Capturing the Long-Distance Dependency in the Control Flow Graph via Structural-Guided Attention for Bug Localization

Yi-Fan Ma , **Yali Du** and **Ming Li***
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{mayf, duyl, lim}@lamda.nju.edu.cn

Abstract

To alleviate the burden of software maintenance, bug localization, which aims to automatically locate the buggy source files based on the bug report, has drawn significant attention in the software mining community. Recent studies indicate that the program structure in source code carries more semantics reflecting the program behavior, which is beneficial for bug localization. Benefiting from the rich structural information in the Control Flow Graph (CFG), CFG-based bug localization methods have achieved the state-of-the-art performance. Existing CFG-based methods extract the semantic feature from the CFG via the graph neural network. However, the step-wise feature propagation in the graph neural network suffers from the problem of information loss when the propagation distance is long, while the long-distance dependency is rather common in the CFG. In this paper, we argue that the long-distance dependency is crucial for feature extraction from the CFG, and propose a novel bug localization model named sgAttention. In sgAttention, a particularly designed structural-guided attention is employed to globally capture the information in the CFG, where features of irrelevant nodes are masked for each node to facilitate better feature extraction from the CFG. Experimental results on four widely-used open-source software projects indicate that sgAttention averagely improves the state-of-the-art bug localization methods by 32.9% and 29.2% and the state-of-the-art pre-trained models by 5.8% and 4.9% in terms of MAP and MRR, respectively.

1 Introduction

Once a bug occurs in the software system, a bug report describing the abnormal behavior is issued to the software maintenance team for locating the bug, i.e., determining which source files caused the reported bug. However, it is costly for the maintenance team to manually locate the bug, especially when the software system is large. To alleviate the burden of software maintenance, *bug localization*, which aims to automatically locate the corresponding buggy source files according to the textual description in the bug report, has drawn significant attention in the software mining community [Lukins *et al.*, 2008; Lam *et al.*, 2015; Huo *et al.*, 2016; Zhang *et al.*, 2020; Ma and Li, 2022].

Traditional methods treat source code as pure text and locate buggy source files by measuring the lexical similarity between the source code and the bug report [Gay *et al.*, 2009; Zhou *et al.*, 2012; Lam *et al.*, 2017]. Recent works indicate that the program structure in source code carries more semantics reflecting the program behavior, which is beneficial for bug localization. For example, NP-CNN [Huo *et al.*, 2016] models the correlations between neighboring statements for improving bug localization. LS-CNN [Huo and Li, 2017] further exploits the inherent sequential nature of source code. KGBugLocator [Zhang *et al.*, 2020] builds a code knowledge graph to model interrelations between classes, parameters, variables, methods, and properties for bug localization. Benefiting from the rich structural information in the Control Flow Graph (CFG), CFG-based bug localization methods have achieved the state-of-the-art performance. CG-CNN [Huo *et al.*, 2020] decomposes the CFG into multiple execution paths according to complex structures like branches and loops for multi-instance learning. To model the correlations between execution paths, cFlow [Ma and Li, 2022] employs the flow-based GRU, a specially designed graph neural network, to extract the semantic feature from the CFG, where statement features are step-wisely propagated from the entry to the exit along all the possible execution paths.

Unfortunately, the step-wise feature propagation in the graph neural network suffers from the problem of information loss when the propagation distance is long. Consequently, when using the graph neural network to extract the feature from the CFG, it is difficult for nodes to capture the key information such as variable types in the distance. However, the long-distance dependency is rather common in the CFG. Figure 1 illustrates an example of the source code and the corresponding CFG, where each statement corresponds to one node in the CFG and edges denote the possible successive execution relationship. It can be observed that the variable declaration information in the red box is crucial for understanding the return value of the function in the green box. However, it takes entire eight steps to propagate the feature of

*Ming Li is the corresponding author.

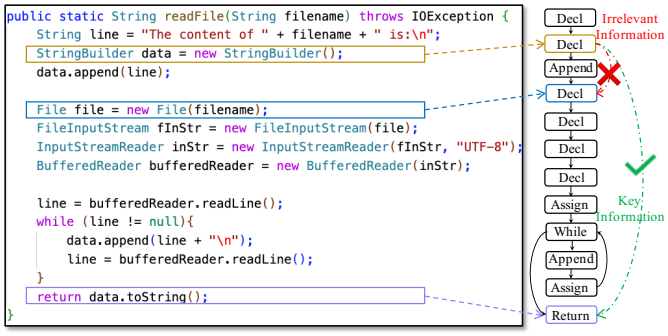

Figure 1: An example of the source code and the corresponding CFG. The declaration in the red box is the key information about the return value in the green box, while it takes entire eight steps to propagate it via the graph neural network.

the declaration statement to the return statement if the graph neural network is used. Therefore, using the graph neural network to extract the feature from the CFG may lose the key information in the distance.

To alleviate the information loss problem and capture the long-distance dependency in the CFG, a straightforward way is to extract the feature of the CFG via the global attention [Vaswani *et al.*, 2017], which allows each node to directly take features from all other nodes in the form of a weighted sum. While benefiting from the convenience of directly taking features from other nodes, employing the global attention to propagate node features in the CFG may be humbled by the irrelevant information in the global. For example, as shown in Figure 1, the two variable declaration statements in the red and blue boxes are irrelevant in semantics. Therefore, the blue node should avoid bringing in the feature of the red node in feature propagation, and vice versa.

