Machine Learning (2022) 111:853–870
https://doi.org/10.1007/s10994-021-06078-4

The flowing nature matters: feature learning from the control flow graph of source code for bug localization

Yi-Fan Ma[1] · Ming Li[1]

Received: 15 May 2021 / Revised: 14 August 2021 / Accepted: 22 September 2021 /
Published online: 17 February 2022
© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2022

Abstract
Bug localization plays an important role in software maintenance. Traditional works treat the source code from the lexical perspective, while some recent researches indicate that exploiting the program structure is beneficial for improving bug localization. Control flow graph (CFG) is a widely used graph representation, which essentially represents the program structure. Although using graph neural network for feature learning is a straightforward way and has been proven effective in various software mining problems, this approach is inappropriate since adjacent nodes in the CFG could be totally unrelated in semantics. On the other hand, previous statements may affect the semantics of subsequent statements along the execution path, which we call the *flowing nature* of control flow graph. In this paper, we claim that the *flowing nature* should be explicitly considered and propose a novel model named cFlow for bug localization, which employs a particular designed flow-based GRU for feature learning from the CFG. The flow-based GRU exploits the program structure represented by the CFG to transmit the semantics of statements along the execution path, which reflects the *flowing nature*. Experimental results on widely-used real-world software projects show that cFlow significantly outperforms the state-of-the-art bug localization methods, indicating that exploiting the program structure from the CFG with respect to the *flowing nature* is beneficial for improving bug localization.

Keywords Control flow graph · Bug localization · Flow-based GRU · Software mining

Editor: Yu-Feng Li, Mehmet Gönen, Kee-Eung Kim.

✉ Ming Li
lim@lamda.nju.edu.cn

Yi-Fan Ma
mayf@lamda.nju.edu.cn

[1] Nanjing University, Nanjing, China

1 Introduction

Bug reports are generated when the software fails to behave as it is expected or follow the technical requirements of the system. Unfortunately, it is costly for the developer to manually locate the corresponding buggy source files according to the bug report, especially when the software system is large. To reduce the maintenance cost, bug localization, which aims to automatically locate the buggy source files according to the bug report, has drawn significant attention in software mining community and many models have been proposed (Huo et al. 2020; Poshyvanyk et al. 2007; Ye et al. 2014; Zhou et al. 2012).

Traditional works treat the source code as pure text and locate buggy files by measuring the lexical similarity between the bug report and the source code (Gay et al. 2009; Lukins et al. 2008; Zhou et al. 2012). Recent researches indicate that the program structure of source code carries more semantics reflecting the program behavior, which should be exploited in feature learning and is beneficial for bug localization. Such structures include: correlations among neighboring statements (Huo et al. 2016), long term sequential dependency of source code (Huo and Li 2017), abstract syntax tree (AST) of source code (Youm et al. 2017), etc. However, these structures can only represent part of the program structure. A more essential and widely used representation is the control flow graph (CFG). Recently, Huo et al. (2020) propose a CG-CNN model to exploit more complex program structures such as branches and loops from the CFG for improving bug localization. However, CG-CNN decomposes the CFG into multiple paths and merges all the representations with average-pooling, which breaks the integrity of the graph and fails to consider the inherent correlation among paths. Therefore, to comprehensively exploit the program structure and consider the correlation among execution paths, the integrity of CFG should not be broken.

A straightforward way for feature learning from the entire CFG is to use the graph neural network (GNN). Although GNN has been widely used in various software mining problems and proven to be effective in embedding the AST (Allamanis et al. 2018; Mou et al. 2016; Wei and Li 2017; Zhou et al. 2019), it is inappropriate to embed the CFG with GNN. In general, GNN models assume that two adjacent nodes in the graph are related (Grover and Leskovec 2016; Kipf and Welling 2017; Niepert et al. 2016; Veličković et al. 2018), which means that the learned features between adjacent nodes should be closer than non-adjacent nodes. GNN performs well in embedding the AST, since two adjacent nodes in the AST are inherently semantically related and their learned semantic features ought to be similar.

