AUTOMATIC CODE SUMMARIZATION VIA MULTI-DIMENSIONAL SEMANTIC FUSING IN GNN

Anonymous authors
Paper under double-blind review

ABSTRACT

Source code summarization aims to generate natural language summaries from structured code snippets for better understanding code functionalities. Recent works attempt to encode programs into graphs for learning program semantics and yield promising results. However, these methods only use simple code representations (e.g., AST), which limits the capability of learning the rich semantics for complex programs. Furthermore, these models primarily rely on graph-based message passing, which only captures local neighborhood relations. To this end, in this paper, we combine diverse representations of the source code (i.e., AST, CFG and PDG) into a joint code property graph. To better learn semantics from the joint graph, we propose a retrieval-augmented mechanism to augment source code semantics with external knowledge. Furthermore, we propose a novel attention-based dynamic graph to capture global interactions among nodes in the static graph and followed a hybrid message passing GNN to incorporate both static and dynamic graph. To evaluate our proposed approach, we release a new challenging benchmark, crawled from **200+** diversified large-scale open-source *C* projects (total **95k** functions in the dataset). Our method achieves the state-of-the-art performance, improving existing methods by **1.66**, **2.38** and **2.22** in terms of BLEU-4, ROUGE-L and METEOR metrics.

1 INTRODUCTION

With software growing in size and complexity, developers tend to spend nearly 90% Wan et al. (2018) effort on software maintenance (e.g., version iteration and bug fix) in the completed life cycle of software development. Source code summary, in the form of natural language, plays a critical role in comprehension and maintenance process and greatly reduces the effort of reading and comprehending programs. However, manually writing code summaries is tedious and time-consuming, and with the acceleration of software iteration, it has become a heavy burden for software developers. Hence, source code summarization which automates concise descriptions of programs is meaningful.

Automatic source code summarization is a crucial yet far from settled problem. The key challenge is to learn the complex semantics from the source code. Conventionally, information retrieval (IR) techniques have been widely used in code summarization Eddy et al. (2013); Haiduc et al. (2010); Wong et al. (2015; 2013). Since code duplication Kamiya et al. (2002); Li et al. (2006) is common in "big code" Allamanis et al. (2018), early works summarize the new programs by retrieving the code snippet that is very similar with one of the existing code database, in which the summaries of the source code are known. The retrieval-based approaches may achieve promising performance on similar programs, but are limited in utility and generalization, i.e., they have poorer performance on programs that are very different from the code database. Furthermore, these approaches utilize text similarity matching, which may only capture simple semantics. To improve the generalization performance, some works explore Seq2Seq architectures Bahdanau et al. (2014); Luong et al. (2015) for summarization. These Seq2Seq-based approaches Iyer et al. (2016); Hu et al. (2018a); Alon et al. (2018) usually treat the source code or abstract syntax tree (AST) parsed from programs as a sequence and follow a paradigm of encoder-decoder with attention mechanism for generating a summary. However, these works rely on sequential models, struggling to capture the rich semantics of source code due to the complex control dependencies and data dependencies.

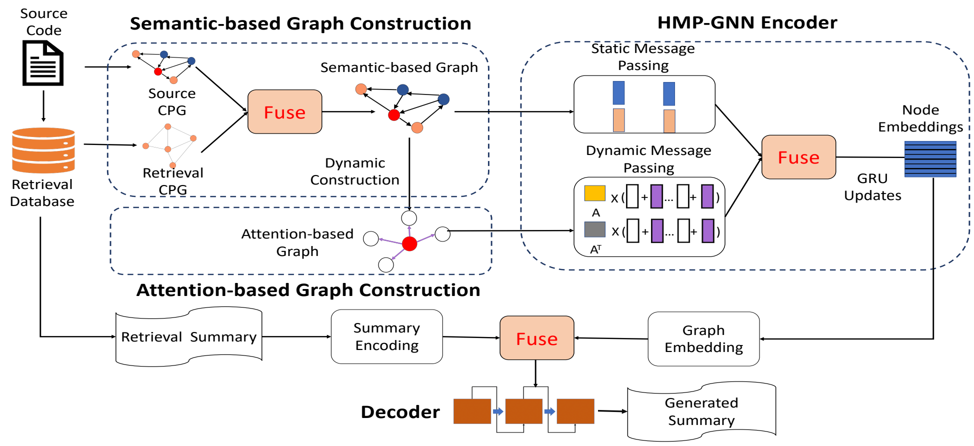

Figure 1: The framework of our *FusionGNN*. Best viewed in color.

To better learn the semantics of the source code, Allamanis et al. Allamanis et al. (2017) lighted up this field by representing programs as graphs. Some follow-up works LeClair et al. (2020); Fernandes et al. (2018) attempted to encode code semantics into code graphs with graph neural networks (GNNs), and achieved the state-of-the-art performance. However, existing graph-based approaches still have the following limitations: 1) existing methods mainly use simple representations e.g., AST, to encode semantics for GNNs. Although GNN-based techniques could improve the effectiveness than the Seq2Seq model, it is still hard to learn full semantics of the program with the simple representation. One way is to add more semantic knowledge to the graph by combining multiple representations, e.g., control flow graph (CFG), program dependency graph (PDG) and AST. However, effectively combining such representations is challenging. 2) Existing works Allamanis et al. (2017); Fernandes et al. (2018); LeClair et al. (2020) convert code into graph-structured input during preprocessing, and directly consume it via modern neural networks (e.g., GNNs) for computing node and graph embeddings. However, most GNN-based encoders only allow message passing among nodes within a k-hop neighborhood (where k is usually a small number), thus capture only local neighborhood information and ignore global interactions among nodes. This might limit their model capacity of fully encoding the source code semantics, especially for a large and complex program.