One question arises here: how to take advantage of the global attention to capture the long-distance dependency in the CFG, while avoiding bringing in the features of irrelevant nodes? In fact, if the features of irrelevant nodes can be masked, we can still enjoy the convenience brought by the attention mechanism. To this end, we further exploit the structural information in the CFG to determine which nodes are related to it for each node. There are two kinds of dependencies that could be mined from the CFG. One is the execution dependency, which refers to whether one node will be executed or not is determined by another node, and the other is the computation dependency, which refers to the computation of one node depending on the results of other nodes or the declarations of the variables in it. By taking the features of nodes with dependency and masking the features of other irrelevant nodes, the long-distance dependency in the CFG would be carefully modeled.

In this paper, we propose a novel bug localization model named sgAttention (structural-guided Attention), which aims to capture the long-distance dependency in the CFG. In sgAttention, the structural-guided attention is employed to capture the information globally, where the execution and computation dependencies derived from the CFG are used to guide the masking of irrelevant information to facilitate better feature extraction. Experimental results based on four widely-used open-source software projects show that sgAttention outperforms the state-of-the-art bug localization methods and pre-trained models, indicating that capturing the long-distance dependency in the CFG via structural-guided attention is beneficial for improving bug localization.

The contributions of this work are summarized as follows:

- We argue that the long-distance dependency is crucial for feature extraction from the CFG. However, when capturing the long-distance dependency in the CFG, the features of irrelevant nodes should not be brought in.
- We propose a novel bug localization model named sgAttention, which employs a particularly designed structural-guided attention to capture the long-distance dependency in the CFG. In sgAttention, features of irrelevant nodes are masked for each node to facilitate better feature extraction from the CFG.
- We evaluate the performance of sgAttention on four widely-used open-source software projects. The results indicate that sgAttention averagely improves the state-of-the-art bug localization methods by 32.9% and 29.2% and the state-of-the-art pre-trained models by 5.8% and 4.9% in terms of MAP and MRR, respectively.

The rest of this paper is organized as follows. In Section 2, the proposed sgAttention model is introduced in detail. Experiments are provided in Section 3 and the related work is discussed in Section 4. Finally, this paper is concluded in Section 5 and the future work is discussed.

2 The Proposed Method

The bug localization problem is formulated as determining whether a given bug report and source file pair are correlated. Formally, let $\mathcal{C} = \{c_1, c_2, \ldots, c_{N_c}\}$ denotes the collection of source code files in a software project, $\mathcal{R} = \{r_1, r_2, \ldots, r_{N_r}\}$ denotes the set of bug reports received by the software maintenance team, and $y_{ij} \in \mathcal{Y} = \{0, 1\}$ indicates whether the source file $c_j \in \mathcal{C}$ is correlated to the bug report $r_i \in \mathcal{R}$, where N_c, N_r denote the number of source files and bug reports, respectively. The learning task of bug localization aims to learn a prediction function $f : \mathcal{R} \times \mathcal{C} \mapsto \mathcal{Y}$ by minimizing the following objective function:

$$\min_f \sum_{i,j} \mathcal{L}(f(r_i, c_j), y_{ij}) + \lambda \Omega(f), \tag{1}$$

where $\mathcal{L}(\cdot, \cdot)$ is the empirical loss, $\Omega(f)$ is the regularization term, and λ is the trade-off hyper-parameter.

2.1 The General Framework

The general framework of sgAttention is shown in Figure 2, which is composed of three encoders and a prediction layer. These three encoders aim to extract the structural feature of code $\mathbf{z}^s$, the lexical feature of code $\mathbf{z}^l$, and the bug report feature $\mathbf{z}^r$, respectively. Then, all the features are concatenated for further fusion and prediction in the prediction layer:

$$\hat{y}_{ij} = f(r_i, c_j) = f_{\text{fuse}}(\mathbf{z}^s_j, \mathbf{z}^l_j, \mathbf{z}^r_i). \tag{2}$$

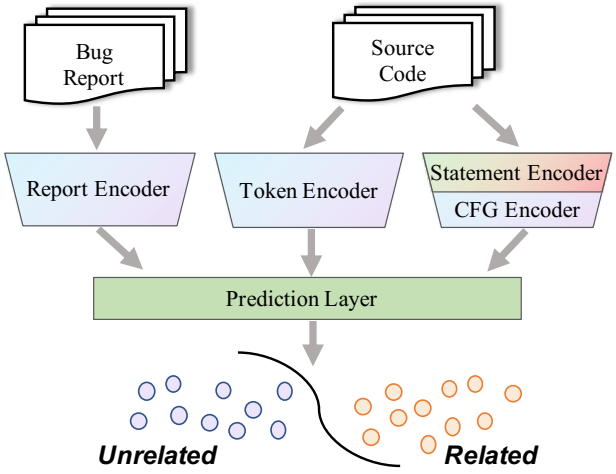

Figure 2: The general framework of sgAttention.

The prediction layer is implemented with a three-layer fully connected network, and batch normalization is applied after the first fully-connected layer to prevent overfitting.

In sgAttention, the cross-entropy loss is used as the empirical loss function:

$$Loss = -\sum_{i,j} y_{ij} \log(\hat{y}_{ij}) + (1 - y_{ij}) \log(1 - \hat{y}_{ij}), \quad (3)$$

and L_2 norm is employed for regularization.

2.2 The CFG Encoder

The CFG encoder aims to extract the structural feature $\mathbf{z}_j^s$ from the source code c_j. Since each node in the CFG corresponds to one statement in the code, we need to extract the semantic features of statements first. Therefore, we additionally insert a statement encoder before the CFG encoder to extract the statement feature x_v from the corresponding tokens.

