Boosting Code Summarization by Embedding Code Structures

Jikyeong Son, Joonghyuk Hahn, Hyeon-Tae Seo and **Yo-Sub Han**
Department of Computer Science, Yonsei University
Seoul, Republic of Korea
{jikyeong.son, greghahn, dchs504, emmous}@yonsei.ac.kr

Abstract

Recent research on code summarization relies on the structural information from the abstract syntax tree (AST) of source codes. It is, however, questionable whether it is the most effective to use AST for expressing the structural information. We find that a program dependency graph (PDG) can represent the structure of a code more effectively. We propose PDG Boosting Module (PBM) that encodes PDG into graph embedding and the framework to implement the proposed PBM with the existing models. PBM achieves improvements of 6.67% (BLEU) and 7.47% (ROUGE) on average.

We then analyze the experimental results, and examine how PBM helps the training of baseline models and its performance robustness. For the validation of robustness, we measure the performance of an out-of-domain benchmark dataset, and confirm its robustness. In addition, we apply a new evaluation measure, SBERT score, to evaluate the semantic performance. The models implemented with PBM improve the performance of SBERT score. This implies that they generate summaries that are semantically more similar to the reference summary.

1 Introduction

In the early stage of generating code summaries, researchers adopted information retrieval techniques (Marcus et al., 2004; Poshyvanyk and Marcus, 2007; Haiduc et al., 2010) to capture source code semantics. However, code summaries produced by such techniques are often inaccurate to use in practice (Wong et al., 2015). With the help of deep learning, researchers proposed the neural machine translation (NMT) frameworks that automatically produce summaries from source code (Iyer et al., 2016; LeClair et al., 2019; Liang and Zhu, 2018; Sridhara et al., 2010). Rather than using code sequence as a sole input, several models built upon the Transformer (Vaswani et al., 2017) used additional data structures and information (Lin et al., 2021; Shi et al., 2021; Choi et al., 2021) to learn obscure features that otherwise would be discarded.

Recent studies use pretrained models and implementations of graph structures to improve the performance of code summarization. Pretrained models are built upon training a huge quantity of benchmark datasets within a long period. CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021) are popular pretrained models for various code-related tasks. The graph embedding is regarded as effective on providing code semantics; especially abstract syntax tree (AST) is the most popular supplement types for reflecting the hierarchical structure of codes. Several researchers used ASTs and improved the performance of source code summarization (Alon et al., 2019; Zhang et al., 2019; Shido et al., 2019; LeClair et al., 2020). Furthermore, combining with the Transformer, there are a few improved models including mAST+GCN (Choi et al., 2021), BASTS (Lin et al., 2021), CAST (Shi et al., 2021) and SiT (Wu et al., 2021).

While ASTs are widely used to capture code structure information, they cannot capture global information between tokens well due to the deep depths (Lin et al., 2021; Shi et al., 2021; Zhang et al., 2019). Thus, recent researches consider graphs other than ASTs to capture structural information. Lin et al. (2021) pretrain the model after dividing the code based on control flow graph (CFG), and Gao et al. (2021) proposed a method of capturing the global structure by learning the data flow relationship between variables. Liu et al. (2021) utilize a new type of graph called code property graph (CPG), which combines CFG and AST, and propose a hybrid GNN using CPG.

When models utilize AST as auxiliary information, structural information is treated and delivered by token level representation. Many researchers are performed through extracting structural information from the token representations. However, such

Proceedings of the 29th International Conference on Computational Linguistics, pages 5966–5977
October 12–17, 2022.

token representations of ASTs fail to provide semantics of statements and predicates (Zhang et al., 2019). We perform code summarization by capturing the structural information through a program dependency graph (PDG) to solve such problem.

We implement the encoder module that takes PDGs of source codes as inputs to several baseline models, and evaluate the improvement from our graph module for the summary generation performance of each model. Baseline models we perform experiments are SiT (Wu et al., 2021) that applies AST on a transformer and CodeBERT (Feng et al., 2020), including the Transformer (Ahmad et al., 2020).

The experimental results of our implementation show the performance improvements of average corpus-BLEU 6.67% and Rouge-L 7.47%. However, these improvements are not sufficient to demonstrate that our implementation accurately captures structural information. Thus, we further ask the following questions based on the initial experimental results.

RQ1: Is the structural code information such as graph embedding indeed helpful for generating a code summary?

RQ2: What difference does the proposed graph structure have compared to the popularly used AST and what is better between two graph structures?

RQ3: Does the proposed model show the robust performance for out-of-domain data?

Our code is available at `https://github.com/sjk0825/coling2022`.

2 Related Works

2.1 Sequential-based Approach

Iyer et al. (2016) first proposed a method using a neural network for code summarization. Wei et al. (2019) proposed a dual framework that uses the correlation between code summarization and code generation tasks. Hu et al. (2018b) proposed a summary method using API information as well as sequence information. Ahmad (Ahmad et al., 2020) proposed a method to effectively capture the long-range dependency of a code sequence using the Transformer.