However, this assumption no longer holds in the CFG. Edges in the CFG only represent the successively execution relationship, and two adjacent nodes in the CFG could be unrelated in semantics. Instead, previous statements may affect the semantics of subsequent statements along the execution path, which we call the *flowing nature* of CFG. Figure 1 illustrates an example of the source code and the corresponding CFG, where each statement in the source code corresponds to one node in the CFG. It can be observed that two adjacent statements (node 1 and 2) are unrelated in semantics. On the other hand, although node 6 is far away from node 1 and 4, the semantics of node 6 is jointly determined by them. In general, a statement p could affect the semantic of another statement q only if there exists an execution path (i.e., a *walk* in the CFG), where p is executed before q. Therefore, instead of aggregating semantics from neighbors like GNN, the semantics of statements should *flow* in a directional manner from the entry node to the exit nodes along execution paths.

In this paper, we claim that the *flowing nature* of CFG should be explicitly considered in feature learning and propose a novel model named cFlow for bug localization. In cFlow, a

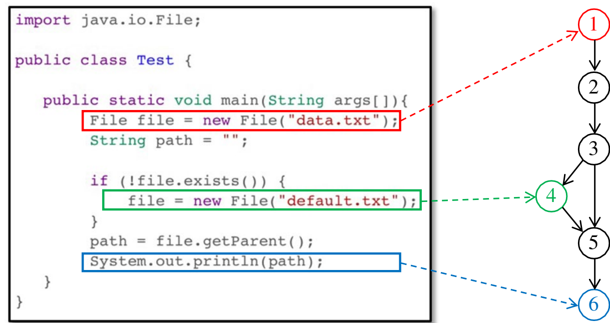

Fig. 1 An example of source code snippet (left) and the corresponding CFG (right). Node 1 and 2 are adjacent but semantically-unrelated. Node 1 and 4 together determine the semantics of Node 6, even if they are far away

flow-based GRU is particularly designed to transmit the semantics of statements along the execution paths, where the program structure represented by the CFG is fully exploited. In order to further consider the inherent correlation among different paths, our flow-based GRU merges paths when they converge on the same statement. Experimental results on widely-used real-world software projects show that cFlow significantly outperforms the state-of-the-art bug localization methods, indicating that exploiting the program structure represented by the CFG with respect to the *flowing nature* is beneficial for improving the bug localization.

The contributions of our work are summarized as follows:

- We claim that the control flow graph holds the unique *flowing nature*, that is, previous statements may affect the semantics of subsequent statements along the execution path, while the semantics of adjacent nodes may be irrelevant. The *flowing nature* is essential and should be explicitly considered in feature learning.
- We propose a novel model named cFlow for bug localization, where a particularly designed flow-based GRU is employed for feature learning from the CFG. The design of our flow-based GRU explicitly considers the *flowing nature* and the inherent correlations among paths, where the semantics of statements are transmitted along the execution paths and paths are merged when they converge on the same statement, respectively.

The rest of this paper is organized as follows. In Sect. 2, the proposed cFlow model will be introduced in detail. Experiments will be provided and discussed in Sect. 3 and the related works will be introduced in Sect. 4. In Sect. 5, this paper is concluded and some future works are discussed.

2 The proposed method

The goal of bug localization is to locate potentially buggy source files that produce the program behaviors specified in the given bug report. Here comes our formulation of the bug localization problem:

Let $\mathcal{R} = \{r_1, r_2, \dots, r_{N_r}\}$ denotes the set of bug reports received by the developer and $\mathcal{C} = \{c_1, c_2, \dots, c_{N_c}\}$ denotes the set of source code files of a software project, where N_r, N_c denotes the number of bug reports and source code files, respectively. The learning task of

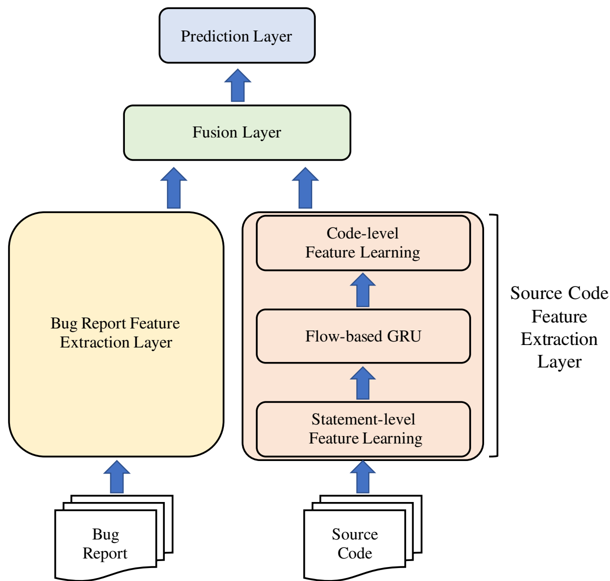

Fig. 2 The general framework of cFlow. The semantic features of bug report and source code are extracted by two separate layers since they are heterogeneous. After that, two consequent layers are designed to fuse them into a unified feature and make the final prediction, respectively