In sgAttention, we employ the encoder of the Transformer [Vaswani *et al.*, 2017] as the basic building block, which is composed of N architecturally identical stacking layers:

$$H^n = \text{EncoderLayer}^n \left(H^{n-1}\right), n \in [1, N], \quad (4)$$

where H^0 is the embedding of input tokens. For each encoder layer, it applies a multi-head self-attention (MHA) operation followed by a feed-forward network (FFN) over the input:

$$G^n = \text{LN}(\text{MHA}(H^{n-1}) + H^{n-1}), \quad (5)$$
$$H^n = \text{LN}(\text{FFN}(G^n) + G^n), \quad (6)$$

where FFN is composed of two linear transformations with a ReLU activation in between, and LN represents the layer normalization operation. The MHA is computed via:

$$\text{MHA}(H^{n-1}) = [\text{head}_1; \ldots; \text{head}_h] W^O, \quad (7)$$
$$\text{head}_i = \text{Attn}(H^{n-1} W_i^Q, H^{n-1} W_i^K, H^{n-1} W_i^V), \quad (8)$$
$$\text{Attn}(Q, K, V) = \text{Softmax}(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}})V, \quad (9)$$

where $W^O \in \mathbb{R}^{d_h \times d_h}, W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_h \times d_k}$ are parameters, $h = d_h/d_k$ is the number of heads, and d_h, d_k are the dimension of the representation and a head, respectively.

The statement encoder only contains one encoder layer introduced in E.q. 4. We first tokenize each statement and remove unimportant punctuation such as braces, commas, and quotation marks. Then, each token term is embedded with the token embedding in CodeBERT [Feng *et al.*, 2020], which is a Transformer-architecture model pre-trained on pairs of natural language and programming language tokens. The statement encoder encodes each statement individually, respecting the atomicity of the statement explicitly.

$$x_v = \text{Max-Pooling}(\text{EncoderLayer}(\hat{X}^t)), \quad (10)$$

where $\hat{X}^t = \{[CLS], X^t, [SEP]\}$ is the input sequence, $X^t \in \mathbb{R}^{L_t \times d_t}$ is the token vector matrix of one statement, and L_t, d_t denote the number tokens in the statement and the dimension of the token vector, respectively. The $[CLS]$ is a special symbol added in front of the input sequence, and $[SEP]$ is a special separator token indicating the end of the input sequence. In order to extract the most distinguishing signal, we employ the max-pooling over the output feature sequence to extract the statement feature x_v, which is used for the input node feature in the CFG encoder.

The CFG encoder contains 6 stacking encoder layers introduced in E.q. 4, except that the global attention is replaced by the structural-guided attention. The structural feature $\mathbf{z}_j^s$ is extracted by employing the mean-pooling over the node features outputted by the last encoder layer:

$$\mathbf{z}_j^s = \text{Mean-Pooling}(\text{CFGEncoder}(X_j^v)), \quad (11)$$

where $X_j^v \in \mathbb{R}^{L_v \times d_v}$ is the node feature matrix of the CFG, and L_v, d_v denote the number nodes in the CFG and the dimension of the node feature, respectively.

Structural-Guided Attention. As aforementioned, nodes in the CFG should only take the features of nodes with dependency. To this end, instead of using the global attention introduced in E.q. 9, we particularly design a structural-guided attention in the CFG encoder. Specifically, we add a masking matrix M on the attention weight:

$$\text{sgAttn}(Q, K, V) = \text{Softmax}(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}} + M)V. \quad (12)$$

The value of the masking matrix M relies on the node dependency in the CFG. Formally, M_{ij} equals 0 if and only if the ith node is depended on the jth node, and $-\infty$ otherwise.

Figure 3 illustrates an example of the structural-guided attention. Each row in the masking matrix denotes the attention masking of other statements to the corresponding statement, where only statements with white boxes placed allow being attended to. There are two kinds of dependencies that could be mined from the CFG. One kind is the execution dependency, which refers to whether one node will be executed or not is determined by another node. Specifically, nodes in the *Loop*-body are execution-depended on the *Loop*-condition, and nodes in the *If*-body or the *Switch*-body are execution-depended on the *If*-condition or the corresponding *Case*-statement, respectively. Thus, nodes in the *Loop*-body should take the feature of the *Loop*-condition, and so