2.2 Graph-based Approach

Wan et al. (2018) applied reinforcement learning for code summarization after giving AST as sequenced information. LeClair et al. (2019) proposed a method that provides sequential and AST to independent GRUs. Hu et al. (2018a) proposed traversing the AST in a structure-based traversal method for code summarization. Transformer-based method for learning has also been proposed. Choi et al. (2021) proposed AST representing through GCN based on the Transformer. Wu et al. (2021) also suggested using a transformer-based multiview graph.

For other graph types such as CFG and PDG, the nodes are in a statement level varying from AST's nodes in a token level. Such graphs are used for a code representation method in several studies. Lin et al. (2021) proposed a method of pretraining syntax information after splitting based on a control flow of CFG. Yamaguchi et al. (2014) showed CPG combined with CFG and AST. Liu et al. (2021) performed code summarization of CPG through Hybrid GNN. We also use PDG for code structural representation to obtain the structure information of the code.

2.3 Pretrained Model-based Approach

CodeBERT is a bimodal pretrained model that performs the NL-PL task. It is a model for Masked Language Modeling and Replaced Token Detection tasks, pretrained with the dataset CodeSearchNet (Husain et al., 2019). GraphCodeBERT (Guo et al., 2021) is the first pretrained model using data flow. The model was constructed through Masked Language Modeling, Edge Prediction, and Node Alignment, and the dataset is CodeSearchNet like CodeBERT. CodeT5 (Wang et al., 2021) is also an integrated model of encoder-decoder pretrained for code related tasks. It was pretrained on tasks such as Identifier-aware Denoising Pretraining, Identifier Tagging, and Masked Identifier Prediction through CodeSearchNet and BigQuery dataset.

3 Methodology

We propose PBM (PDG Boosting Module) which improves the capability of capturing the structure information by embedding PDG to the encoder. In this section, we explain what PDG is and then, show how our PBM embeds PDG. Finally, we demonstrate the implementation of our PBM with the baseline models of the code summarization task. We illustrate the overview of the framework implementation of PBM in Figure 2. Our module applies the PDG to improve the baseline models but for better analysis, we also develop a module for ASTs that act the same as our PBM and show

the experimental results in Section 4.

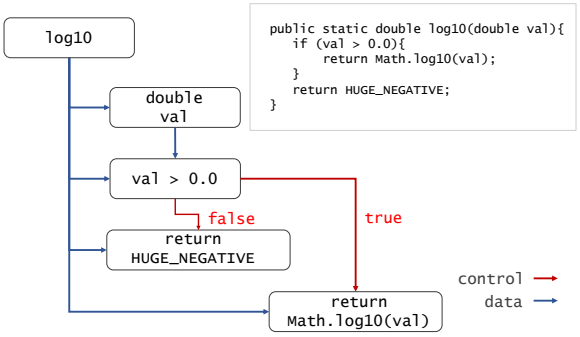


Figure 1: An example of PDG corresponding to the JAVA code instance. We represent control flow of the code with red colored edges and data flow with blue colored edges.

3.1 Program Dependency Graph

PDG is a type of graph to represent the dependency flow of code in statement level, proposed by Ferrante et al. (1987). The graph consists of statement nodes and predicated nodes, which express operators and operands of a source code, respectively. Edges between nodes express dependencies including data dependency and control dependency. Depicted in Figure 1, edges in blue color show a data flow among variables and represent data dependency. Similarly, edges in red color represent control dependency that corresponds to the dependency influenced by the values of predicate nodes. In Figure 1, control dependencies starting from a conditional statement '`val > 0.0`' depend on the value of '`val`'. Unlike AST, each node of PDG contains the partial semantics of a source code in statement level and each edge shows a connection between statements. We suspect that such tendency can express the structural information of the source code effectively and helps the models to train better for generating code summaries. The AST of the source code in Figure 1 is in Appendix A.

3.2 Graph Embedding

Formally, we represent the input graph as $G(N, E_c, E_d)$, where N is the set of PDG nodes, E_c is the set of edges for control dependency and E_d is the set of edges for data dependency. An edge $(u, v) \in E_i$ for $i \in \{c, d\}$ denotes an edge from u to v, where u and v are nodes of G. Given a PDG $G(V, E)$, where $E = E_c \cup E_d$, each edge is represented as an embedding matrix M of size $|N| \times |N|$. Each node is also represented with an embedding matrix of the same size to M.

We propose a graph embedding module that takes an extracted PDG from source codes as inputs. The extracted PDGs divide codes into statement level that are embedded as nodes and edges from the PDGs express the control and data dependencies between nodes. Then, the output of our module is concatenated with the output of a baseline model encoder. The detailed implementation is provided in Section 3.3.

Node Encoder A node encoder extracts structural information from the input graph. The encoder follows the structure of a baseline model encoder and takes the same approach of a baseline for encoding inputs. The difference from the encoder of baselines is that our node encoder calculates the attention of Key (K), Query (Q) and Value (V) that comes from the program dependency.

