLMs). Initially, general LLMs like GPT-J (Radford et al., 2023), GPT-NeoX (Black et al., 2022), and LLaMA (Touvron et al., 2023a), despite not being specifically tailored for code generation, showed notable NL2Code capabilities due to their training on datasets containing extensive code data like the Pile (Gao et al., 2020) and ROOTS (Laurençon et al., 2022). To further enhance these capabilities, additional pretraining specifically focused on code has been employed. PaLM-Coder, an adaptation of the PaLM model (Chowdhery et al., 2023), underwent further training on an extra 7.8 billion code tokens, significantly improving its performance in code-related tasks. Similarly, Code LLaMA (Roziere et al., 2023) represents an advancement of LLaMA2 (Touvron et al., 2023b), benefiting from extended training on over 500 billion code tokens, leading to marked improvements over previous models in both code generation and understanding. These developments underscore the potential of adapting generalist LLMs to specific domains like NL2Code through targeted training, leading to more effective and efficient code translation solutions.

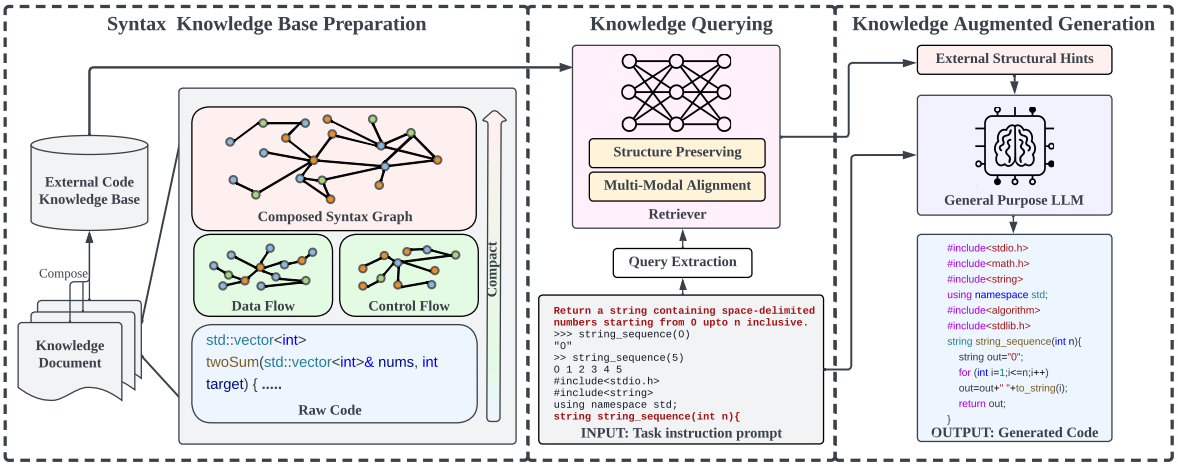

Figure 2: The illustration of the overall process of CodeGRAG.

Code Search. The code search methods can be summarized into three folds. Early methods utilizes sparse search to match the query and codes (Hill et al., 2011; Yang and Huang, 2017), which suffers from mismatched vocabulary due to the gap between natural language and codes. Neural methods (Cambronero et al., 2019; Gu et al., 2021) then focus on mapping the query and codes into a joint representation space for more accurate retrieval. With the success of pretrained language models, many methods propose to use pretraining tasks to improve the code understanding abilities and align different language spaces. For example, CodeBERT (Feng et al., 2020) is pretrained on NL-PL pairs of 6 programming languages with the masked language modeling and replaced token detection task. CodeT5 (Wang et al., 2021) supports both code-related understanding and generation tasks through bimodal dual generation. UniXcoder (Guo et al., 2022) integrates the aforementioned pretraining tasks, which is a unified cross-modal pre-trained model.