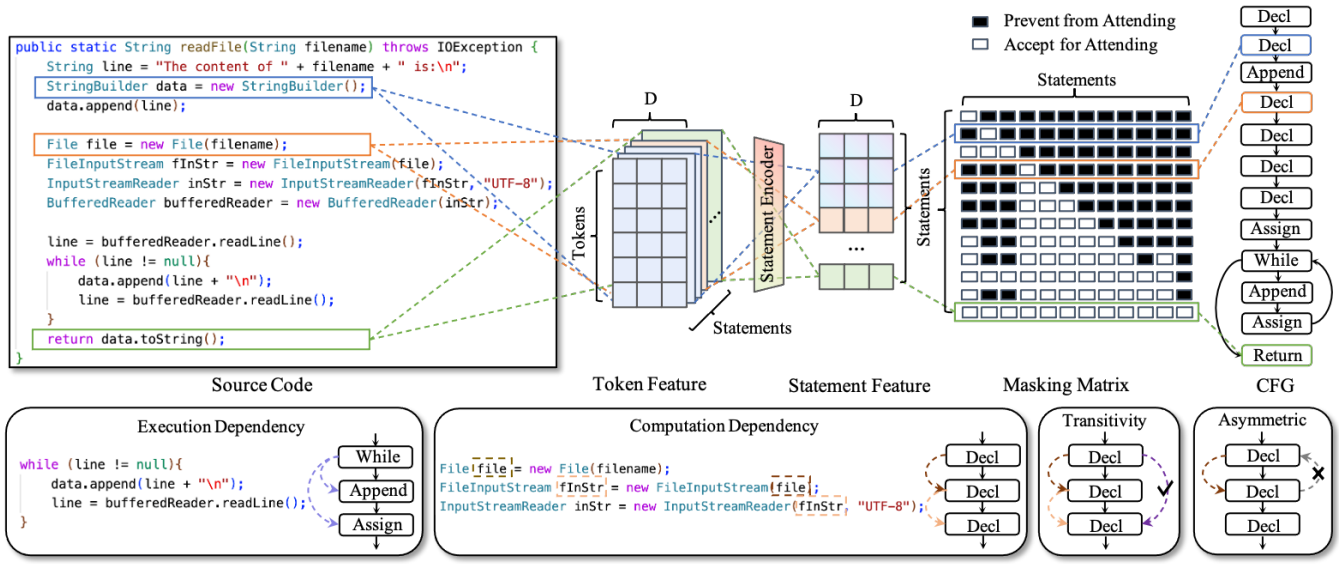

Figure 3: An example of the structural-guided attention. Only the features of the execution-depended or the computation-depended nodes are taken, which are placed with white boxes. Both the execution dependency and the computation dependency are transitive and asymmetric.

do nodes in the *If*-body or the *Switch*-body. For example, as shown in the bottom left of Figure 3, whether the *Append* and the *Assign* statements in the *While*-statement will be executed or not depends on the *While*-condition. Thus, the *Append* and the *Assign* statements are execution-depended on the *While*-condition, meaning that they could take the feature of the *While*-condition in the structural-guided attention. The other kind is the computation dependency, which refers to the computation of one node requiring the results of other nodes or relying on those nodes that declare the variables in it. For example, as shown in the bottom middle of Figure 3, the declaration of the *fInStr* variable uses the *file* variable, meaning that the declaration of *fInStr* is computation-depended on the declaration of *file*. Thus, the declaration statement of *fInStr* could take the feature of the declaration statement of *file*. Similarly, the declaration of *inStr* could take the feature of the declaration of *fInStr* in the structural-guided attention.

In addition, both the execution dependency and the computation dependency are transitive and asymmetric. As shown in the bottom right of Figure 3, the declaration of *inStr* is computation-depended on the declaration of *fInStr* and the declaration of *fInStr* is computation-depended on the declaration of *file*. We believe in that knowing the declaration of *file* ought to be beneficial for better understanding the *fInStr* variable, which is used in the declaration of *inStr*. According to the transitivity, the declaration of *inStr* is depended on the declaration of *file*. Thus, the declaration of *inStr* could also take the feature of the declaration of *file*. On the other hand, the declaration of *file* does not use the *fInStr* variable, which should be regarded as irrelevant information for it. Although the declaration of *fInStr* is computation-depended on the declaration of *file*, the asymmetry makes sure that the declaration of *fInStr* is masked for the declaration of *file*.

By taking the features of nodes with dependency and masking the features of irrelevant nodes, the long-distance dependency in the CFG could be caught without being humbled by the irrelevant information in the global.

It should be noted that adding the $[CLS]$ and the $[SEP]$ symbols in front of and behind the input is a normal operation in the sequential input and has been proven effective [Devlin *et al.*, 2019; Feng *et al.*, 2020]. However, we did not use such normal operation in the CFG encoder since it can hardly define which statement is relevant to the special symbol, resulting in special symbols would be masked by all the nodes. Although Graphormer [Ying *et al.*, 2021] suggests that adding an additional super-node connected to all the nodes and including the feature of the super-node as the graph feature are beneficial for the performance, we empirically found that including the mean of node features performs better in bug localization. See Section 3.4 for more details.

2.3 The Lexical Encoder

The lexical encoder aims to extract the lexical feature $\mathbf{z}_j^l$ from the input source code c_j. Here we use the same token embedding as in the statement encoder, while tokens from different statements are concatenated as a sequence X_j^l. Then, we finetune the CodeBERT model as the lexical feature encoder with the input sequence $\hat{X}_j^l = \{[CLS], X_j^l, [SEP]\}$:

$$\mathbf{z}_j^l = \text{CodeBERT}(\hat{X}_j^l). \tag{13}$$

We follow the suggestion in [Feng *et al.*, 2020] to include the feature of the $[CLS]$ token outputted by the last encoder layer as the lexical feature $\mathbf{z}_j^l$ of the input source code.

2.4 The Report Encoder

The report encoder takes the raw data of bug report r_i as input, and aims to extract the semantic feature $\mathbf{z}_i^r$ of it.

We use the normal natural language preprocessing method for bug report preprocessing. Specifically, we first tokenize

the bug report and remove the stop words and the punctuation. Then, all the word token terms are embedded into word vectors. We employ a Transformer encoder with 6 encoder layers, which is shown in E.q. 4, as the report encoder. The report encoder takes the $\hat{X}_i^r = \{[CLS], X_i^r, [SEP]\}$ as the input sequence, where $X_i^r \in \mathbb{R}^{L_r \times d_r}$ is the token vector matrix of the bug report, and L_r, d_r are the length of the bug report and the dimension of the token vector, respectively.

$$\mathbf{z}_i^r = \text{Transformer}(\hat{X}_i^r). \tag{14}$$

As suggested in [Devlin *et al.*, 2019], the feature of the $[CLS]$ token outputted by the last encoder layer is included as the report feature $\mathbf{z}_i^r$, which is used for further prediction.