We present an attention equation for learning structural information from the graph embeddings. Given a sequence, let N be the set of nodes that consists of PDG of the sequence. Then, $node \in N$ is a node from the PDG and $node_e$ is an embedding of $node$. The following equation of $N\text{-}Att$ is the attention function of the node encoder. Note that K_e, Q_e, and V_e are pooled node representations of $node_e$ and d_K is the dimension of K_e. The node encoder encodes each $node_e$ as $Node_e$. When the size of N is n, the node encoder outputs node representations N_e, where $N_e = (Node_{1_e}, \ldots, Node_{n_e})$.

$$N\text{-}Att(Q_e, K_e, V_e) = softmax(\frac{E * Q_e K_e^\top}{\sqrt{d_k}})V_e,$$
$$E = E_c + E_d.$$

Node Pooler The node pooler is the process of pooling tokens within the same node through a given sequence and statement mask. The sequence of token embedding is represented as seq_e. The node pooler takes seq_e as an input and outputs its corresponding node embedding, $node_e$. $MASK^{|seq_e|}$ consists of one hot vector and W is a trainable weight. t is a token in sequence and k is a sequence length. The following equation demonstrates the procedure of node pooler.

$$node_e = ReLU((MASK * seq_e) * W),$$
$$MASK = \begin{cases} 1 \text{ if } t_j \in node \text{ for } j = 1, \ldots, k \\ 0 \text{ otherwise.} \end{cases}$$

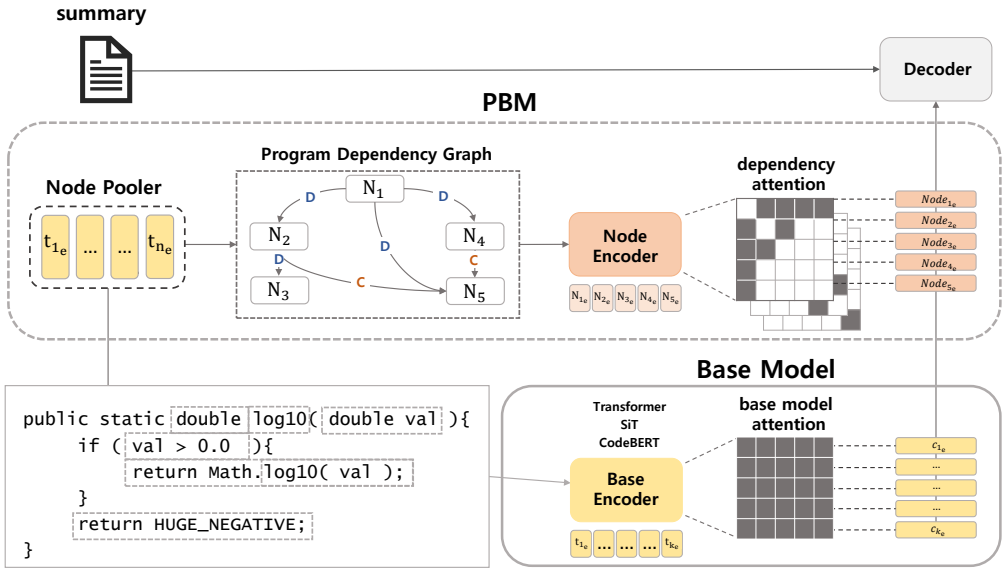


Figure 2: An architecture of PBM. Dashed boxes from the code snippet represent nodes.

3.3 PDG Boosting Module (PBM)

Our PBM module improves the performance of generating source code summaries by combining with baseline models. We use the Transformer (Ahmad et al., 2020), SiT (Wu et al., 2021), CodeBERT (Feng et al., 2020) as baseline models. The encoder of each baseline model is based on a transformer and emits token level output. We show how the separate embeddings of a sequence and graph for a code instance proceeds and combines in PBM.

Combine Encoder The input sequence consists of $(t_1, \ldots, t_k)$ tokens. The token passes through the base encoder ($B\text{-}encoder$) with the embedding vector $(t_{1_e}, \ldots, t_{k_e})$ of tokens and output $C_e = (c_{1_e}, \ldots, c_{k_e})$ containing sequence information. Token representations are concatenated with $N_e = (Node_{1_e}, \ldots, Node_{n_e})$ of the PDG module. Let k be the length of code sequence and n be the length of node sequence.

$$C_e = B\text{-}encoder(t_{1_e}, \ldots, t_{k_e}),$$
$$PBM = Concat([C_e; N_e]).$$

Decoder The decoder for PBM is dependent on baseline models we combine with. As the encoder of PBM combines both the sequential information and node information, the decoder takes the attention for both information to the target summaries. Figure 2 illustrates the full architecture of the PBM module and how PBM is connected to a baseline model.

4 Experiments and Analysis

Our experiment uses two benchmark datasets for the code summarization task. The first is CCSD (Liu et al., 2021) which is the dataset of C programs and the next is TL-CodeSum (Hu et al., 2018b) which is the dataset for JAVA programs. The details of each dataset are illustrated in Table 1.

Dataset	TL-CodeSum	CCSD
Train	69,708	84,316
Valid	8,714	4,432
Test	8,714	4,093
Out-domain Test	-	2,440

Table 1: Statistics on the number of data for the benchmark datasets.

