CodeGRAG: Bridging the Gap between Natural Language and Programming Language via Graphical Retrieval Augmented 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 this paper, we propose CodeGRAG, a Graphical Retrieval Augmented Code Generation framework to enhance the performance of LLMs. CodeGRAG builds the graphical view of code blocks based on the control flow and data flow of them to fill the gap between programming languages and natural language, which can facilitate natural language based LLMs for better understanding of code syntax and serve as a bridge among different programming languages. To take the extracted structural knowledge into the foundation models, we propose 1) a hard meta-graph prompt template to transform the challenging graphical representation into informative knowledge for tuning-free models and 2) a soft prompting technique that injects the domain knowledge of programming languages into the model parameters via finetuning the models with the help of a pretrained GNN expert model. CodeGRAG significantly improves the code generation ability of LLMs and can even offer performance gain for cross-lingual code generation.

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.

However, syntatic gap and mismatched vocabulary among between natural language and program-

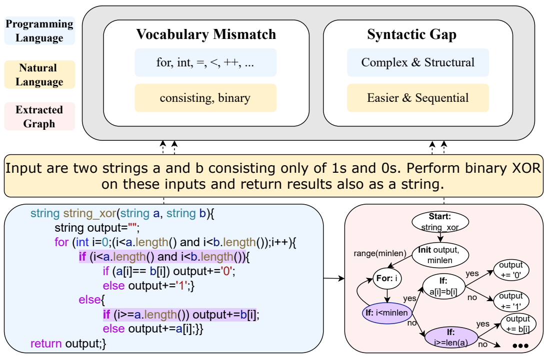


Figure 1: Illustration of the gap between the programming language and the natural language.

ming languages, hindering LLM's performance on code generation. As illustrated in Figure 1, programming language (marked in blue) contains special tokens such as "int" or "++" that natural language (marked in yellow) doesn't possess, leading to vocabulary mismatch. Besides, the relations between tokens in programming languages are often structural, e.g., the complex branching and jumps, whereas natural language is arranged simply in sequential manner, leading to syntactic gap. For example, in the control flow graph of the raw code (marked in pink), two "if" blocks (marked in purple) are adjacent and are executed sequentially under certain condition, but they appear to be intervaled in raw textual code.

As discussed above, the innate structures of programming languages are different from that of the sequential-based natural language. The challenges of enhancing a general-purposed large language models for code-related tasks can be summarized into two folds.

(C1) How to solve the gap between different languages and better interpret the inherent logic of code blocks. 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 is challenging due to the gap among different programming languages.

(C2) How to inject the innate knowledge of programming languages into general purpose large language models for enhancement. Despite the well representation of the programming knowledge, the ways to inject the knowledge into the NL-based foundation models is also challenging. The structural representation of code blocks could be hard to understand, which poses a challenge to the capability of the foundation models.

To solve the above challenges, we propose CodeGRAG, a graphical retrieval augmented generation framework for code generation. For **(C1)**, we propose to interpret the code blocks using the composed graph based on the data-flow and control-flow of the code block, which extracts both the semantic level and the logical level information of the code. The composed graphical view could 1) better capture the innate structural knowledge of codes for NL-based language models to understand and 2) model the innate function of code blocks that bridging different programming languages. For **(C2)**, we propose a meta-graph prompting technique for tuning-free models and a soft-prompting technique for tuned models. The meta-graph prompt summarizes the overall information of the extracted graphical view and transforms the challenging and noisy graphical representation into informative knowledge. The soft-prompting technique deals with the graphical view of codes with a pretrained GNN expert network and inject the processed knowledge embedding into the parameters of the general-purpose foundation models with the help of supervised finetuning.