To address these challenges, we propose a framework for automatic code summarization via multi-dimensional semantic fusing in GNN, namely *FusionGNN*. Specifically, to learn comprehensive code semantics, we construct a joint code property graph (CPG) combining diverse program representations e.g., AST, CFG and PDG. To better fuse the semantics of these representations, we construct a static graph by leveraging a novel retrieval-augmented mechanism to augment source code semantics with the injected retrieved semantics into CPG. In order to capture global relations in the program, we further propose an attention-based dynamic graph by learning global attention scores (i.e., edge weights) in the aforementioned static graph. Then, a hybrid message passing GNN is performed on both static and dynamic graphs. Last, we release a new code summarization benchmark by crawling data from **200+** popular and diversified libraries in *C/C++* programming language and make it public at c-c. Our main contributions are as follows:

- We propose to fuse diverse program representations (i.e., AST, CFG, PDG) into a joint graph with a novel retrieval-augmented mechanism for better encoding code semantics.
- We innovate a hybrid message passing GNN performed on both static graph (based on code property graph) and dynamic graph (via structure-aware global attention mechanism).
- We release a new challenging benchmark for the task of source code summarization.
- Our proposed model is end-to-end trainable, achieves the state-of-the-art performance and improves existing approaches by **1.66**, **2.38** and **2.22** in terms of BLEU-4, ROUGE-L and METEOR metrics.

2 *FusionGNN* FRAMEWORK

In this section, we introduce our proposed *FusionGNN* framework, as shown in Figure 1, which mainly includes four components: 1) Semantic-based Graph Construction, which encodes the source code of a function into a code property graph with a novel semantic-augmented mechanism by retrieving the similar code in the retrieval code database to augment code semantics. 2) Attention-based Dynamic Graph Construction, which dynamically constructs a graph to capture the global relations among nodes. 3) Hybrid Message Passing (HMP)-GNN Encoder, which fuses the messages from the constructed semantic-based graph and attention-based graph to learn comprehensive code semantics. 4) Decoder, which utilizes an attention-based BiLSTM to generate a summary.

2.1 PROBLEM FORMULATION

In this work, we focus on generating summaries for functions Wan et al. (2018); Zhang et al. (2020). We define a dataset as $D = \{(c, s)|c \in C, s \in S\}$, where c is a source code in the function set C, s represents its targeted summary in S. The task of code summarization is to generate the best summary consisting of a sequence of tokens with T length $\hat{S} = \{s_1, s_2, ..., s_T\}$ which maximize the conditional likelihood $\hat{S} = \mathrm{argmax}_S P(S|C)$.

2.2 SEMANTIC-BASED STATIC GRAPH CONSTRUCTION

2.2.1 CODE PROPERTY GRAPH

We leverage the Code Property Graph (CPG) Yamaguchi et al. (2014) to combine diverse graph representations (i.e., AST, CFG, and PDG) into a single graph structure. Thus, CPG could capture comprehensive semantics of a program from different perspectives. For more details on CPG with a simple example, please refer to Appendix A. Here we describe each representation briefly as follows:

- **Abstract Syntax Tree (AST).** AST is a representation of the abstract syntactic structure of source code, which omits irrelevant details that have no effect on the semantics. Each node in AST is constituted by node type i.e., *identifier*, *callee* and the subsequence i.e., "*a*++" from the source code "*int b = a*++".
- **Control Flow Graph (CFG).** Compared with AST highlighting the syntactic structure, CFG displays statement execution order, i.e., the possible order in which statements may be executed and the conditions that must be met for this to happen. Each statement in the program is treated as an independent node as well as a designated entry and exit node. Based on the keywords *if*, *for*, *goto*, *break* and *continue*, control flow graphs can be easily built and "Flow to " edge describes this flow order between statements.
- **Program Dependency Graph (PDG).** PDG includes **data dependencies** and **control dependencies**: 1) data dependencies are described as the definition of a variable in a statement reaches the usage of the same variable at another statement. For example, a variable "*b*" is defined in the statement "*int b = a*++" and used in "*call (b)*". Hence, there is a "Reach" edge points from "*int b = a*++" to "*call (b)*". Furthermore, "Define/Use" edge denotes the definition and usage of the variable. 2) different from CFG displaying the execution process of the complete program, control dependencies define the execution of a statement may be dependent on the value of a predicate. For instance, suppose there are two statements "*int b = a*++" and "*call (b)*" that are only performed when "*a*" is even. Then, a "Control" edge points from "*if (a % 2) == 0*" to "*int b = a*++" and "*call (b)*".

Formally, one raw function c could be represented by a multi-edged CPG $g(V, \boldsymbol{A})$. Let m be the total number of nodes in V, $\boldsymbol{A} \in \{0, 1\}^{k \times m \times m}$ is the adjacency matrix, where k is the total number of edge types. An element $a^e_{i,j} \in \boldsymbol{A}$ equal to 1 indicates that node v_i, v_j is connected via an edge of type e, and 0 otherwise.

2.2.2 NODE INITIALIZATION REPRESENTATION

The nodes in CPG are represented by node type and their subsequences. To better capture the dependency in the subsequence, we utilize a BiLSTM to initialize the node. We first embed the node type into label embedding, *i.e.,* each node type is assigned a unique integer h_{type}. Then a BiLSTM is used to encode the subsequence within the node. Specifically, for a node $v \in V$, its subsequence is defined as $S_v = \{s_{v,1}, s_{v,2}, ..., s_{v,l}\}$ where l is the length of sequence. Each $\boldsymbol{s}_{v,i}$ is embedded with a learnt embedding matrix $\boldsymbol{E}$ and we use a BiLSTM to encode S_v to get the final states $\boldsymbol{h}_{seq}$. Finally, we concatenate node type representation with sequence representation and employ a linear projection to represent node feature $\boldsymbol{h}_v$. The initial node feature $\boldsymbol{h}_v \in \boldsymbol{H}$ can be expressed as follows:

$$\boldsymbol{h}_{v,1}, ..., \boldsymbol{h}_{v,l} = \text{BiLSTM}(\boldsymbol{E}_{v,1}, ..., \boldsymbol{E}_{v,l}) \quad \boldsymbol{h}_{\text{seq}} = [\boldsymbol{h}^{\rightarrow}_{v,1}; \boldsymbol{h}^{\leftarrow}_{v,l}] \quad \boldsymbol{h}_v = \text{linear}(\text{concat}[h_{type}; \boldsymbol{h}_{seq}]) \quad (1)$$

By node initialization, $g(V, \boldsymbol{A})$ can be expressed as $g(V, \boldsymbol{H}, \boldsymbol{A})$, where $\boldsymbol{H} \in \mathbb{R}^{m \times d}$ is the initial node feature matrix and each vertex v in V is represented by a d-dim real-valued vector $\boldsymbol{h}_v \in \mathbb{R}^d$.