The evaluation metrics are corpus-BLEU (Papineni et al., 2002) and ROUGE-L (Lin, 2004) score that are widely used for the verification of code summarization performance. We denote them as BLEU and ROUGE. Additionally, we use SBERT (Reimers and Gurevych, 2019) score to address the semantic performance that cannot be measured by the prior two metrics.

As we apply PBM to baseline models, one experiment is to compare the performance improvements by adding PBM to baseline models. Another experiment is to check which graph type is more suitable for our task. For the second question, we consider two graph types, AST and PDG, each of which is

constructed from data via joern[1] and srcml[2].

4.1 Baselines

PBM compare with four code summarization models that have neural network archtecture. Our baseline models include models based on the architecture of an RNN and a transformer. We also take a pretrained model as one of the baseline models.

Seq2Seq is based on the recurrent neural network architecture. Iyer et al. (2016) proposed a code summarization using Seq2Seq.

Transformer (Ahmad et al., 2020) is consturcted from a transformer (Vaswani et al., 2017)-based model using copy mechanism and relative positional encoding. Through self-attention of a transformer, the long-range dependency of code is effectively captured.

CodeBERT (Feng et al., 2020) is a pretrained encoder model with PL-NL bimodal. CodeBERT supports code-related downstream tasks including the code documentation generation task. We reproduce the CodeBERT-base model.

SiT (Wu et al., 2021) is a model trained with multiview on the structure of codes. Multiview includes AST, data dependency and statement. Multiview is trained through weighted attention at the token level.

4.2 Evaluation Metrics

Our analysis relies on BLEU and ROUGE as evaluation metrics that are popularly used in recent studies. These metrics check how the summaries capture the actual words that are used in the reference summaries. As the metrics only check whether a token or a sequence of tokens are the same, researchers argue about their reliability (Reiter, 2018; Mathur et al., 2020). Therefore, we also use another metric, SBERT (Reimers and Gurevych, 2019) score, to measure how the generated summaries capture the semantics of the source code.

BLEU (Papineni et al., 2002) measures the performance of predicted summaries through n-gram comparison with reference. The average performance is measured for the range of n to 1-4.

ROUGE (Lin, 2004) is an n-gram measurement method based on recall. The Rouge-L we use is the F-measure of prediction and reference based on the longest common sequence.

SBERT (Reimers and Gurevych, 2019) is a siamese network using pretrained BERT and measures the similarity between two sentences with a fixed sentence representation. We use the checkpoint, `all-mpnet-base-v2 mode` for the evaluation.

4.3 Experimental Setup

Hyperparameter Generally, we follow the same hyperparameter settings of baseline models to reproduce performance of the considered models. For adding our PBM to the baseline models, we set the size of the PBM layer as the size of an encoder of the corresponding baseline encoder. However, as we run experiments with multiple baseline models and the models attached with PBM, we regulate some hyperparameters such as batch size, number of epochs, and learning rate for fair performance comparison.

Device We conduct experiments on a workstation on Ubuntu 18.04 with two RTX3090 GPUs. The version of CUDA and cuDNN for GPU usage are 11.0.3 and 8, respectively.

4.4 Analysis

RQ1: Effectiveness of graph embedding We implement PBM to baselines for the code summarization task and the performance improved as shown in Table 2.

Table 2 shows the overall performance of the experimental models. PBM raises the BLEU and ROUGE performance of the Transformer, SiT and CodeBERT. The BLEU performance of the CCSD benchmark dataset of the three models increases by 7.67% on average after PBM, and the ROUGE is improved by 10.37%. The BLEU performance of the TL benchmark dataset is improved by 5.67% on average after PBM, and the ROUGE is improved by 4.57%.

Figure 3 shows the generated summaries by baseline models and the models implemented with PBM for a given source code instance. Implementation of PBM captures a word ‘gap’ of the reference that was not captured in the baseline Transformer and SiT models. The Transformer captures sequential information such as ‘calculate’ and ‘true’. But the model does not capture the objective for the ‘calculate’ word. After PBM application, the Transformer captures the object for the calculation. However, it does not capture the information of cells being rectangle. The SiT also inferences that the source code instance is a calculation. In addition, the SiT captures the information in if-statement of

[1] https://github.com/joernio/joern
[2] https://www.srcml.org/

	TL-CodeSum			CCSD		
	BLEU	ROUGE-L	SBERT	BLEU	ROUGE-L	SBERT
Seq2Seq	39.12	50.33	0.6333	20.81	23.12	0.3619
Transformer	44.34	53.74	0.6352	24.26	26.50	0.3965
CodeBERT	36.82	50.07	0.6824	22.98	29.06	**0.5256**
SiT	45.76	55.58	0.6694	25.00	26.83	0.4289
Transformer+PDG (w/o data dependency)	45.93	55.21	0.6557	25.38	28.64	0.4326
Transformer+PDG (w/o control dependency)	45.91	55.39	0.6580	26.00	29.04	0.4307
Transformer+PDG	46.07	56.68	0.6608	26.83	30.14	0.4419
CodeBERT+PDG (w/o data dependency)	40.96	52.59	0.6897	23.73	29.78	0.5235
CodeBERT+PDG (w/o control dependency)	41.17	52.91	0.6960	23.37	30.03	0.5241
CodeBERT+PDG	40.75	52.85	**0.6968**	23.41	29.45	0.5252
SiT+PDG (w/o data dependency)	45.93	56.66	0.6728	27.30	30.83	0.4452
SiT+PDG (w/o control dependency)	46.71	56.50	0.6719	27.26	27.26	0.4417
SiT+PDG	**46.86**	**56.69**	0.6752	**27.63**	**31.15**	0.4898