Code Representation. Early methods regard code snippets as sequences of tokens, assuming the adjacent tokens will have strong correlations. This line of methods (Harer et al., 2018; Ben-Nun et al., 2018; Feng et al., 2020; Ciniselli et al., 2021) take programming languages as the same with the natural language, using language models to encode the code snippets too. However, this ignoring of the inherent structure of codes leads to a loss of expressiveness. Methods that take the structural information of codes into consideration then emerge. Mou et al. (2016) used convolution networks over the abstract syntax tree (AST) extracted from codes. Alon et al. (2019) encoded paths sampled from the AST to represent codes. Further exploration into the graphical representation of codes (Allamanis et al., 2017) is conducted to better encode the structures of codes, where more intermediate states of the codes are considered.

3 Methodology

3.1 Overview

In this paper, we leverage both generative models and retrieval models to produce results that are both coherent and informed by the external syntax knowledge. The overall process of CodeGRAG is illustrated in Figure 2, which consists of three stages as follows.

Syntax Knowledge Base Preparation. We analyze the syntax and control information of code blocks and extract their graphical views to better represent the codes. This process can be formulated as, $\forall c_i \in D_{\text{pool}}$:

$$g_i \leftarrow \text{GraphExtractor}(c_i), \quad (1)$$

$$\text{KB.append}(\langle c_i, g_i \rangle), \quad (2)$$

where c_i is the raw code block and g_i is the extracted graphical view of the corresponding code block.

Knowledge Querying. Given a target problem to be completed, we generate informative query of it and use it to retrieve knowledge from the constructed knowledge base. The process can be formulated as:

$$q \leftarrow \text{QueryExtractor}(p), \quad (3)$$

$$i \xleftarrow{\text{Top-1}} \text{Retriever}(q, KB), \quad (4)$$

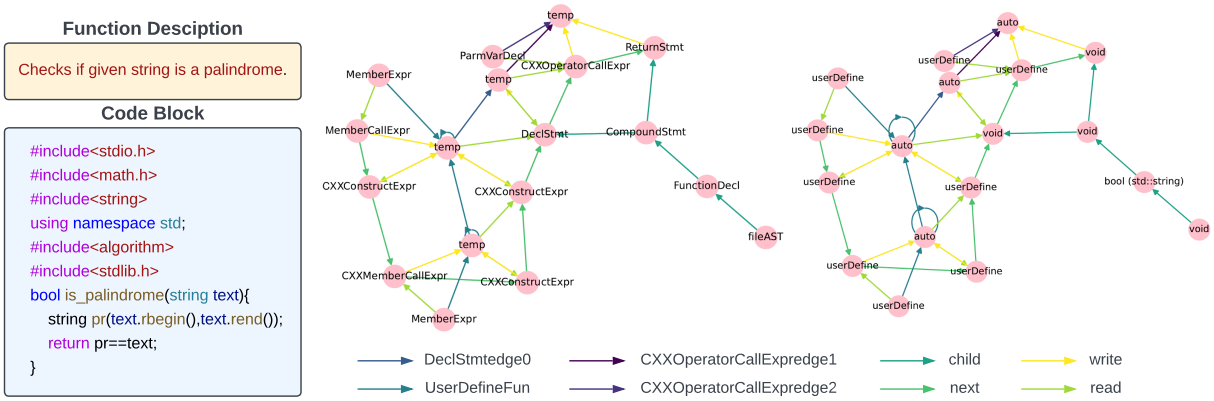

Figure 3: Illustration of the extracted composed syntax graph from the code block. The arrows in the bottom part indicate the names of different edges, which are extracted based on the ASTs.

where q denotes the query content, p denotes the target problem, and i is the returned index of the Top-1 relevant content stored in the constructed knowledge base.