2.2.3 Semantic-augmented Mechanism

Suppose there is a CPG $g(V_c, \boldsymbol{H}_c, \boldsymbol{A}_c)$ of source code c, our goal is to learn a function f to generate $s = f(g(V_c, \boldsymbol{H}_c, \boldsymbol{A}_c))$, however, due to the complexity of learning code semantics, f is hard to learn. Our novel semantic-augmented mechanism is motivated that for another known code-summary pair (c', s') satisfies $s' = f(g(V_{c'}, \boldsymbol{H}_{c'}, \boldsymbol{A}_{c'}))$, where c and c' are similar. A simple linear transformation can be performed $s = f(g(V_c, \boldsymbol{H}_c, \boldsymbol{A}_c)) - f(g(V_{c'}, \boldsymbol{H}_{c'}, \boldsymbol{A}_{c'})) + s'$. Compared to learn f directly from $g(V_c, \boldsymbol{H}_c, \boldsymbol{A}_c)$ to s, if we can make full use of $f(g(V_{c'}, \boldsymbol{H}_{c'}, \boldsymbol{A}_{c'}))$ and s', it can be regraded as a well supplementary for original c to learn the comprehensive semantics. Based on this, our novel semantic-augmented mechanism is performed with three steps:

Step 1: Code Retriever. For each training sample $(c, s) \in D$, we retrieve the most similar sample: $(c', s') = \text{argmax}_{(c',s') \in D'} cos\_sim(c, c')$, where $c \neq c'$, D' is a given retrieval database and $cos\_sim(c, c')$ is the cosine similarity function. Following Liu et al. Liu et al. (2018), we treat the raw source code c as a "bag of words" (BOW) vector and calculate the cosine similarity between the source code c and the retrieved code c' (i.e., $cos\_sim(c, c')$).

Step 2: Complementary Graph. After retrieving the source code c' for the training sample c, we get their corresponding CPGs $g(V_c, \boldsymbol{H}_c, \boldsymbol{A}_c)$ and $g(V_{c'}, \boldsymbol{H}_{c'}, \boldsymbol{A}_{c'})$. Then we build a complementary graph by injecting the retrieved graph into the graph of the training sample.

- To capture the relevance between the two graphs, we design an attention function, which computes the attention score matrix $\boldsymbol{A}_{i,j}$ for each pair of nodes $v_i \in V_c$ and $v_j \in V_{c'}$:

$$\boldsymbol{A} \propto exp(\text{ReLU}(\boldsymbol{W}\boldsymbol{H}_c)\text{ReLU}(\boldsymbol{W}\boldsymbol{H}_{c'})^T) \quad (2)$$

  where $\boldsymbol{W} \in \mathbb{R}^{d \times d}$ are weight matrices with d-dim size and ReLU is the rectified linear unit.
- Multiply the attention matrix $\boldsymbol{A}$ with the retrieved node representation $\boldsymbol{H}_{c'}$ to inject retrieved node features into $\boldsymbol{H}_c$:

$$\boldsymbol{H}'_c = z\boldsymbol{A}\boldsymbol{H}_{c'} \quad (3)$$

  where $z = cos\_sim(c, c')$ is the similarity score, which is introduced to weaken the negative impact of c' on c, i.e., when the similarity of c and c' is slow.
- Finally, we merge $\boldsymbol{H}'_c$ to the original $\boldsymbol{H}_c$ to get the final node representation.

$$\boldsymbol{comp} = \boldsymbol{W}_c\boldsymbol{H}_c + \boldsymbol{W}'_c\boldsymbol{H}'_c \quad (4)$$

  where $\boldsymbol{W}_c, \boldsymbol{W}'_c \in \mathbb{R}^{d \times d}$ are weight matrices and $\boldsymbol{comp}$ is the semantic-augmented node representation, namely *semantic-based static graph.*

Step 3: Retrieved Summary Encoder. The retrieved summary (i.e., s') may have a semantic overlap with the targeted summary, especially when c and c' are similar. Inspired by the existing work Yang et al. (2019), we further encode the semantics of s' with a BiLSTM model i.e., $\{\boldsymbol{h}_{s'_i}, \forall s'_i \in s'\}$ and fuse with the graph encoding results (i.e., the outputs of the GNN encoder) for the decoder, as shown in Figure 1.

2.3 Attention-based Dynamic Graph Construction

Due to that GNN-based encoders usually consider the k-hop neighborhood, the global relation among nodes in the static graph (see Section 2.2) may be ignored. In order to better capture the global semantics of source code, based on the static graph, we propose to dynamically construct a graph via structure-aware global attention mechanism, which allows message passing among any pair of nodes. We expect this attention-based dynamic graph can better capture the global dependency among nodes, and thus supplement the static graph.