Table 2: Our result for Java and C dataset. The best scores for each metric are in bold.

the code that the Transformer omitted. The if statement, however, does not catch the gap of the two rectangles. On the other hand, after applying PBM, the SiT captures the semantics of calculating the gap between two rectangles.

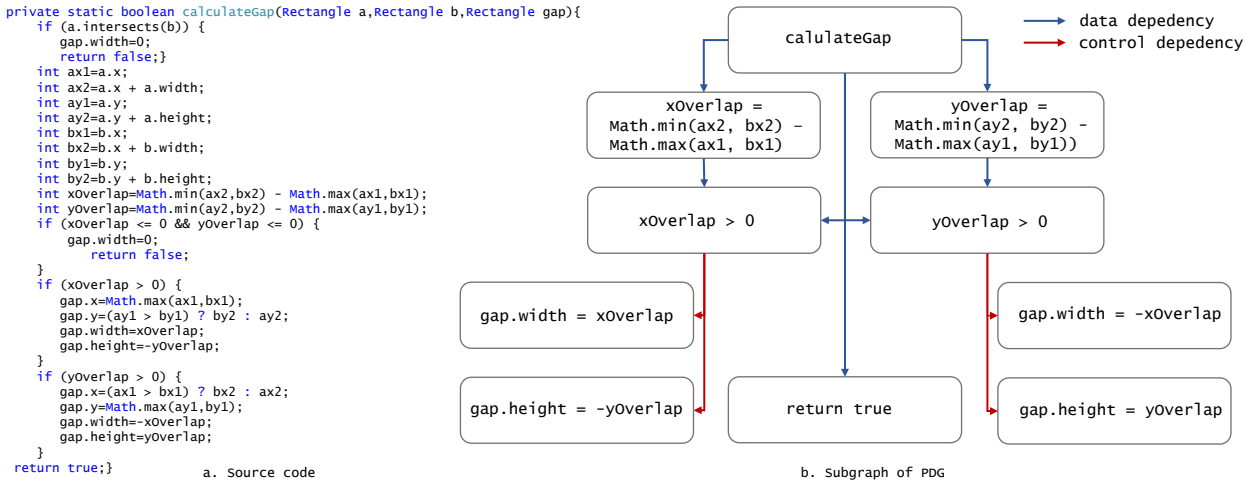
BLEU and ROUGE are performance measures based on word overlap. These methods consider the importance of capturing the exact words that are used in the reference summaries. Even though widely used, the metrics still have drawbacks that they cannot capture the semantics of the generated sequences (Haque et al., 2022). For instance, a code instance given in Figure 3 has a reference summary of 'calculate the gap rectangle between two rectangles'. If the model generates a summary such as 'calculate the gap between rectangles (a | b (b)', it does not have any defects in semantics. However, the BLEU and ROUGE score metrics conclude that the generated summary is not perfect. On the other hand, summaries illustrated in Figure 3 show the similar or better BLEU performance to 'calculate the gap between rectangles (a | b (b)' even though the sentence does not make sense.

Therefore, moving forward from only checking whether the models generate summaries that capture the words used in the reference, we also implement an auxiliary evaluation metric SBERT score to measure the performance in semantics. SBERT is pretrained from a large corpus of natural language sentence pairs and can measure the similarity between sentences. The result of the semantic measurement is depicted in Table 2. SBERT score increases by 8.53% and 2.30% in CCSD and TL-CodeSum, respectively. When applied to CodeBERT, PBM shows weaker performance, but as the score difference is not critical and as the average performance compared with other models increases significantly, we find PBM effective.

RQ2: Comparison with graph types What difference does the proposed graph structure have compared to the popularly used AST and what is better between two graph structures?

We propose the approach that implements a graph embeddings to capture the structural information and semantics by combining both structural and sequential sequences of a source code instance. The approach shows that the graph embeddings and the structural information improves the performance of code summarization. It is, however, questionable which graph structure is suitable for generating summaries. Based on the characteristics of graph structure, we implement AST, a frequently used graph type and PDG, which we find the most effective in code summarization.