Knowledge Augmented Generation. After obtaining the knowledge index, we access the corresponding entry of the knowledge base and obtain the graphical syntax knowledge to inform the LLMs for enhanced code generation. The process can be formulated as:

$$c_i, g_i \leftarrow \text{KB}[i], \tag{5}$$

$$\text{result} \leftarrow \text{LLM}(p, g_i). \tag{6}$$

3.2 Syntax Graph Knowledge Base

In this section, we discuss how to extract informative views for code blocks and how to summarize the retrieved knowledge. An illustration is given in Figure 3. To capture both the semantic and the logical information, we propose to combine the data flow graph (Aho et al., 2006) and the control flow graph (Allen, 1970) with the read-write signals (Long et al., 2022) to represent the code blocks, both of them are constructed on the base of the abstract syntax tree.

Abstract Syntax Tree (AST). An abstract syntax tree (AST) is a tree data structure that represents the abstract syntactic structure of source code. An AST is constructed by a parser, which reads the source code and creates a tree of nodes. Each node in the tree represents a syntactic construct in the source code, such as a statement, an expression, or a declaration. ASTs are used in a variety of compiler construction and program analysis tasks, including: parsing, type checking, optimization, and program analysis. ASTs have good compactness and can represent the structure of the source code in a clear and concise way.

Data Flow Graph (DFG). The data flow graph (DFG) is a graphical representation of the flow of data dependencies within a program. It is a directed graph that models how data is transformed and propagated through different parts of a program. In DFG, nodes are operands and edges indicate data flows. Two types of edges are considered: 1) operation edges that connect the nodes to be operated and the nodes that receive the operation results; 2) function edges that indicate data flows for function calls and returns. These edges connect nodes, including non-temporary operands and temporary operands, which refer to variables and constants that explicitly exist in the source code, and variables existing only in execution, respectively.

Control Flow Graph (CFG). The control flow graph (CFG) is a graphical representation of the flow of control or the sequence of execution within a program. It is a directed graph that models the control relationships between different parts of a program. Based on compiler principles, we slightly adjust the design of CFG to better capture the key information of the program. Nodes in CFG are operations in the source code, including standard operations, function calls and returns. Edges indicate the execution order of operations.

Composed Syntax Graph. A composed syntax graph composes the data flow graph and the control flow graph with the read-write flow existing in the code blocks. An illustration of the extracted composed syntax graph is displayed in Figure 3. Different edge types along with their concrete names are given in colors. As for the node names, the middle figure displays the concrete types of nodes (operands) and the right figure displays the proper-

ties of nodes.

After obtaining the composed syntax graphs for code blocks in the retrieval pool, we use them for retrieval enhancement, where both the semantic level and the logic level information are preserved.

3.3 Knowledge Querying

The main problems of the retrieval lie in: 1) how to fill the gap between natural languages and programming language and the gap among different programming languages; 2) how to properly measure the similarity of code blocks, e.g., the unique structure existing in programming languages; 3) how to design the informative query content.

3.3.1 Query Content

Since the styles of different code problems can diversify, the query content of the retrieval process matters. We consider the following contents: 1) Problem description, which describes the task to be completed by the target function code. Potential ambiguity and style diversity may exist among different problems set, which lead to a decrease in retrieval accuracy. 2) Function declaration, which gives the function name and the input variables.

Before knowledge querying, we first extract the problem description of each task to reduce the ambiguity and then concatenate it with the function declaration to serve as the query content, where the functionality and input format of the expected code block are contained.

3.3.2 Retrieval Function

The query of the retrieval includes problem description Q_p and function description Q_c, while each content of the retrieval pool includes raw code block V_c and its graphical view V_g.

To expressively represent each of the components, we use the pretrained NL2Code model encoder $\phi_1(\cdot)$ to represent the problem description and code snippets and design a graph neural network $\phi_2(\cdot)$ to represent the graphical view. The retrieval function can be formulated as:

$$\mathbf{h}^{\mathbf{V}} = \psi(\phi_1(V_c), \phi_2(V_g)), \tag{7}$$

$$\mathbf{h}^{\mathbf{Q}} = \phi_1(Q_p \| Q_c), \tag{8}$$

$$\text{Distance} = 1 - \frac{\mathbf{h}^{\mathbf{Q}} \cdot \mathbf{h}^{\mathbf{V}}}{\|\mathbf{h}^{\mathbf{Q}}\| \cdot \|\mathbf{h}^{\mathbf{V}}\|}. \tag{9}$$