3 Experiment

In this section, we evaluate sgAttention against a number of baseline methods based on four widely used datasets. We first introduce the datasets and the baselines in Section 3.1 and 3.2, respectively. Then, experimental results are reported in Section 3.3, followed by the ablation study in Section 3.4.

3.1 Datasets

The four datasets we used are extracted from real-world open-source software projects. As suggested in [Fischer *et al.*, 2003], the ground truth correlations between bug reports and source files are extracted from the bug tracking system (i.e., Bugzilla) and the version control system (i.e., GitHub). The statistics of the four datasets are shown in Table 1.

Eclipse Platform[1] defines a set of frameworks and common services that makes up Eclipse infrastructures. The first dataset *Platform* is extracted from the "UI" component of it. Plug-in Development Environment[2] is a tool for creating and deploying Eclipse plug-ins. The second dataset *PDE* is extracted from the "UI" component of it. Java Development Tools[3] is an Eclipse project that provides the tool plug-ins supporting the development of any Java application. The third dataset *JDT* is extracted from the "UI" component of it. The AspectJ[4] project is an aspect-oriented extension to the Java programming language, which is the last dataset and denoted as *AspectJ*. All these datasets have been extensively used in previous bug localization studies [Ye *et al.*, 2014; Huo *et al.*, 2016; Ma and Li, 2022].

As suggested in [Kochhar *et al.*, 2014], we have filtered those *fully localized* bug reports from the dataset, that is, the names of all corresponding buggy source files have been included in the bug report. For such bug reports, they no longer need a machine learning model to automatically locate the buggy source files. It should be noted that this approach makes the performance seem to be worse, but more in line with the real world situation.

[1]http://projects.eclipse.org/projects/eclipse.platform

[2]http://www.eclipse.org/pde

[3]http://www.eclipse.org/jdt

[4]http://www.eclipse.org/aspectj

Project	# source files	# bug reports	# matches	Avg. buggy files per report
Platform	6,125	5,016	18,055	3.60
PDE	5,330	2,612	10,721	4.10
JDT	10,845	5,060	14,408	2.85
AspectJ	6,908	368	1,522	4.14

Table 1: The statistic information of the four datasets.

3.2 Baseline Methods

In order to show the effectiveness of sgAttention, we compare it with the following state-of-the-art bug localization methods and pre-trained models:

- CG-CNN [Huo *et al.*, 2020]: A CFG-based bug localization method, which decomposes the CFG into multiple execution paths for multi-instance learning.
- KGBugLocator [Zhang *et al.*, 2020]: A knowledge-graph-based bug localization method, which extracts the code feature based on the code knowledge graph embedding and keywords supervised bi-directional attention.
- CodeBERT [Feng *et al.*, 2020]: A Transformer-architecture model that pre-trained on natural language and programming language pairs using both masked language modeling and replaced token detection tasks.
- GraphCodeBERT [Guo *et al.*, 2021]: A Transformer-architecture model that enhances the code representation with the data flow, pre-trained on masked language modeling, edge prediction, and node alignment tasks.
- cFlow [Ma and Li, 2022]: A CFG-based bug localization method, which employs a particularly designed flow-based GRU to step-wisely propagate node features from the entry to the exit along all the execution paths.
- FLIM [Liang *et al.*, 2022]: An information-retrieval-based bug localization method, which locates buggy files based on the cosine similarity between the report feature and the code feature extracted by CodeBERT.

To compare with these baselines, we follow the same experiment settings suggested in their studies. For hyper-parameters in sgAttention, each encoder layer contains 12 attention heads, and the dimensions of all the features are set as 768. AdamW [Loshchilov and Hutter, 2019] with a learning rate of 1e-5 and linear schedule with a warm-up ratio 0.1 are employed for optimization in sgAttention. We train our model on NVIDIA Tesla A100 with 128GB RAM, and the source code is publicly available[5].

3.3 Experimental Results

We use three metrics for evaluation: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and Top-10 Rank, all of which have been widely used in previous bug localization studies [Lam *et al.*, 2017; Zhang *et al.*, 2020; Ma and Li, 2022]. For each dataset, we randomly split the dataset into the training, validation, and test sets in a ratio of 8:1:1. We first determine the batch size, the number of training epochs, and the trade-off hyper-parameter λ based on the performance on

[5]https://www.lamda.nju.edu.cn/mayf/

Method	*Platform*	*PDE*	*JDT*	*AspectJ*	*Avg.*
KGBL	0.446	0.462	0.469	0.515	0.473
CG-CNN	0.453	0.471	0.478	0.541	0.486
CodeBERT	0.585	0.609	0.626	0.657	0.619
GCodeB	0.596	0.626	0.633	0.659	0.629
cFlow	0.464	0.486	0.489	0.563	0.501
FLIM	0.428	0.432	0.425	0.457	0.436
sgAttention	**0.632**	**0.661**	**0.669**	**0.698**	**0.665**

Table 2: The performance evaluation in terms of MAP, and the best performance is boldfaced. KGBL and GCodeB are shorts for KGBugLocator and GraphCodeBERT, respectively.