AST represents auxiliary structural information widely used in code summarization. AST is a tree representing the code structure as an abstract syntax and consists of syntax nodes for the grammar structure of program and syntax tokens for the code sequence. Each node of AST consists of a single token used in the source code. The AST for the source code instance in Figure 3 is in Appendix A. AST captures the grammar of source code which is different from that of natural language sequence. On the other hand, PDG expresses the program structure with control and data dependency information and each node of PDG consists of a segment of source code in a statement level. In that sense, the module helps the model to train the code infor-




```
Transformer       : calculates true if the rectangle rectangle ( square into keyqualifier so that are drawn in the specified rectangle
Transformer + PDG : calculate the minimum of the two cells that are equal .

SiT               : calculates true if two rectangles are equal . the rectangles . the bounds .
SiT + PDG         : calculates the gap between two rectangles ( a | b ( b ) .

CodeBERT          : calculates the gap between two rectangles .
CodeBERT + PDG    : calculates the gap between two boxes . returns the size of the gap that is the gap , which is the only valid for a bounding box .

reference         : calculate the gap rectangle between two rectangles
```

Figure 3: An illustrated example of PDG and generated summaries from the baseline models and the ones implemented with PBM.

	TL-CodeSum			CCSD		
	BLEU	ROUGE-L	SBERT	BLEU	ROUGE-L	SBERT
Transformer+AST	45.57	54.86	0.6480	26.51	29.62	0.4413
CodeBERT+AST	40.54	52.41	0.6877	23.36	29.55	0.5222

Table 3: Peformance of AST module.

mation in statement and predicate level.

We analyze the difference between the AST and PDG modules by comparing the performance of both implementations. The result is shown in Table 3 and by the performance, we confirm that the implementation of the PDG module is superior in both capturing the exact words used in the reference and the semantic similarity.

For a fair comparison, we use the same graph embedding implementation of PBM and perform experiments for baseline models, Transformer, SiT and CodeBERT. Table 3 shows the overall scores of BLEU, ROUGE and SBERT score of both modules. For each baseline model, PBM shows the average of TL-CodeSum performance of BLEU and ROUGE respectively by 0.8% and 2.1% better than that of the AST module and the average of CCSD performance of BLEU and ROUGE respectively by 0.9% and 0.8% better than that of the AST module depicted in Table 3. In addition to this, SBERT shows that PBM achieves better semantic similarity than the AST module in TL-CodeSum and shows similar performance in CCSD.

Aside from the Transformer and CodeBERT, SiT already uses structural information in the AST. Implementation of AST module results in adding overlapped the same information of AST, so we do not perform additional experiments implementing the AST module for SiT. We still can confirm that the usage of PDG information is better than AST implicitly, as the performance of SiT improves when implementing PBM.

The AST and PDG both use the graph information extracted from the same source code but the performance varies. We find that each node of AST corresponds to the token of source code sequence and the structure of a graph is too complex compared to its corresponding source code sequence to capture a valid structure information. Each node of the PDG, however, consists of statements rep-

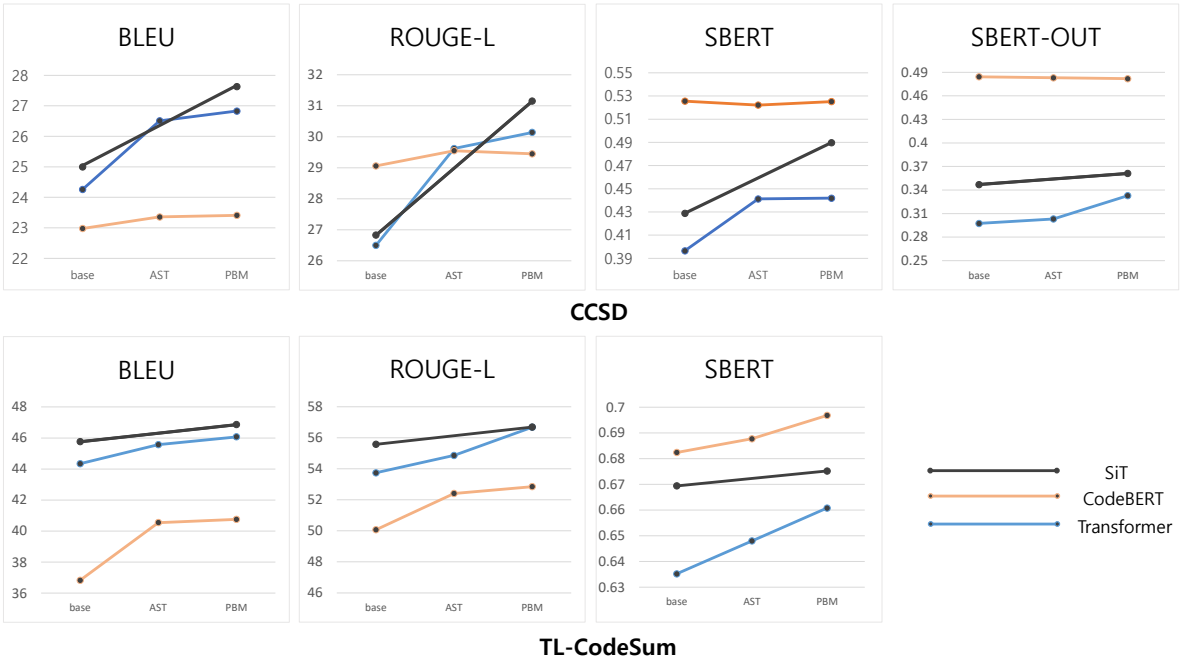


Figure 4: The performance on CCSD and TL-CodeSum. Regardless of BLEU, the higher the in-domain similarity is, the higher the out-domain similarity tends to be. Compared with AST, PBM shows a high overall performance improvement in the three category model.

resenting a relationship between statements in the form of control and data flow. This makes the PDG relatively simple and by the results shown in Table 3, we find applying PBM more effective than applying the AST module.