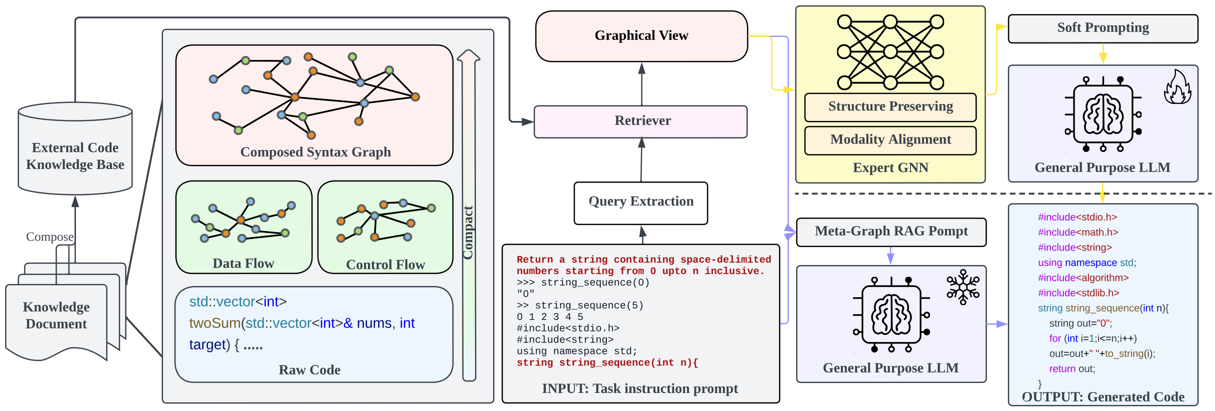
The main contributions of the paper can be summarized as follows:

- We propose CodeGRAG that bridges the gap among natural language and programming languages, transfers knowledge among different programming languages, and enhances the ability of LLMs for code generation. CodeGRAG requires only one calling of LLMs and can offer multi-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 soft prompting technique, which injects the domain knowledge of programming languages into the model parameters via finetuning LLMs with the assistance of a pretrained GNN expert model.

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.

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



(a) Graphical Knowledge Base Preparation (b) Knowledge Querying (c) Graphical Knowledge Augmented Generation (Top: Finetuning with soft prompting Bottem: Hard prompt with Meta-Graph)

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

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 expert graphical knowledge of programming language. The overall process of CodeGRAG is illustrated in Figure 2, which mainly consists of three stages: graphical knowledge base preparation, knowledge querying, and graphical knowledge augmented generation.

3.2 Graphical Knowledge Base Preparation

In this section, we discuss how to extract informative graphical views for code blocks. 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, $\forall c_i \in D_{\text{pool}}$:

$$g_i \leftarrow \text{GraphExtractor}(c_i), \tag{1}$$
$$\text{KB.append}(\langle c_i, g_i \rangle), \tag{2}$$

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

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

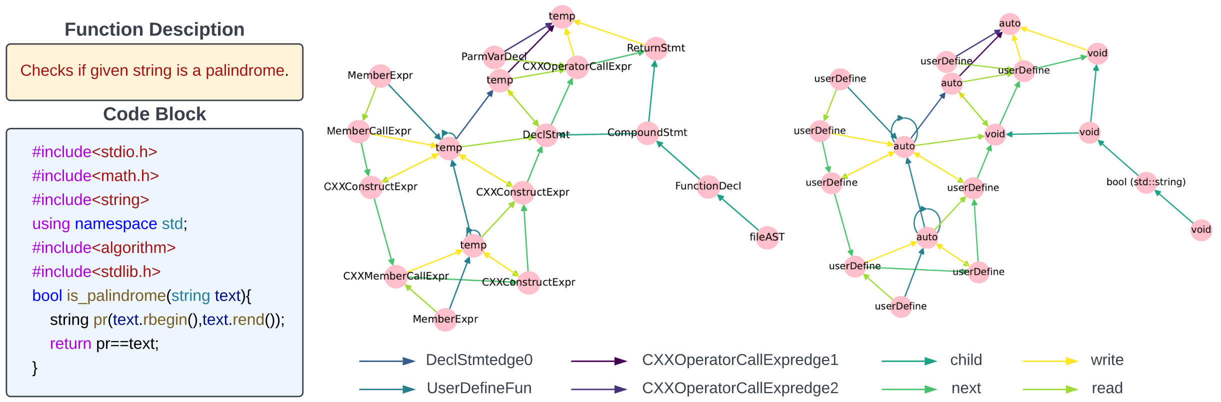


Figure 3: Illustration of the extracted composed syntax graph from the code block. The arrows in the bottom part indicate the names of different edges, which are extracted based on the ASTs.