Structure-aware Global Attention. The construction of the dynamic graph is motivated by the structure-aware self-attention mechanism proposed in Zhu et al. (2019). Given the static graph, we compute a corresponding dense adjacency matrix $\boldsymbol{A}$ based on a novel structure-aware global attention mechanism and the constructed graph namely *attention-based dynamic graph*. Unlike regular self-attention mechanisms, we consider not only the node semantics but also the edges in the static graph when computing attention scores between any pair of nodes. We assume that the static graph contains useful structure information which could be utilized for computing the global relationship among nodes.

$$\boldsymbol{A}_{i,j} = \frac{\mathrm{ReLU}(\boldsymbol{h}_i \boldsymbol{W}^Q)(\mathrm{ReLU}(\boldsymbol{h}_j \boldsymbol{W}^K) + \mathrm{ReLU}(\boldsymbol{e}_{ij} \boldsymbol{W}^R))}{\sqrt{d}} \tag{5}$$

where $\boldsymbol{h}_i, \boldsymbol{h}_j \in \boldsymbol{comp}$ are the node representation for any node pair (v_i, v_j), $\boldsymbol{e}_{ij} \in \mathbb{R}^{d_e}$ is the connected edge representation and $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d \times d}$, $\boldsymbol{W}^R \in \mathbb{R}^{d_e \times d}$ are parameter matrices. We also separate two normalized adjacency matrices $\boldsymbol{A}^{\dashv}$ and $\boldsymbol{A}^{\vdash}$ from $\boldsymbol{A}$ according to the incoming/outgoing directions. The dynamic graph $A^{\dashv}, A^{\vdash}$ will be used to compute dynamic message passing (see Section 2.4).

$$\boldsymbol{A}^{\dashv} = \mathrm{softmax}(\boldsymbol{A}) \quad \boldsymbol{A}^{\vdash} = \mathrm{softmax}(\boldsymbol{A}^T) \tag{6}$$

2.4 Hybrid Message Fusion

To better incorporate the information of the static graph and the dynamic graph, we propose a novel Hybrid Message Passing GNN (HMP-GNN), which are performed on both semantic-based static graph and attention-based dynamic graph.

Static Message Passing. Since the semantics-based static graph is a directed and unweighted graph, we opt to employ a bidirectional message passing GNN Chen et al. (2019) to encode the graph. Specifically, for every node v at each computation hop k, where $\boldsymbol{h}_v^0 \in \boldsymbol{comp}$, we apply a simple mean aggregation function that takes as input a set of incoming/outgoing neighboring node embeddings computed from the previous hop, and outputs a backward/forward aggregated vector $\boldsymbol{h}^k_{\mathcal{N}_{\dashv(v)}}$ / $\boldsymbol{h}^k_{\mathcal{N}_{\vdash(v)}}$. Then we fuse the above two aggregated vectors via a fusion function.

$$\begin{gathered}\boldsymbol{h}^k_{\mathcal{N}_{\dashv(v)}} = \mathrm{SUM}(\{\boldsymbol{h}_u^{k-1} + \boldsymbol{e}_{v,u}, \forall u \in \mathcal{N}_{\dashv(v)}\}) \qquad \boldsymbol{h}^k_{\mathcal{N}_{\vdash(v)}} = \mathrm{SUM}(\{\boldsymbol{h}_u^{k-1} + \boldsymbol{e}_{v,u}, \forall u \in \mathcal{N}_{\vdash(v)}\}) \\ \boldsymbol{h}^k_{sta} = \mathrm{Fuse}(\boldsymbol{h}^k_{\mathcal{N}_{\dashv(v)}}, \boldsymbol{h}^k_{\mathcal{N}_{\vdash(v)}})\end{gathered} \tag{7}$$

Here the fusion function is designed as a gated sum of two inputs.

$$\mathrm{Fuse(a, b)} = \boldsymbol{z} \odot \boldsymbol{a} + (1 - \boldsymbol{z}) \odot \boldsymbol{b} \quad \boldsymbol{z} = \sigma(\boldsymbol{W}_z[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{a} \odot \boldsymbol{b}; \boldsymbol{a} - \boldsymbol{b}] + \boldsymbol{b}_z) \tag{8}$$

where $\odot$ is the component-wise multiplication, σ is a sigmoid function and z is a gating vector.

Dynamic Message Passing. The node and edge information is propagated on the attention-based dynamic graph with the normalization adjacency matrices $\boldsymbol{A}^{\dashv}$ and $\boldsymbol{A}^{\vdash}$, defined as

$$\boldsymbol{h}^k_{v^{\dashv}} = \sum_{j=1}^{m} \boldsymbol{A}^{\dashv}_{v,j}(\boldsymbol{h}_j^{k-1}\boldsymbol{W}^V + \boldsymbol{e}_{v,j}\boldsymbol{W}^F) \quad \boldsymbol{h}^k_{v^{\vdash}} = \sum_{j=1}^{m} \boldsymbol{A}^{\vdash}_{v,j}(\boldsymbol{h}_j^{k-1}\boldsymbol{W}^V + \boldsymbol{e}_{v,j}\boldsymbol{W}^F) \tag{9}$$

where $\boldsymbol{h}_j^0 \in \boldsymbol{comp}$, m is the total number of nodes and k is the computation hop. Finally, we obtain dynamic aggregated vectors $\boldsymbol{h}^k_{dyn}$ by feeding $\boldsymbol{h}^k_{v^{\dashv}}$ and $\boldsymbol{h}^k_{v^{\vdash}}$ with fusion function defined in Eq. (8).

Hybrid Message Passing. Given the static/dynamic aggregated vectors $\boldsymbol{h}^k_{sta}$/$\boldsymbol{h}^k_{dyn}$ for static and dynamic graphs, respectively, we fuse both vectors and feed the resulting vector to a Gated Recurrent Unit (GRU) to update node representations.

$$\boldsymbol{h}_v^k = \mathrm{GRU}(\boldsymbol{h}_v^{k-1}, \mathrm{Fuse}(\boldsymbol{h}^k_{sta}, \boldsymbol{h}^k_{dyn})) \tag{10}$$

After n hops of GNN computation, we obtain the final node representation $\boldsymbol{h}_v^n$ and then apply max-pooling over all nodes $\{\boldsymbol{h}_v^n, \forall v \in V\}$ to get a d-dim graph representation $\boldsymbol{h}^g$.

Table 1: Automatic evaluation results (in %) on the CCSD test set.