We use models pretrained to align natural language and programming language to serve as the encoder ϕ_1:

- **CodeT5+** (Wang et al., 2023) is an advanced encoder-decoder foundation LLM, tailored for diverse code understanding and generation tasks. It is designed for various code-related tasks, capable of operating in different modes (encoder-only, decoder-only, encoder-decoder). It employs a range of pretraining tasks, including span denoising, causal language modeling on code, and text-code contrastive learning, matching and CLM tasks. This approach improves its performance in retrieval tasks by effectively aligning text and code data.

- **UnixCoder** (Guo et al., 2022) is a cross-modal pre-trained model optimized for both code understanding and generation tasks. Leveraging a multi-layer Transformer structure with mask attention matrices, it outperforms traditional unified encoder-decoder models in autoregressive tasks like code completion. UniXcoder uniquely employs multi-modal inputs, including code comments and abstract syntax trees (AST), enhancing its code representation capabilities. It employs three types of language modeling tasks: masked language modeling, unidirectional language modeling, and denoising objectives. Additionally, it introduces two advanced pre-training tasks: multi-modal contrastive learning and cross-modal generation, which refine the semantic representation of code fragments.

And we design a graph neural network ϕ_2 to preserve the semantic and logical information of code blocks. The representation of each node $\mathbf{n}_i^{(0)}$ and edge $\mathbf{e}_i^{(0)}$ are first initialized with vectors corresponding to the node text and edge text encoded by ϕ_1. A message passing process is first conducted to fuse the semantic and structural information into each node representation.

$$\mathbf{m}_{ij}^{(l)} = \mathbf{W}^{(l)}(\mathbf{n}_j^{(l-1)} \| \mathbf{e}_j^{(l-1)}), \tag{10}$$

$$Q_j^{(l)} = \mathbf{W_Q}^{(l)} \mathbf{n}_j^{(l-1)}, \tag{11}$$

$$\mathbf{K}_{ij}^{(l)} = \mathbf{W_K}^{(l)} m_{ij}^{(l)}, \quad \mathbf{V}_{ij}^{(l)} = \mathbf{W_V}^{(l)} \mathbf{m}_{ij}^{(l)}, \tag{12}$$

$$a_{ij}^{(l)} = \text{softmax}_{i \in N(j)}(\mathbf{Q}_j^{(l)} \mathbf{K}_{ij}^{(l)}), \tag{13}$$

$$\mathbf{n}_j^{(l)} = \sum_{i \in N(j)} a_{ij}^{(l)} \mathbf{V}_{ij}^{(l)}. \tag{14}$$

A global attention-based readout is then applied to obtain the graph representation:

$$\mathbf{g} = \sum_i \text{softmax}(f_{\text{gate}}(\mathbf{n}_i^L)) f_{\text{feat}}(\mathbf{n}_i^L). \tag{15}$$

The different encoding networks are optimized via the contrastive learning based self-supervised training, which includes the intra-modality contrastive learning and inter-modality contrastive learning. The intra-modality constrastive learning serves for preserving the modality information, while the inter-modality contrastive learning serves for modality alignment.

Alignment Contrastive Learning. There are two types of alignment to be ensured during the retrieval process: 1) Question-Answer (QA) alignment and 2) Code-Graph (CG) alignment. We define the positive pairs for QA alignment purpose as $\mathcal{I}^+_{QA} = \{\langle \mathbf{h}^V_i, \mathbf{h}^Q_i \rangle | i \in D_{\text{train}}\}$ and define the negative pairs for QA alignment purpose as $\mathcal{I}^-_{QA} = \{\langle \mathbf{h}^V_i, \mathbf{h}^Q_j \rangle | i \neq j, i \in D_{\text{train}}, j \in D_{\text{train}}\}$.