Method	*Platform*	*PDE*	*JDT*	*AspectJ*	*Avg.*
KGBL	0.526	0.578	0.567	0.618	0.572
CG-CNN	0.534	0.589	0.576	0.641	0.585
CodeBERT	0.667	0.736	0.738	0.808	0.737
GCodeB	0.674	0.739	0.744	0.809	0.742
cFlow	0.548	0.612	0.587	0.659	0.602
FLIM	0.519	0.534	0.512	0.558	0.531
sgAttention	**0.710**	**0.772**	**0.783**	**0.845**	**0.778**

Table 3: The performance evaluation in terms of MRR, and the best performance is boldfaced. KGBL and GCodeB are shorts for KGBugLocator and GraphCodeBERT, respectively.

the validation set. Then, the training and validation sets are mixed up to train the model. This process is repeated for 3 times and the average performance on the test set is reported.

The evaluation results in terms of MAP are listed in Table 2. The best performance on each dataset is boldfaced. It can be observed that sgAttention outperforms all the baselines on all the datasets in terms of MAP. It should be noticed that sgAttention achieves the best average MAP performance (0.665), which improves KGBugLocator (0.473) by 40.6%, CG-CNN (0.486) by 36.9%, CodeBERT (0.619) by 7.4%, GraphCodeBERT (0.629) by 5.8%, cFlow (0.501) by 32.9%, and FLIM (0.436) by 52.7%.

The evaluation results in terms of MRR are listed in Table 3, with the best performance on each dataset shown in bold. Similar to the performance in terms of MAP, sgAttention achieves the best MRR performance on all the datasets. In addition, sgAttention achieves the best average MRR performance (0.778), which improves KGBugLocator (0.572) by 35.9%, CG-CNN (0.585) by 32.9%, CodeBERT (0.737) by 5.5%, GraphCodeBERT (0.742) by 4.9%, cFlow (0.602) by 29.2%, and FLIM (0.531) by 46.5%.

Across all the four datasets, sgAttention improves GraphCodeBERT, the second best method, by 5.6% to 6.0% in terms of MAP and by 4.4% to 5.3% in terms of MRR. Besides, sgAttention improves the state-of-the-art bug localization methods by at least 24.0% in terms of MAP and by at least 26.1% in terms of MRR, which should be considered a huge improvement over the state-of-the-art.

The evaluation results in terms of Top-10 Rank are shown in Figure 4. Not surprisingly, sgAttention once again outperforms all the baselines on all the datasets. Higher Top-10 Rank values indicate that more buggy source files could be correlated by sgAttention when the same number of potential files are examined. Considering that only a limited number of potential files provided by the bug localization model will be checked, sgAttention would be more useful in practice.

In summary, experimental results on four widely-used open-source software projects indicate that sgAttention outperforms the state-of-the-art bug localization methods and pre-trained models in terms of all the three commonly used metrics, demonstrating that capturing the long-distance dependency in the CFG via the structural-guided attention is beneficial for improving bug localization.

Method	*Platform*	*PDE*	*JDT*	*AspectJ*	*Avg.*
sgAttention	**0.632**	**0.661**	**0.669**	**0.698**	**0.665**
- w/o CFG	0.585	0.609	0.626	0.657	0.619
- w/o token	0.567	0.570	0.576	0.632	0.586
- w/o mask	0.600	0.634	0.641	0.677	0.638
- w/o mean	0.618	0.648	0.661	0.685	0.653

Table 4: The ablation study evaluation in terms of MAP.

Method	*Platform*	*PDE*	*JDT*	*AspectJ*	*Avg.*
sgAttention	**0.710**	**0.772**	**0.783**	**0.845**	**0.778**
- w/o CFG	0.667	0.736	0.738	0.808	0.737
- w/o token	0.641	0.654	0.671	0.773	0.685
- w/o mask	0.687	0.741	0.762	0.826	0.754
- w/o mean	0.691	0.754	0.768	0.830	0.761

Table 5: The ablation study evaluation in terms of MRR.

3.4 The Ablation Study

To evaluate the effectiveness of the sgAttention design, we conduct ablation studies on the CFG encoder, the lexical encoder, the attention masking mechanism, and the aggregation method imposed on the CFG encoder, which are denoted as *w/o CFG*, *w/o token*, *w/o mask*, and *w/o mean*, respectively. The evaluation results in terms of MAP and MRR are shown in Table 4 and Table 5, respectively.

Specifically, *sgAttention w/o CFG* denotes a model that only uses the lexical encoder and the report encoder, which exactly is the CodeBERT model. Compared with sgAttention, the MAP drops by 6.2% to 8.5% and the MRR drops by 4.6% to 6.4%, indicating that the CFG plays an important role in bug localization. By adding a CFG encoder without the attention masking mechanism, the *sgAttention w/o mask* model performs better on all the datasets, and even better than GraphCodeBERT, which means that CFG carries more crucial information than the data flow graph for bug localization. However, compared with sgAttention, it still decreases by 3.1% to 5.3% in terms of MAP and by 2.3% to 4.2% in terms of MRR, indicating that only focusing on relevant nodes in the CFG improves the performance of bug localization.

In addition, *sgAttention w/o lexical* denotes a model that only uses the CFG encoder and the report encoder. It can be observed that compared with sgAttention, the MAP decreases by over 10.4% and the MRR decreases by over 9.3%, indicating that the lexical information is crucial for bug localization.