Methods	In-domain			Out-of-domain			Overall		
	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR
TF-IDF	15.20	27.98	25.91	5.50	15.37	13.12	12.19	23.49	21.34
NNGen	15.97	28.14	26.11	5.74	16.33	14.27	12.76	23.93	21.96
CODE-NN	9.02	26.94	22.54	4.77	21.91	18.52	7.77	25.15	21.11
Hybrid-DRL	9.30	30.01	24.60	6.30	22.20	**21.85**	8.43	26.65	23.62
Transformer	11.82	23.25	20.22	4.76	13.80	11.35	9.64	19.88	17.06
Rencos	14.47	31.61	28.55	6.50	22.81	18.74	11.74	28.47	24.41
SeqGNN	10.51	29.84	25.04	4.94	22.80	19.17	8.87	27.34	22.97
AST2seq	11.59	29.98	26.03	5.68	22.54	20.06	9.82	27.35	23.92
FusionGNN	**16.01**	**34.07**	**29.80**	**7.31**	**24.99**	20.83	**13.40**	**30.85**	**26.63**

2.5 Decoder

The decoder is similar with other state-of-the-art Seq2seq models Bahdanau et al. (2014); Luong et al. (2015) where an attention-based BiLSTM decoder is used. The decoder takes the graph-level representation $\boldsymbol{h}^g$ as initial hidden states and concatenate the node representations with retrieved summary representations $\{\boldsymbol{h}_v^n, \forall v \in V\} \cup \{\boldsymbol{h}_{s_i'}, \forall s_i' \in s'\}$ as the attention memory and generate the summary.

2.6 Training and Inference

We train the model with regular cross-entropy loss, defined as $\mathcal{L} = \sum_t -\log P(s_t^*|V, s', s_{<t}^*)$, where s_t^* is the word at the t-th position of the ground-truth. To alleviate the exposure bias, we utilize schedule teacher forcing Bengio et al. (2015). During the inference, we use beam search to generate final results.

3 Experiments

We evaluate our proposed model against a number of state-of-the-art methods on the benchmark. We divide the existing baseline methods into three groups 1) Retrieval-based approaches: TF-IDF Haiduc et al. (2010), NNGen Liu et al. (2018) 2) Sequence-based approaches: CODE-NN Iyer et al. (2016), Transformer Vaswani et al. (2017), Hybrid-DRL Wan et al. (2018), Rencos Zhang et al. (2020) 3) Graph-based approaches: SeqGNN Fernandes et al. (2018), AST2seq LeClair et al. (2020). Note that Rencos Zhang et al. (2020) combines the retrieval information into Seq2Seq model, we classify it into Sequence-based approaches. Detailed descriptions are provided in Appendix B. Experiments on Iyer et al. (2016); Wan et al. (2018); Zhang et al. (2020) are conducted with the released code and Liu et al. (2018); Fernandes et al. (2018); LeClair et al. (2020) are utilized with default settings from the corresponding papers on our benchmark. For *FusionGNN* settings, please refer to Appendix C.

3.1 Datasets and Evaluation Metrics

Existing benchmarks LeClair et al. (2020); Barone & Sennrich (2017); Hu et al. (2018b) are all based on high-level programming language i.e., Java, Python. Furthermore, they have been confirmed to have extensive duplication, making model overfit to the training data that overlapped with the testset Fernandes et al. (2018); Allamanis (2019). We are the first to explore summarization on *C/C++* programming language and make our benchmark public c-c to benefit the academia and industry. We crawled **200+** popular *C/C++* repositories on GitHub and extract function-summary pairs. After a strict deduplication process, we kept **99k** unique function-summary pairs and name it *C/C++* Code Summarization Dataset (CCSD). To testify model generalization ability, we randomly selected some projects as the out-of-domain test set with 2,264 examples and the remaining were randomly split into train/development/test with 82,656/4,340/4,124 examples, respectively. We also use the training set as the retrieval database, i.e., $D' = D$ (see Step 1 in Section 2.2.3). The open-source code analysis platform for *C/C++* Joern Yamaguchi et al. (2014) was applied to construct code property graphs. More details about data processing, please refer to Appendix D.

Similar to previous works Zhang et al. (2020); Wan et al. (2018); Fernandes et al. (2018); LeClair et al. (2020); Iyer et al. (2016), BLEU Papineni et al. (2002). METEOR Banerjee & Lavie (2005) and ROUGE-L Lin (2004) are used as our automatic evaluation metrics. These metrics are popular

Table 2: Human evaluation results on the CCSD test set.

Methods	Syntactically correct	Semantically correct	Releance	Similarity
NNGen	3.88	3.80	3.16	3.08
Transformer	3.85	3.79	3.17	3.02
Rencos	3.96	3.84	3.31	3.16
AST2seq	3.82	3.82	3.46	3.14
FusionGNN	**4.11**	**3.95**	**3.64**	**3.47**

in machine translation, text summarization. Since these metrics are computed on text similarity, we also conduct a human evaluation study to evaluate semantic similarity. We invite 5 Ph.D students and 10 master students from computer science, who have rich C/C++ programming experiences to rate generated summaries from a set of anonymized approaches based on syntactically correct, semantically correct, relevant to the source code and similar to the ground-truth, ranking from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) on each category. Specifically, we randomly choose 50 programs for per approach with the corresponding generated summaries and ground-truths. Evaluators are required to rank the generated summary based on the defined categories. Evaluation scores are collected and averaged as final scores, where higher scores mean better quality.

3.2 Experimental Results

Table 1 shows the automatic evaluation results as compared to other state-of-the-art baselines. We find that our proposed model outperforms existing methods by a significant margin on both in-domain and out-of-domain datasets. First, on the in-domain dataset, since comprehensive semantics are embedded into graphs for learning, the performance is superior to Seq2Seq models, i.e., CODE-NN, Transformer, Rencos. Second, as we fuse the retrieved code semantics, the scores are also higher than graph2seq models, i.e., SeqGNN, AST2Seq. On the out-of-domain dataset, the scores decrease as models with no prior knowledge. The scores of retrieval methods, i.e., TF-IDF, NNGen, decrease dramatically as compared to other methods. We attribute to more unseen programs are in the out-of-domain dataset. Furthermore, an interesting phenomenon that Hybrid-DRL has a better performance on the out-of-domain dataset comparing to the in-domain dataset. We ascribe to the advantages of deep reinforcement learning to optimize evaluation metrics. On the overall dataset, combining in-domain and out-of-domain testsets, our model achieves **13.40**, **30.85** and **26.63**, outperforming existing methods by **1.66**, **2.38** and **2.22** in terms of BLEU-4, ROUGE-L and METEOR metrics.