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 properties of nodes.

An illustration of the composed graphical view is given in Figure 3. After obtaining the composed syntax graphs for code blocks in the retrieval pool, we use them to inform the general-purpose LLMs to bridge the gap between NL and PLs, where both the semantic level and the logic level information are preserved.

3.3 Knowledge Querying

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

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

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

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.

The main problems of the retrieval lie in: 1) how to design the informative query content and 2) how to align the different modalities.

3.3.1 Query Extractor

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 Retriever

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 the components, we use the encoder $\phi(\cdot)$ of the pretrained NL2Code model to represent the problem description and code snippets. The retrieval function is:

$$\mathbf{h}^{\mathbf{V}} = \phi(V_c \| V_g), \tag{5}$$

$$\mathbf{h}^{\mathbf{Q}} = \phi(Q_p \| Q_c), \tag{6}$$

$$\text{Distance} = 1 - \frac{\mathbf{h}^{\mathbf{Q}} \cdot \mathbf{h}^{\mathbf{V}}}{\|\mathbf{h}^{\mathbf{Q}}\| \cdot \|\mathbf{h}^{\mathbf{V}}\|}. \tag{7}$$

3.4 Graphical Knowledge Augmented Generation

After we obtain the returned graphical view, we inject it to the foundation LLMs for graphical knowledge augmented generation. Since the graphical view is hard to understand, we propose 1) a meta-graph template to transform the graphical view into informative knowledge for tuning-free model and 2) a soft prompting technique to tune the foundation models for their better understanding of the graphical views with the assistance of an expert GNN model.

3.4.1 Hard Meta-Graph Prompt

The original graphical view of a code block could contain hundreds of nodes and edges. A full description of it could cost a overly long context, along with the understanding challenge posed by the long edge lists. Therefore, we propose to use a meta-graph template to abstract the information of the graphical view, which describes the number of different nodes, that of different edges, and the overall topology.

The template for the meta-graph is displayed as below.

Graph(
num_nodes={node_type : *#nodes},*
num_edges={(src_node_type, edge_type, dst_node_type) : *#edges},*
metagraph=[(src_node_type, edge_type, dst_node_type)]

Then we use the meta-graph template to transform the retrieved graphical view into digestable knowledge and insert it into the final prompt for generation. As illustrated in Figure 4, the final 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.

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 description*]
You can refer to the above knowledge to do the completion. \n

Problem
The problem:\n
[*problem prompt*]

Figure 4: Illustration of the hard meta-graph prompt.

3.4.2 Soft Prompting with the Expert

Directly hard prompt to the LLMs poses a challenge to the digesting capability of the backbone LLMs, which could fail under the case where the backbone LLMs cannot well understand the graph components.

To compress the graphical knowledge into model parameters and help the backbone LLMs to better understand the programming language, we propose a soft prompting technique. The overall procedure can summarized into expert encoding of graphical views, finetuning with the expert signal, and inference.

Expert Encoding of Graphical Views. We design a graph neural network 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.

$$\mathbf{m}_{ij}^{(l)} = \mathbf{W}^{(l)}(\mathbf{n}_i^{(l-1)} \| \mathbf{e}_{ij}^{(l-1)}), \quad (8)$$

$$Q_j^{(l)} = \mathbf{W_Q}^{(l)} \mathbf{n}_j^{(l-1)}, \quad (9)$$

$$\mathbf{K}_{ij}^{(l)} = \mathbf{W_K}^{(l)} m_{ij}^{(l)}, \quad \mathbf{V}_{ij}^{(l)} = \mathbf{W_V}^{(l)} \mathbf{m}_{ij}^{(l)}, \quad (10)$$

$$a_{ij}^{(l)} = \text{softmax}_{i \in N(j)}(\mathbf{Q}_j^{(l)} \mathbf{K}_{ij}^{(l)}), \quad (11)$$

$$\mathbf{n}_j^{(l)} = \sum_{i \in N(j)} a_{ij}^{(l)} \mathbf{V}_{ij}^{(l)}. \quad (12)$$