And we define the positive pairs for CG alignment purpose as $\mathcal{I}^+_{CG} = \{\langle \phi_1(c_i), \phi_2(g_i) \rangle | i \in D_{\text{train}}\}$ and define the negative pairs for CG alignment purpose as $\mathcal{I}^-_{CG} = \{\langle \phi_1(c_i), \phi_2(g_j) \rangle | i \neq j, i \in D_{\text{train}}, j \in D_{\text{train}}\}$.

Structure Preserving Contrastive Learning. To preserve the structural information of the graphical views, we perform intra-modality contrastive learning among the graphical views and their corrupted views. Concretely, we corrupt each of the graphical view g_i with the edge dropping operation to obtain its corrupted view g'_i. The positive pairs for structure-preserving purpose are then designed as $\mathcal{I}^+_{\text{preserve}} = \{\langle \phi_2(g_i), \phi_2(g'_i) \rangle | i \in D_{\text{train}}\}$. The negative pairs for structure preserving purpose are designed as $\mathcal{I}^-_{\text{preserve}} = \{\langle \phi_2(g_i), \phi_2(g'_j) \rangle | i \neq j, i \in D_{\text{train}}, j \in D_{\text{train}}\}$.

3.4 Knowledge Augmented Generation

After we obtain the summarized retrieved knowledge, we append it to the prompt text for knowledge augmented generation. The prompt used is illustrated in Figure 4.

The prompt consists of three components: the system prompt illustrated in the blue part, the retrieved knowledge and hints illustrated in the green part, and the problem (including task description, function declaration, etc.) illustrated in the yellow part. The three parts are concatenated to be fed into LLMs for knowledge augmented generation.

4 Experiments

In this section, we conduct experiments to answer the following research questions (RQs).

Prompt for Knowledge Augmentated Generation

System Prompt

Please continue to complete the [*lang*] function according to the requirements and function declarations. You are not allowed to modify the given code and do the completion only.\n

Retrieved Knowledge

The syntax graph of a similar code might be:\n
[*composed syntax graph desciption*]
You can refer to the above knowledge to do the completion. \n

Problem

The problem:\n
[*problem prompt*]

Figure 4: The illustration of the used prompt.

RQ1 Does the proposed CodeGRAG offer performance gain against the base model?

RQ2 Does the proposed graph view abstract more informative knowledge compared with the raw code block?

RQ3 Does the proposed CodeGRAG model the high-level thought-of-codes? Can CodeGRAG offer cross-lingual augmentation?

RQ4 How does the retrieval algorithms impact the performance?

RQ5 What is the impact of each of the components of the graphical view?

RQ6 How is the compatibility of CodeGRAG?

4.1 Setup

In this paper, we evaluate CodeGRAG with the widely used HumanEval-X (Zheng et al., 2023) dataset, which is a multi-lingual code benchmark. We use greedy decoding strategy to do the generation. The evaluation metric is Pass@1.

We evaluate the multi-lingual code generation abilities of 1) models with less than 10 billion parameters: GPT-J (Radford et al., 2023), CodeGen-Multi (Nijkamp et al., 2022), and InCoder(Fried et al., 2022); 2) models with 10-20 billion parameters: CodeGeeX (Zheng et al., 2023), CodeGen-Multi (Nijkamp et al., 2022), CodeGen-Mono (Nijkamp et al., 2022), StarCoder (Li et al., 2023), WizardCoder (Luo et al., 2023), and Pangu-Coder2 (Shen et al., 2023); 3) close-sourced GPT-3.5

Table 1: Main results of Pass@1 on Humaneval-X.