As shown in Table 2, we perform a human evaluation on the overall dataset to assess the quality of the generated summaries by our model, NNGen, Transformer, Rencos and AST2seq in terms of syntax, semantic, relevance and similarity. As depicted in Table 1, NNGen, Rencos and AST2seq are the best retrieval-based, sequence-based, and graph-based approaches, respectively. We also compare with Transformer as it has been widely used in natural language processing. Inspection on the results, we can find that our approach can generate more natural (syntactically and semantically) summaries as compared to other baselines. Furthermore, since comprehensive code semantics are embedded by our approach, the generated summaries are more relevant to the source code.

3.3 Ablation Study

We also conduct an ablation study to evaluate the impact of different components, e.g., semantic-augmented mechanism, static graph and dynamic graph on the in-domain and overall dataset, as shown in Table 3. Since the improvement between *FusionGNN* and other methods on the out-of-domain dataset is not very obvious, due to the lack of prior knowledge, we omit this part to save space. To testify our semantic-augmented mechanism, we also conduct an experiment, namely Semantic-aug/Seq2Seq, which takes code sequences rather than graphs as the input $\boldsymbol{H}$ to compute $\boldsymbol{comp}$ and followed by Seq2Seq model for the generation. The scores are higher than Rencos, shown in Table 1, but lower than *FusionGNN*, which proves the effectiveness of the semantic-augmented mechanism. Another experiment *FusionGNN* w/o summary-encoder is performed by closing retrieved summary encoder to show both Complementary Graph, Retrieved Summary Encoder in the semantic-augmented mechanism are effective in augmenting semantics. By turning off static graph (*FusionGNN* w/o static) or dynamic graph (*FusionGNN* w/o dynamic), the performance decreases correspondingly, which shows our hybrid message passing combining both static and dynamic graphs is effective.

Table 3: Ablation study on the CCSD test set.

Methods	In-domain			Overall		
	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR
Semantic-aug/ Seq2Seq	15.21	32.88	28.87	12.62	29.12	25.28
FusionGNN w/o summary-encoder	13.26	32.24	27.77	11.16	28.54	24.72
FusionGNN w/o dynamic	15.37	32.79	28.85	12.72	29.32	25.68
FusionGNN w/o static	15.16	33.13	28.43	12.72	29.97	25.59
HMP-GNN w/ static	11.05	28.92	24.81	9.36	26.61	22.86
HMP-GNN w/ static-forward	10.77	28.29	24.45	9.11	25.90	22.31
HMP-GNN w/ static-backward	10.72	28.49	24.36	9.07	26.10	22.37
HMP-GNN w/ dynamic	11.01	28.37	24.38	9.26	25.83	22.08
HMP-GNN w/ dynamic-forward	10.70	28.50	24.44	9.07	25.80	22.19
HMP-GNN w/ dynamic-backward	10.71	28.15	24.05	9.03	25.64	21.90
HMP-GNN	11.49	29.20	25.25	9.61	27.41	23.23
HMP-GNN w/ src	12.46	31.57	27.33	10.91	27.76	24.03
FusionGNN	**16.01**	**34.07**	**29.80**	**13.40**	**30.85**	**26.63**

Table 4: Examples of generated summaries on the CCSD test set.

<table>
<tr><th>Example</th><th>Example 1</th><th>Example 2</th></tr>
<tr><td>Source Code</td><td><pre>static void strInit(Str *p){
    p->z = 0;
    p->nAlloc = 0;
    p->nUsed = 0;
}</pre></td><td><pre>void ReleaseCedar(CEDAR *c){
    if (c == NULL)
        return;
    if (Release(c->ref) == 0)
        CleanupCedar(c);
}</pre></td></tr>
<tr><td>Ground-Truth</td><td>initialize a str object</td><td>release reference of the cedar</td></tr>
<tr><td>NNGen</td><td>free the string</td><td>release the virtual host</td></tr>
<tr><td>Transformer</td><td>initialize the string</td><td>release cedar communication module</td></tr>
<tr><td>Rencos</td><td>initialize a floating poing string</td><td>release of the cancel object</td></tr>
<tr><td>AST2Seq</td><td>initialize the string</td><td>release cedar communication cedar</td></tr>
<tr><td>FusionGNN</td><td>initialize a str object</td><td>release reference of cedar</td></tr>
</table>

Consider the results of HMP-GNN, the static graph (HMP-GNN w/ static) performs slightly better than dynamic graph (HMP-GNN w/ dynamic), but combining both (HMP-GNN) can achieve better performance. We can also find that doing both forward and backward message passing is beneficial. HMP-GNN achieves 11.49 and 9.61 in the BLEU-4 on the in-domain and overall data set, which is lower than generation models (i.e., Transformer and AST2seq), shown in Table 1. However, as compared to Transformer, ROUGE-L and METEOR of HMP-GNN are much higher on in-domain and overall dataset and our *FusionGNN* outperforms Transformer by a significant margin. Since AST2seq takes source code sequence as another input, we also perform a comparative experiment by combining source code sequences with HMP-GNN (i.e., HMP-GNN w/ src). The results demonstrate that HMP-GNN w/ src outperforms HMP-GNN and AST2seq.