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

$$\mathbf{g} = \sum_i \text{softmax}(f_{\text{gate}}(\mathbf{n}_i^L)) f_{\text{feat}}(\mathbf{n}_i^L). \quad (13)$$

The expert encoding network is 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: 1) NL-Code (NC) alignment and 2) Code-Graph (CG) alignment. We define the positive pairs for NC alignment purpose as $\mathcal{I}_{NC}^{+} = \{\langle \mathbf{h}_i^V, \mathbf{h}_i^Q \rangle | i \in D_{\text{train}}\}$ and define the negative pairs for NC alignment purpose as $\mathcal{I}_{NC}^{-} = \{\langle \mathbf{h}_i^V, \mathbf{h}_j^Q \rangle | i \neq j, i \in D_{\text{train}}, j \in D_{\text{train}}\}$.

  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 $\mathcal{I}_{\text{preserve}}^{-} = \{\langle \phi_2(g_i), \phi_2(g_j') \rangle | i \neq j, i \in D_{\text{train}}, j \in D_{\text{train}}\}$.

Finetuning with the Expert Soft Signal. To help the backbone LLMs to digest the graphical views, we tune the LLMs with the expert soft signal using supervised finetuning. The prompt for finetuning consists of the system prompt, retrieved knowledge where the expert encoded graphical view is contained using a token embedding, and task prompt, which is illustrated in Figure 5.

Soft Prompt for Knowledge Augmented Generation

System Prompt
Please use *[lang]* to write a correct solution to a programming problem. You should give executable completed code and nothing else.\n

Retrieved Knowledge
We also have the syntax graph embedding of a similar problem encoded in <GraphEmb> for you to refer to.\n

Problem
The problem:\n
[*problem prompt*]

Figure 5: Illustration of the soft prompting.

Inference. After the finetuning stage, we used the tuned models to generate codes using the soft prompting template as illustrated in Figure 5.

4 Experiments

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 Can soft prompting enhance the capability of the backbone LLMs? Does finetuning with the soft prompting outperforms the simple supervised finetuning?

RQ4 Does the proposed CodeGRAG model the high-level thought-of-codes? Can CodeGRAG offer cross-lingual augmentation?

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

RQ6 How is the compatibility of the graphical view?

Table 1: Results of code generation on Humaneval-X. (Pass@1)

Model Size	Model	Retrieved Knowledge	C++	Python
6B	GPT-J	N/A	7.54	11.10
6B	CodeGen-Multi	N/A	11.44	19.41
6.7B	InCoder	N/A	9.50	16.41
13B	CodeGeeX	N/A	17.06	22.89
16B	CodeGen-Multi	N/A	18.05	19.22
16B	CodeGen-Mono	N/A	19.51	29.28
15B	StarCoder	N/A	31.55	32.93
15B	WizardCoder	N/A	29.27	57.30
15B	Pangu-Coder2	N/A	45.12	64.63
-	GPT-3.5-Turbo	N/A	57.93	71.95
-	GPT-3.5-Turbo	Code Block	60.37	72.56
-	GPT-3.5-Turbo	Meta-Graph	**62.20**	**72.56**
-	GPT-3.5-Turbo	(Multi-Lingual) Code-Block	62.20	70.12
-	GPT-3.5-Turbo	(Multi-Lingual) Meta-Graph	**64.02**	**77.44**

Table 2: Results of finetuning with soft prompting on CodeForce. (Pass@1)

Model	Finetuning	CodeForce
Gemma-7B	N/A	0.0128
	SFT	0.0255
	CodeGRAG (soft prompting)	**0.0299**

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 and CodeForce dataset in which we collect real-world programming problems from codeforces[1] website. For CodeForce adataset we include problems categorized by different difficulty levels corresponding to the website and selecte 469 problems of difficulty level A for testing. 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), InCoder(Fried et al., 2022) and Gemma(Mesnard et al., 2024); 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 model.