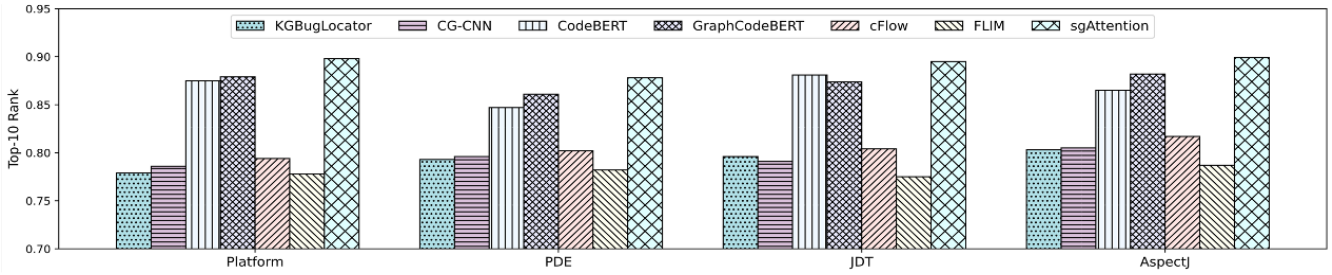

Figure 4: The performance evaluation in terms of Top-10 Rank. sgAttention (red) achieves the best performance (highest) on all the datasets.

At last, *sgAttention w/o mean* denotes a variant of sgAttention that follows the suggestion in Graphormer [Ying *et al.*, 2021] to aggregate node features. Specifically, it adds an additional super-node connected to all nodes in the CFG, and includes the feature of it as the CFG feature. We empirically found that such operations do not perform better than simply including the mean of node features as the CFG feature.

4 Related Work

In this section, we introduce a number of works related to bug localization and some transformer-based models designed for software mining tasks.

Bug localization aims to automatically locate buggy source files based on the textual description in the bug report. A large class is the information-retrieval-based bug localization methods, which locate buggy files based on the lexical similarity between the bug report and the source code [Poshyvanyk *et al.*, 2007; Lukins *et al.*, 2008; Saha *et al.*, 2013; Wang *et al.*, 2018]. For example, BugLocator [Zhou *et al.*, 2012] employs the revised Vector Space Model (rVSM) to measure the lexical similarity between the report and the code. DNNLOC [Lam *et al.*, 2017] further combines the rVSM with the auto-encoder to alleviate the lexical mismatch problem in information retrieval. BLIZZARD [Rahman and Roy, 2018] and FineLocator [Zhang *et al.*, 2019] improve the information retrieval quality with the query reformulation. Fejzer *et al.* [2022] adaptively reweight 19 similarity scores in [Ye *et al.*, 2016] to improve the retrieval quality, which is further incorporated by FLIM [Liang *et al.*, 2022]. However, many recent studies indicate that mining the program structure from the source code is beneficial for improving bug localization [Huo *et al.*, 2016; Youm *et al.*, 2017; Huo and Li, 2017]. CG-CNN [Huo *et al.*, 2020] decomposes the CFG of source code into multiple execution paths for multi-instance learning. KGBugLocator [Zhang *et al.*, 2020] builds a code knowledge graph to model interrelations between classes, parameters, variables, methods, and properties for bug localization. MRAM [Yang *et al.*, 2021] builds the code revision graph from past code commits and reports for method-level bug localization. cFlow [Ma and Li, 2022] designs a flow-based GRU to extract the feature of the CFG by step-wisely feature propagation.

In addition, many methods deal with the bug localization problem from other perspectives. Locus [Wen *et al.*, 2016] and FBL-BERT [Ciborowska and Damevski, 2022] retrieve buggy source files from the code change logs. BRTracer [Wong *et al.*, 2014] improves bug localization with segmentation and stack-trace analysis. Pathidea [Chen *et al.*, 2022] improves bug localization by reconstructing the execution path from the bug report. TRANP-CNN [Huo *et al.*, 2019] and COOBA [Zhu *et al.*, 2020] are designed for the cross-project bug localization with the problem of insufficient history data. The usages of bug localization models in the education [Gupta *et al.*, 2019] and the industrial setting [Jarman *et al.*, 2022] are also explored.

Nowadays, with the success of the Transformer on the natural language processing area, a number of Transformer based models have been proposed for software mining tasks. For example, GREAT [Hellendoorn *et al.*, 2020] stacks the Transformer encoder layers and the graph neural network to extract the feature of the abstract syntax tree for the variable misuse task. TPTrans [Peng *et al.*, 2021] integrates the encoding of the absolute and the relative path in the abstract syntax tree into the Transformer for the code summarization task. MuST [Zhu *et al.*, 2022] is a Transformer-architecture program translation model that leverages multilingual code snippets for training. CodeBERT [Feng *et al.*, 2020] is a pre-trained model that pre-trained on natural language and programming language pairs using both masked language modeling and replaced token detection tasks, and GraphCodeBERT [Guo *et al.*, 2021] further enhances the code representation with the data flow.

5 Conclusion

In this paper, we propose a novel bug localization model named sgAttention, which employs a particularly designed structural-guided attention to capture the long-distance dependency in the CFG. In sgAttention, each node only takes the features of the execution-depended and the computation-depended nodes, and masks the features of irrelevant nodes to facilitate better feature extraction of the CFG. Experimental results based on four widely-used open-source software projects show that sgAttention averagely improves the state-of-the-art bug localization methods by 32.9% and 29.2% and the state-of-the-art pre-trained models by 5.8% and 4.9% in terms of MAP and MRR, respectively. The empirical evaluation indicates that capturing the long-distance dependency in the CFG via the structural-guided attention is beneficial for improving bug localization.

For future work, it is interesting to extract the features of other code graph representations like the abstract syntax tree or the call graph with the structural-guided attention.

Acknowledgments

This research was supported by NSFC (62076121, 61921006).

References

[Chen *et al.*, 2022] An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, 48(8):2905–2919, 2022.