3.4 Case Study

To perform qualitative analysis, we present two examples with generated summaries by different methods from the overall data set, shown in Table 4. More examples will be presented on our website c-c. We can see that, in the first example, our approach learns more code semantics, i.e., p is a self-defined struct variable. Thus, we could generate a token *object* for the variable p. However, other models can only produce *string*. Example 2 is a more difficult function with the functionality to "release reference", as compared to other baselines, our approach effectively captures the functionality and generates more precise summary.

4 Related Work

Source Code Summarization Early works Eddy et al. (2013); Haiduc et al. (2010); Wong et al. (2015; 2013) for code summarization focused on using information retrieval to retrieve summaries. Later works attempted to employ attentional Seq2Seq model on the source code Iyer et al. (2016) or some variants from code text, i.e., AST Hu et al. (2018a); Alon et al. (2018) to generate summaries. However, these works are based on sequential models, ignoring rich code semantics. Some latest attempts LeClair et al. (2020); Fernandes et al. (2018) embedded program semantics into graph neural networks. However, these works mainly use simple representations, which are limited to learn full program semantics.

Graph Neural Networks Over the past few years, GNNs Li et al. (2015); Hamilton et al. (2017); Kipf & Welling (2016) have attracted increasing attention with many successful applications in computer vision Norcliffe-Brown et al. (2018), natural language processing Xu et al. (2018a). Because by design GNNs can model graph-structured data, recently, some works have extended the widely used Seq2Seq architectures to Graph2Seq architectures for various tasks including machine translation Beck et al. (2018), and graph (e.g., AMR, SQL)-to-text generation Zhu et al. (2019); Xu et al. (2018b). Some works have also attempted to encode programs with graphs for diverse tasks e.g., VARNAMING/VARMISUSE Allamanis et al. (2017), Source Code Vulnerability Detection Zhou et al. (2019). As compared to these works, we innovate a hybrid message passing GNN performed on both static graph and dynamic graph for better message fusion.

5 Conclusion and Future Work

In this paper, we proposed to fuse diverse program representations into a joint graph with the semantic-augmented mechanism for better source code summarization. To capture global semantics among nodes, we developed a hybrid message passing GNN performed on both static and dynamic graphs. The evaluation shows that our approach improves state-of-the-art techniques substantially. Future directions include exploring more effective ways to learn graph structures, combining other information, e.g., API knowledge for code summarization, and graph robustness analysis for GNN.

References

C code summarization. `https://sites.google.com/view/c-code-summarization`.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

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

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.

Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017.

Daniel Beck, Gholamreza Haffari, and Trevor Cohn. Graph-to-sequence learning using gated graph neural networks. *arXiv preprint arXiv:1806.09835*, 2018.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 1171–1179, 2015.

Yu Chen, Lingfei Wu, and Mohammed J Zaki. Reinforcement learning based graph-to-sequence model for natural question generation. *arXiv preprint arXiv:1908.04942*, 2019.

Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 13–22. IEEE, 2013.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1165–1173. JMLR. org, 2017.

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

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pp. 1024–1034, 2017.

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

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018b.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 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)*, pp. 2073–2083, 2016.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 135–146. IEEE, 2017.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. Improved code summarization via a graph neural network. *arXiv preprint arXiv:2004.02843*, 2020.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3): 176–192, 2006.

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

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 373–384, 2018.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Will Norcliffe-Brown, Stathis Vafeias, and Sarah Parisot. Learning conditioned graph structures for interpretable visual question answering. In *Advances in Neural Information Processing Systems*, pp. 8344–8353, 2018.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pp. 311–318. Association for Computational Linguistics, 2002.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

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

Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 562–567. IEEE, 2013.

Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 380–389. IEEE, 2015.

Kun Xu, Lingfei Wu, Zhiguo Wang, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*, 2018a.

Kun Xu, Lingfei Wu, Zhiguo Wang, Mo Yu, Liwei Chen, and Vadim Sheinin. Sql-to-text generation with graph-to-sequence model. *arXiv preprint arXiv:1809.05255*, 2018b.

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

Liu Yang, Junjie Hu, Minghui Qiu, Chen Qu, Jianfeng Gao, W Bruce Croft, Xiaodong Liu, Yelong Shen, and Jingjing Liu. A hybrid retrieval-generation neural conversation model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pp. 1341–1350, 2019.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering. IEEE*, 2020.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pp. 10197–10207, 2019.

Jie Zhu, Junhui Li, Muhua Zhu, Longhua Qian, Min Zhang, and Guodong Zhou. Modeling graph structure in transformer for better amr-to-text generation. *arXiv preprint arXiv:1909.00136*, 2019.

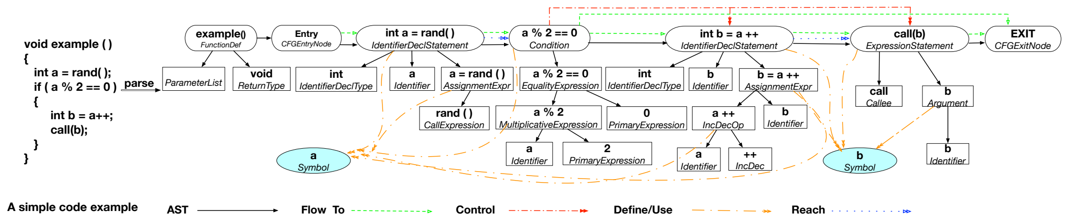

Figure 2: A example of code property graph (CPG)

Appendices

A Details on Code Property Graph

Code Property Graph (CPG) Yamaguchi et al. (2014) combines diverse graph representations (i.e., AST, CFG, and PDG) into a single graph. We describe each representation combining with Figure 2 as follows:

- **Abstract Syntax Tree (AST).** AST contains syntactic information for a program and omits irrelevant details that have no effect on the semantics. Figure 2 shows the completed AST nodes on the left simple program and each node has a code sequence in the first line and type attribute in the second line. The black arrows represent the child-parent relations among ASTs.
- **Control Flow Graph (CFG).** Compared with AST highlighting the syntactic structure, CFG displays statement execution order, i.e., the possible order in which statements may be executed and the conditions that must be met for this to happen. Each statement in the program is treated as an independent node as well as a designated entry and exit node. Based on the keywords *if*, *for*, *goto*, *break* and *continue*, control flow graphs can be easily built and "Flow to" with green dashed arrows in Figure 2 represents this flow order.
- **Program Dependency Graph (PDG).** PDG includes **data dependencies** and **control dependencies**: 1) data dependencies are described as the definition of a variable in a statement reaches the usage of the same variable at another statement. In Figure 2, the variable "*b*" is defined in the statement "*int b = a++*" and used in "*call (b)*". Hence, there is a "Reach" edge with blue arrows point from "*int b = a++*" to "*call (b)*". Furthermore, Define/Use edge with orange double arrows denotes the definition and usage of the variable. 2) different from CFG displaying the execution process of the complete program, control dependencies define the execution of a statement may be dependent on the value of a predicate. For instance, the statements "*int b = a++*" and "*call(b)*" are only performed if a is even. Therefore, a red double arrow "Control" points from "*if (a % 2) == 0*" to "*int b = a++*" and "*call(b)*".

B Details on Baseline Methods

We compare our approach with existing baselines. They can be divided into three groups: Retrieval-based approaches, Sequence-based approaches and Graph-based approaches. Experiments on Iyer et al. (2016); Wan et al. (2018); Zhang et al. (2020) are conducted with released code and Liu et al. (2018); Fernandes et al. (2018); LeClair et al. (2020) are utilized with default settings from the corresponding papers on our benchmark.

B.1 Retrieval-based approaches

TF-IDF Haiduc et al. (2010) is the abbreviation of Term Frequency-Inverse Document Frequency, which is adopted in the early code summarization Haiduc et al. (2010). It transforms programs into weight vectors by calculating term frequency and inverse document frequency. We retrieve the summary of the most similar programs by calculating the cosine similarity on the weight vectors.

NNGen Liu et al. (2018) is a retrieved-based approach to produce commit messages for code changes. We reproduce such an algorithm on code summarization. Specifically, we retrieve the most similar top-k code snippets on a bag-of-words model and prioritizes the summary in terms of BLEU-4 scores in top-k code snippets.

B.2 Sequence-based approaches

CODE-NN Iyer et al. (2016) is the first neural approach on source code summarization and adopts an attention-based Seq2Seq model to generate summaries.

Transformer Vaswani et al. (2017) is a well-known architecture and achieves a promising performance on machine translation. We use the open-source implementation provided by the OpenNMT library and train the model from scratch as one of the baselines.

Hybrid-DRL Wan et al. (2018) is a reinforcement learning-based approach, which incorporates AST and sequential code snippets into a deep reinforcement learning framework and employ evaluation metrics e.g., BLEU as the reward.

Rencos Zhang et al. (2020) is the retrieval-based Seq2Seq model for code summarization. it utilized a pretrained Seq2Seq model during the testing phase by computing a joint probability conditioned on both the original source code and retrieved source code for the summary generation.

B.3 Graph-based approaches

We also compared with two latest works, employing graph neural network for code summarization. **SeqGNN** Fernandes et al. (2018) combines GGNNs and standard sequence encoders for summarization. They take the code and relationships between elements of the code as input. Specially, a BiLSTM is employed on the code sequence to learn representations and each source code token is modelled as a node in the graph, and employed GGNN for graph-level learning. Since our node sequences are sub-sequence of source code rather than individual token, we adjust to slice the output of BiLSTM and concatenate each token representation in node sequences as node initial representation for summarization.

AST2seq LeClair et al. (2020) utilizes a recurrent layer for the source code sequence and a ConvGNN for the AST nodes and edges and then combines both for summarization. We also reproduce this approach on our benchmark.

C Model Settings

We embed the most frequent 40,000 words in the training set with 512-dims and set the hidden size of BiLSTM to 256 and the concatenated state size for both directions is 512. The dropout is set to 0.3 after word embedding layer and BiLSTM. We set GNN hops to 3 for the best performance. The optimizer is selected with Adam with an initial learning rate 0.001. We also use teacher forcing strategy with forcing probability equals to 0.8 and forcing decay is set to 0.9999. The batch size is set to 64 and early stop for 10. The beam search width is set to 5 as usual. All experiments are conducted on the dgx server with four Nvidia Graphics Tesla V100 and each epoch takes averagely 20min. All hyperparameters are tuned with grid search Franceschi et al. (2017) on the validation set.

D Details on Data Preparation

It is non-trivial to obtain high-quality datasets for code summarization. We noticed that despite some previous works LeClair et al. (2020); Barone & Sennrich (2017); Hu et al. (2018b) released their datasets, however, they are all based on high-level programming languages i.e. Java, Python. Furthermore, they have been confirmed to have extensive duplication to make model overfit to the training data that overlapped with the test set Fernandes et al. (2018); Allamanis (2019). We are the first to explore summarization on *C/C++* programming language and make our benchmark public to benefit the community research.

We crawled **200+** popular *C/C++* repositories (e.g., Linux and Redis) on GitHub, and then extracted separate function-summary pairs from these projects. Specifically, we extracted functions and associated comments marked by special characters "/**" and "*/" over the function declaration. These comments can be considered as explanations of the functions. We filtered out functions with line exceeding 1000 and any other comments inside the function, and the first sentence was selected as summary. A similar practice can be found in Jiang et al. (2017). We totally collected **360k** raw function-summary pairs. Furthermore, functions with token size greater than 150 were removed for computational efficiency and there were **130k** functions left. Since duplication is very common in existing datasets Fernandes et al. (2018), we performed a strict deduplication process and removed functions with text similarity over 80% and finally kept **99k** unique functions. We name this dataset *C/C++* Code Summarization Dataset (CCSD). To testify model generalization ability, we randomly selected some projects as the out-of-domain test set with 2,264 examples and the remaining were randomly split into train/development/test with 82,656/4,340/4,124 examples. The open-source code analysis platform for *C/C++* Joern Yamaguchi et al. (2014) was applied to construct code property graphs.