4.2 Main Results

The main results are summarized in Table 1 and Table 2. 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 graphical retrieval augmented generation for code generation framework.

RQ2. The model informed by the meta-graph (CodeGRAG) could beat model informed by the raw code block. 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. From Table 2, we can see that finetuning with the expert soft prompting could offer more performance gain than that brought by simple supervised finetuning. This validates the effectiveness of the designed pretraining expert network and the technique of finetuning with soft prompting.

4.3 Study on Cross-Lingual Modeling (RQ4)

To study the capability of graphical view modeling cross-lingual thoughts of codes, we use the graphical view of each source code block to serve as a bridge for translation to another programming language. The results are in Table 3.

From the results, we could see that the bridged graphical view could offer augmentation for translation among different programming languages. 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

[1] https://codeforces.com/

Table 3: Results of code translation on Humaneval-X.

Model Size	Model	Bridge Content	Python to C++	C++ to Python
6.7B	InCoder	N/A	26.11	34.37
13B	CodeGeeX	N/A	26.54	27.18
16B	CodeGen-Multi	N/A	35.94	33.83
15B	StarCoder	N/A	0.61	26.22
15B	WizardCoder	N/A	50.00	67.07
-	GPT-3.5-Turbo	N/A	61.59	81.71
-	GPT-3.5-Turbo	Meta-Graph	**62.80**	**82.32**

Table 4: 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

different programming languages regarding solving the same problem.

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

In this section, we adjust the inputs of the graphical components to the LLMs. Concretely, we study the information contained in node names, edge names, and the topological structure. The results are presented in Table 4.

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 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 Compatibility Discussion of the Graphical Views(RQ5)

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 5.

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

Generated Codes	Pass@1	Extraction Rate
(C++) Code-RAG	62.20	92.07
(C++) CodeGRAG	64.02	92.68
(Python) Code-RAG	71.95	91.46
(Python) CodeGRAG	77.44	96.95

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 as the retrieval knowledge.

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.

Limitations

In this paper, we propose a graphical retrieval augmented generation method that can offer enhanced code generation. Despite the efficiency and effectiveness, there are also limitations within this work. For example, 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.

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.

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*.

Gemma Team Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, L. Sifre, Morgane Riviere, Mihir Kale, J Christopher Love, Pouya Dehghani Tafti, L'eonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Am'elie H'eliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Cl'ement Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikula, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Pier Giuseppe Sessa, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vladimir Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Brian Warkentin, Ludovic Peran, Minh Giang, Cl'ement Farabet, Oriol Vinyals, Jeffrey Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open models based on gemini research and technology. *ArXiv*, abs/2403.08295.

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, 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.

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*.

A Example of the inserted graphical view

An illustration of the inserted graphical view is given below.

Graph(
num_nodes='node': 24,
num_edges=('node', '-0', 'node'): 1, ('node', '-1', 'node'): 1, ('node', 'ArraySubscriptExpredge0', 'node'): 1, ('node', 'ArraySubscriptExpredge1', 'node'): 1, ('node', 'CXXOperatorCallExpredge1', 'node'): 1, ('node', 'CXXOperatorCallExpredge2', 'node'): 2, ('node', 'ImplicitCastExpredge0', 'node'): 1, ('node', 'UserDefineFun', 'node'): 1, ('node', 'falseNext', 'node'): 1, ('node', 'next', 'node'): 5, ('node', 'read', 'node'): 10, ('node', 'trueNext', 'node'): 1, ('node', 'write', 'node'): 9,
metagraph=[('node', 'node', '-0'), ('node', 'node', '-1'), ('node', 'node', 'ArraySubscriptExpredge0'), ('node', 'node', 'ArraySubscriptExpredge1'), ('node', 'node', 'CXXOperatorCallExpredge1'), ('node', 'node', 'CXXOperatorCallExpredge2'), ('node', 'node', 'ImplicitCastExpredge0'), ('node', 'node', 'UserDefineFun'), ('node', 'node', 'falseNext'), ('node', 'node', 'next'), ('node', 'node', 'read'), ('node', 'node', 'trueNext'), ('node', 'node', 'write')]
)