Model Size	Model	Humaneval-X C++	Humaneval-X Python
6B	GPT-J	7.54	11.10
6B	CodeGen-Multi	11.44	19.41
6.7B	InCoder	9.50	16.41
13B	CodeGeeX	17.06	22.89
16B	CodeGen-Multi	18.05	19.22
16B	CodeGen-Mono	19.51	29.28
15B	StarCoder	31.55	32.93
15B	WizardCoder	29.27	57.3
15B	Pangu-Coder2	45.12	64.63
175B	GPT-3.5-Turbo	57.93	71.95
175B	GPT-3.5-Turbo + Code-RAG	60.37	72.56
175B	GPT-3.5-Turbo + CodeGRAG	62.20	72.56
175B	GPT-3.5-Turbo + Cross-Lingual Code-RAG	62.20	70.12
175B	GPT-3.5-Turbo + Cross-Lingual CodeGRAG	**64.02**	**77.44**

model. The detailed statistics of the model sizes are displayed in Table 1.

4.2 Main Results

The main results are summarized in Table 1. Code-RAG represents directly using the raw code block as the retrieved knowledge and CodeGRAG refers to using the graphical view as the knowledge. From the results, we can draw the following conclusions:

RQ1. The proposed CodeGRAG could offer performance gain against the base model, which validates the effectiveness of the proposed RAG for code generation framework.

RQ2. The model with CodeGRAG could beat model with raw code block augmented generation. From the results, we can see that the proposed graph view could summarize the useful structural syntax information and filter out the noises, which could offer more informative knowledge hints than the raw code block.

RQ3. Python code generation is improved with cross-lingual retrieval, which uses C++ code and corresponding graphical views as the retrieved knowledge for an enhanced generation. Therefore, the proposed CodeGRAG could offer cross-lingual augmentation. This validates that the proposed graphical view could abstract the high-level and inherent information (e.g., the control and data flow to solve a specific problem) of the code blocks, which are shared across different programming languages regarding solving the same problem.

4.3 Impact of Retrieval Methods (RQ4)

In this section, we explore how different retrieval methods influence the final performance. We use CodeT5 and UnixCoder as the text and code encoder, which are both trained with multi-lingual data and dedicated for multi-lingual search. The results are summarized in Table 2.

From the results, we could see that the augmentation performance can differ with different retrieval content encoders. Specifically, the cross-lingual augmentation performance increases with UnixCoder being the encoder compared with that of CodeT5 being encoder. Since UnixCoder involves in the abstract syntax tree for structure capturing, while CodeT5 does not. This validates the effectiveness of involving structure information. In addition, we can see that with better cross-lingual retrieval accuracy, directly using raw codes as cross-lingual hints will bias the model generation, while feeding graphical views of syntax will boost the model performance.

4.4 Impacts of the Components of the Graphical View (RQ5)

In this section, we adjust the inputs of the graph components to the LLMs as the external knowledge. Concretely, we study the information contained in node names, edge names, and the topological structure. The results are presented in Table 3.

The edge type refers to the type of flows between operands (child, read, write, etc.), the node type refers to the type of operands (DeclStmt, temp, etc.), the node name refers to the name of the intermediate variables, and the topological information refers to the statistics of the concrete numbers of different types of edges. From the results, we can observe that 1) the edge features matter the most in constructing the structural view of code blocks

Table 2: Results with different content encoders (yielding different retrieval models).

Content Encoder $\phi_1(\cdot)$	Method	Humaneval-X C++	Humaneval-X Python
NA	GPT-3.5-Turbo	57.93	71.95
CodeT5	+ Code-RAG	60.37	72.56
	+ CodeGRAG	62.20	72.56
	+ Cross-Lingual Code-RAG	61.59	70.12
	+ Cross-Lingual CodeGRAG	62.20	76.83
UnixCoder	+ Code-RAG	58.54	71.95
	+ CodeGRAG	62.20	71.95
	+ Cross-Lingual Code-RAG	62.20	66.46
	+ Cross-Lingual CodeGRAG	64.02	77.44

Table 3: The impacts of the graph components.