[Ciborowska and Damevski, 2022] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with BERT. In *44th IEEE/ACM International Conference on Software Engineering, Pittsburgh, PA, USA*, pages 946–957. ACM, 2022.

[Devlin *et al.*, 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA*, pages 4171–4186. Association for Computational Linguistics, 2019.

[Fejzer *et al.*, 2022] Mikolaj Fejzer, Jakub Narebski, Piotr Przymus, and Krzysztof Stencel. Tracking buggy files: New efficient adaptive bug localization algorithm. *IEEE Transactions on Software Engineering*, 48(7):2557–2569, 2022.

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

[Fischer *et al.*, 2003] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, pages 23–32, 2003.

[Gay *et al.*, 2009] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *IEEE International Conference on Software Maintenance*, pages 351–360, 2009.

[Guo *et al.*, 2021] 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. GraphCodeBERT: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, Virtual Event*. OpenReview.net, 2021.

[Gupta *et al.*, 2019] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. Neural attribution for semantic bug-localization in student programs. In *Advances in Neural Information Processing Systems 32, Vancouver, BC, Canada*, pages 11861–11871, 2019.

[Hellendoorn *et al.*, 2020] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, Addis Ababa, Ethiopia*. OpenReview.net, 2020.

[Huo and Li, 2017] Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1909–1915, Melbourne, Australia, 2017.

[Huo *et al.*, 2016] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1606–1612, New York, USA, 2016.

[Huo *et al.*, 2019] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *IEEE Transactions on Software Engineering*, 47(7):1368–1380, 2019.

[Huo *et al.*, 2020] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Control flow graph embedding based on multi-instance decomposition for bug localization. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, volume 34, pages 4223–4230, New York, USA, 2020.

[Jarman *et al.*, 2022] Darryl Jarman, Jeffrey Berry, Riley Smith, Ferdian Thung, and David Lo. Legion: Massively composing rankers for improved bug localization at adobe. *IEEE Transactions on Software Engineering*, 48(8):3010–3024, 2022.

[Kochhar *et al.*, 2014] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, page 803–814, Vasteras, Sweden, 2014.

[Lam *et al.*, 2015] An N. Lam, Anh T. Nguyen, Hoan A. Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, NE, USA*, pages 476–481, 2015.

[Lam *et al.*, 2017] An N. Lam, Anh T. Nguyen, Hoan A. Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, Buenos Aires, Argentina*, pages 218–229. IEEE Computer Society, 2017.

[Liang *et al.*, 2022] Hongliang Liang, Dengji Hang, and Xiangyu Li. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering*, 27(7):186, 2022.

[Loshchilov and Hutter, 2019] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, New Orleans, LA, USA*. OpenReview.net, 2019.

[Lukins *et al.*, 2008] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium*, pages 155–164. IEEE Computer Society, 2008.

[Ma and Li, 2022] Yi-Fan Ma and Ming Li. The flowing nature matters: feature learning from the control flow graph of source code for bug localization. *Machine Learning*, 111(3):853–870, 2022.

[Peng *et al.*, 2021] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. Integrating tree path in transformer for code representation. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, virtual*, pages 9343–9354, 2021.

[Poshyvanyk *et al.*, 2007] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[Rahman and Roy, 2018] Mohammad M. Rahman and Chanchal K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 621–632, New York, NY, USA, 2018.

[Saha *et al.*, 2013] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley, CA, USA*, pages 345–355. IEEE, 2013.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 6000–6010, Red Hook, NY, USA, 2017.

[Wang *et al.*, 2018] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Ming Li, Feng Xu, and Jian Lu. Bug localization via supervised topic modeling. In *18th IEEE International Conference on Data Mining, Singapore*, pages 607–616. IEEE Computer Society, 2018.

[Wen *et al.*, 2016] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore*, pages 262–273. ACM, 2016.

[Wong *et al.*, 2014] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada*, pages 181–190. IEEE Computer Society, 2014.

[Yang *et al.*, 2021] Shouliang Yang, Junming Cao, Hushuang Zeng, Beijun Shen, and Hao Zhong. Locating faulty methods with a mixed RNN and attention model. In *29th IEEE/ACM International Conference on Program Comprehension, Madrid, Spain*, pages 207–218. IEEE, 2021.

[Ye *et al.*, 2014] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China*, pages 689–699. ACM, 2014.

[Ye *et al.*, 2016] Xin Ye, Razvan C. Bunescu, and Chang Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering*, 42(4):379–402, 2016.

[Ying *et al.*, 2021] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In *Advances in Neural Information Processing Systems 34, Virtual*, pages 28877–28888, 2021.

[Youm *et al.*, 2017] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177 – 192, 2017.

[Zhang *et al.*, 2019] Wen Zhang, Ziqiang Li, Qing Wang, and Juan Li. Finelocator: A novel approach to method-level fine-grained bug localization by query expansion. *Information Software Technology*, 110:121–135, 2019.

[Zhang *et al.*, 2020] Jing-Lei Zhang, Rui Xie, Wei Ye, Yu-Han Zhang, and Shi-Kun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, page 219–229, New York, NY, USA, 2020.

[Zhou *et al.*, 2012] Jian Zhou, Hong-Yu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering*, pages 14–24. IEEE Computer Society, 2012.

[Zhu *et al.*, 2020] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. COOBA: Cross-project bug localization via adversarial transfer learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 3565–3571, 2020.

[Zhu *et al.*, 2022] Ming Zhu, Karthik Suresh, and Chandan K. Reddy. Multilingual code snippets training for program translation. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, Virtual Event*, pages 11783–11790. AAAI Press, 2022.