We evaluate the summarization ability of our module by the ablation study of edges in the PDG. Edges for PDG represent data and control dependency and we present how our module applying only one edge type performs on generating summaries in Table 2. The performance of PBM modules with one dependency edge type shows decreased performance compared to the performance of PBM module with both edge types. So we can derive that all edge types, control and data dependencies help the model to learn the structural information.

RQ3: Robustness of our framework Source code summarization is a task to generate a summary sequence for a given source code instance. Benchmark datasets are important in such research as the data to train is critical for the model. Most summaries of datasets are brought from the comments of source codes. In that sense, there is no specific guideline or tendency of summaries. Summaries can vary depending on the purpose of source codes and even the users who wrote the source codes and comments. This is why no model can be always satisfied for every source code even if the evaluation result shows good performance. On the other hand, when source codes and summaries are brought from the same repository, the data would have a similar tendency. In such cases, even the test dataset shares the same tendency and the result is not reliable as the performance score cannot verify the model to have such performance in general data.

	CCSD Out-Domain		
	BLEU	ROUGE-L	SBERT
Seq2Seq	18.70	18.91	0.3619
Transformer	20.16	17.90	0.2974
CodeBERT	20.12	24.95	**0.4843**
SiT	22.16	20.58	0.3470
Transformer+PDG	21.54	20.58	0.3328
CodeBERT+PDG	20.13	**25.08**	0.4819
SiT+PDG	**23.09**	22.93	0.3611

Table 4: Performance of CCSD out-domain dataset. The best scores for each metric are in bold.

Considering such a problem, there is a need to evaluate the model with a dataset that has a different tendency compared to the train dataset. Such aspect is called out-of-domain (OOD) and we use CCSD, the benchmark dataset for C program as

it contains both an in-domain and out-domain test dataset. The in-domain test dataset is an original dataset and the out-domain test dataset is for the OOD measurement.

Referring to the result of Table 4, performance of baseline models for the out-domain dataset is lower than that from Table 2. Such performance decrease can be up to 10% which is a critical loss. Even for this case, PBM has a robust performance improvement as Table 4 shows that it can still improve the performance of baseline models for out-domain test dataset. The performance of baseline models improves by 3.63% in average for BLEU score, by 8.97% in average for ROUGE score and by 5.17% in average for SBERT score after the implementation of PBM.

5 Conclusions

Recently, there have been several researches for improving the code summarization (Iyer et al., 2016; Feng et al., 2020; Ahmad et al., 2020). One approach is to use the structural information of source code by extracting its AST (Abstract Syntax Tree) (LeClair et al., 2020; Shi et al., 2021; Choi et al., 2021; Wu et al., 2021). While this approach works better than the one without ASTs, it turns out that a model with ASTs cannot capture the global structure owing to deep depth structure (Lin et al., 2021; Shi et al., 2021; Zhang et al., 2019).

We have studied the limit of this approach and suggested a new module PBM that utilize PDGs containing the information of control and data dependencies. We have observed that PBM improves the performance of baseline models; PBM captures the structural information of source codes in a statement level. Since BLEU and ROUGE are computed by matching tokens of the reference summary, we have considered another metric, SBERT score, for measuring the semantic difference to fully analyze the effectiveness of PBM; SBERT score evaluates the semantic similarity between sentences. The experimental results have showed that PBM achieves the performance increase in capturing the semantics of source code in the SBERT score as well. We noticed that the benchmark dataset of source codes and summaries can be vulnerable in generalization of a model. Thus, we have ran additional experiments using an out-domain test dataset of CCSD, and confirmed that our PBM is effective for the OOD case as well.

For future directions, we aim to evaluate the generalizability of the code summarization models more precisely. In addition to the out-domain dataset of CCSD, we plan to evaluate PBM with other benchmark datasets for the OOD measurements. Furthermore, we plan to design a model that has a consistently good performance on generating code summaries for different programming languages.

Acknowledgements

This research was supported by the NRF grant (NRF-2020R1A4A3079947) and the AI Graduate School Program (No. 2020-0-01361) funded by the Korea government (MSIT). The first two authors contributed equally to this work. Han is a corresponding author.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th International Conference on Learning Representations*.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics, ACL/IJCNLP 2021*, pages 2842–2851.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics, EMNLP 2020*, pages 1536–1547.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lun Yiu Nie, and Xin Xia. 2021. Code structure guided transformer for source code summarization. *arXiv preprint CoRR*, 2104.09340.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations*.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 35–44.

Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. *arXiv preprint CoRR*, 2204.01632.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2269–2275.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint CoRR*, 1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, pages 795–806.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence*, pages 5229–5236.

Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension*, pages 184–195.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *Proceedings of the 9th International Conference on Learning Representations*.

Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223.

Nitika Mathur, Timothy Baldwin, and Trevor Cohn. 2020. Tangled up in BLEU: reevaluating the evaluation of automatic machine translation evaluation

metrics. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4984–4997.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318.

Denys Poshyvanyk and Andrian Marcus. 2007. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 37–48.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3980–3990.

Ehud Reiter. 2018. A structured review of the validity of BLEU. *Computational Linguistics*, 44(3):393–401.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4053–4062.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-LSTM. In *Proceedings of the 2019 International Joint Conference on Neural Networks*, pages 1–8.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 6559–6569.

Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 380–389.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced Transformer. In *Findings of the Association for Computational Linguistics, ACL/IJCNLP 2021*, pages 1078–1090.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 590–604.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, pages 783–794.

A Abstract Syntax tree

We present also the ASTs that are omitted in the main part of the paper. Figure 5 and Figure 6 each illustrate the AST for the java code instance that was shown in Figure 1 and Figure 3 respectively.

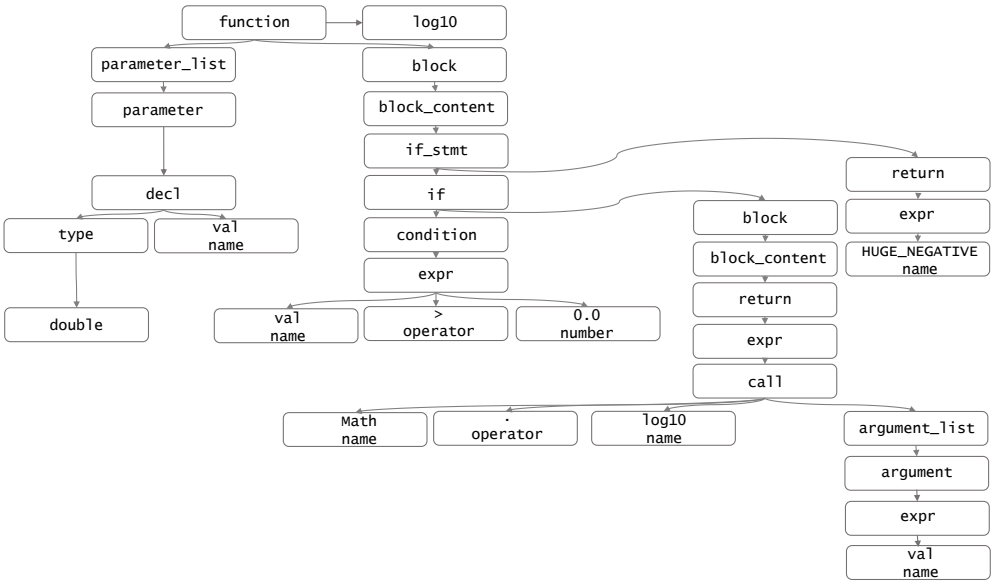


Figure 5: Graph of AST example of source code in Figure 1.

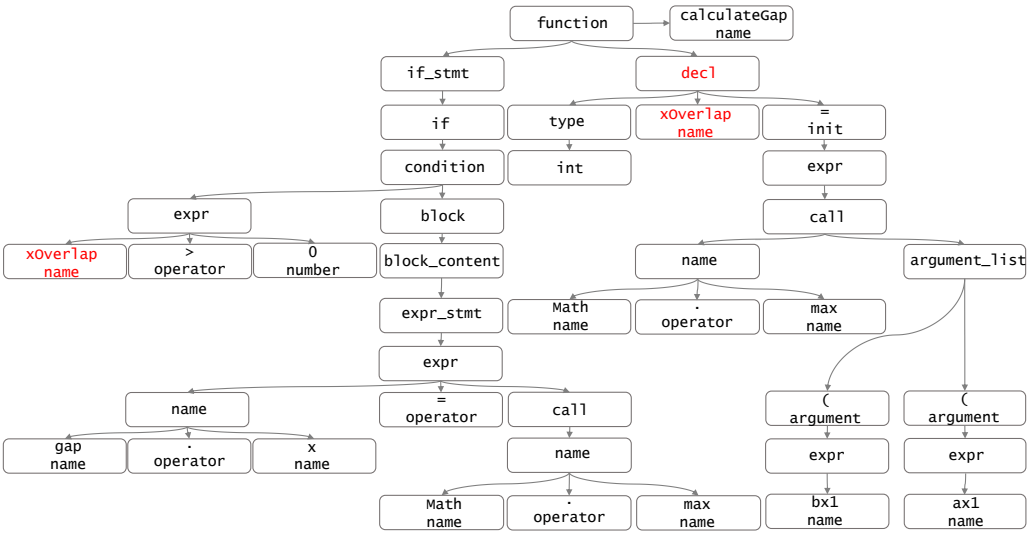


Figure 6: Subgraph of AST example of java test case.