Datasets	Python	C++
Edge Type Only	73.78	61.59
Edge Type + Node Name	75.00	59.76
Edge Type + Node Type	75.61	59.15
Edge Type + Topological	77.44	64.02

for enhancement, 2) the type of nodes expresses the most in representing operands information, and 3) the overall structure of the graphical view also gives additional information.

4.5 Discussion of the Compatibility (RQ6)

Despite the effectiveness of the proposed graphical views to represent the code blocks, the flexibility and convenience of applying the graphical views extraction process is important for wider application of the proposed method. In this section, we discuss the compatibility of CodeGRAG.

First of all, the extraction process of all the graphical views are front-end. Therefore, this extraction process applies to a wide range of code, even error code. One could also use convenient tools to reformulate the code and improve the pass rate of the extraction process.

In addition, we give the ratio of generated results that can pass the graphical views extraction process, which is denoted by Extraction Rate. The Pass@1 and the Extraction Rate of the generated results passing the graphical extraction process are given in Table 4 and Table 5.

Table 4: The extraction rate of the generated results passing the graphical extraction process (C++).

Generated Codes	Pass@1	Extraction Rate
Code-RAG	62.20	92.07
CodeGRAG	64.02	92.68

From the results, we could see that the extraction rates are high for codes to pass the graphical views extraction process, even under the situation where the Pass@1 ratios of the generated results are low. This indicates that the application range of the proposed method is wide. In addition, as the code RAG also offers performance gains, one could use multiple views, including raw code blocks and graphical representations, as the retrieval knowledge base.

Table 5: The extraction rate of the generated results passing the graphical extraction process (Python).

Generated Codes	Pass@1	Extraction Rate
Code-RAG	71.95	91.46
CodeGRAG	77.44	96.95

5 Conclusion

Despite the expanding role of LLMs in code generation, there are inherent challenges pertaining to their understanding of code syntax and their multi-lingual code generation capabilities. This paper introduces the Syntax Graph Retrieval Augmented Code Generation (CodeGRAG) to enhance LLMs for single round and cross-lingual code generation. CodeGRAG extracts and summarizes data flow and control flow information from codes, effectively bridging the gap between programming language and natural language. By integrating external structural knowledge, CodeGRAG enhances LLMs' comprehension of code syntax and empowers them to generate complex and multi-lingual code with improved accuracy and fluency. The method's demonstrated performance gains signal its potential to significantly elevate LLMs' code generation abilities, including facilitating cross-lingual programming language generation such as C++ to Python. Overall, the findings warrant optimism regarding the transformative impact of CodeGRAG in advancing LLMs' proficiency in code generation across various programming languages.

Limitations

In this paper, we propose a tuning-free method that can offer enhanced single round and cross-lingual code generation. Despite the efficiency and effectiveness, there are also limitations within this work. Firstly, dependency on the quality of the external knowledge base could be a potential concern. The quality of the external knowledge base could be improved with regular expression extraction on the noisy texts and codes. Secondly, the potential of combining the RAG with finetuning for improved generation is not explored in this work. Finetuning the LLMs with the assistance of CodeGRAG could be a future exploration.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2006. Compilers: Principles techniques and tools. 2007. *Google Scholar Google Scholar Digital Library Digital Library*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19.

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):1–29.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of bert models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

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)*, pages 483–494. IEEE.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.

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)*, pages 524–527. IEEE.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. 2022. The bigscience roots corpus: A 1.6 tb composite multilingual dataset. *Advances in Neural Information Processing Systems*, 35:31809–31826.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Ting Long, Yutong Xie, Xianyu Chen, Weinan Zhang, Qinxiang Cao, and Yong Yu. 2022. Multi-view graph representation for programming language processing: An investigation into algorithm detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 5792–5799.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

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*.

Yangrui Yang and Qing Huang. 2017. Iecs: Intent-enforced code search via extended boolean model. *Journal of Intelligent & Fuzzy Systems*, 33(4):2565–2576.

Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2023. degraphcs: Embedding variable-based flow graph for neural code search. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–27.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.