bug localization aims to learn a prediction function $f : \mathcal{R} \times \mathcal{C} \mapsto \mathcal{Y}$. Let $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a source code file $c_j \in \mathcal{C}$ is related to a bug report $r_i \in \mathcal{R}$, which can be obtained by investigating software commit logs and bug report descriptions. The prediction function f can be learned 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 and $\Omega(f)$ is a regularization term imposing on the prediction function. The trade-off between $\mathcal{L}(\cdot, \cdot)$ and $\Omega(f)$ is balanced by a hyper-parameter λ.

2.1 The general framework of cFlow

The learning task of bug localization is instantiated by proposing a novel model called cFlow. cFlow is consisted of four layers: source code feature extraction layer, bug report feature extraction layer, fusion layer and prediction layer. We design two independent layers to extract features from bug reports and source files separately, since they are heterogeneous. The general framework of cFlow is shown in Fig. 2.

The bug report feature extraction layer takes the bug report r_i as input and extracts the semantic feature $\mathbf{u}_i^r$ of it:

$$\mathbf{u}_i^r = f_{report}(r_i). \tag{2}$$

Since bug reports are written in natural language, we follow the normal natural language preprocessing method. We first tokenizes the words in the bug report and removes the stop

words, then each token term is embedded with word2vec Mikolov et al. (2013). Following with Kim (2014), we use 1D-CNN with max-pooling to extract the semantic feature of the bug report.

The source code feature extraction layer aims to take the raw data of source code c_j as input and extracts the semantic feature $\mathbf{u}_j^c$ of it, where the program structure is carefully exploited:

$$\mathbf{u}_j^c = f_{code}(c_j). \tag{3}$$

This layer can be further divided into three sub-layers. The first sub-layer is designed for the initial statement-level feature learning. In the second sub-layer, we design a flow-based GRU to enhance the statement-level feature, where the program structure represented by the CFG is exploited to transmit the semantics of statements along the execution path. The third sub-layer merges all the enhanced statement-level features into the code-level semantic feature $\mathbf{u}_j^c$. This layer is the core of cFlow and the details will be introduced in subsection 2.2.

In **the fusion layer**, source code feature $\mathbf{u}_j^c$ and bug report feature $\mathbf{u}_i^r$ are taken as input and they will be fused into a unified feature for prediction. We employ a fully connected network to fuse these two language-specific features. Finally, in **the prediction layer**, fully connected network is employed to predict whether the source code c_j is related to bug report r_i based on the unified feature:

$$\hat{y}_{ij} = f_{prediction}(f_{fusion}(\mathbf{u}_i^r, \mathbf{u}_j^c)). \tag{4}$$

The empirical loss function used in cFlow is cross entropy, and L_2 regularization is employed to avoid overfitting.

In most cases of bug localization, a reported bug may only be related to one or a few source code files, while a large number of source code files are irrelevant. This highly imbalance nature should be carefully considered. Following with the previous bug localization work (Huo et al. 2016), several negative instances are randomly dropped, which can decrease the computational cost and counteract the imbalance nature. Specifically, we randomly select a subset of all negative instances for training, and discard the rest. How many negative instances will be selected is a hyper-parameter in cFlow. In our implementation, the number of negative instances we randomly select is the same as the number of positive instances, aiming to result a relatively balanced positive and negative distribution.

2.2 The source code feature extraction layer

The source code feature extraction layer takes the raw data of source code c_j as input, and aims to learn the semantic feature $\mathbf{u}_j^c$ of it. For the simplicity of notations, we will omit the subscript j in this subsection if there is no ambiguity.

As shown in Fig. 1, each statement in the program corresponds to a node in the CFG. Let $G = (V, E, \mathbf{X})$ indicate the corresponding CFG of source code c. The matrix $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ denotes the statement(node)-level feature matrix, where each node $v_l \in V$ is represented by a d-dimensional real-valued vector $\mathbf{x}_l \in \mathbb{R}^d$. Obviously, G is a directed graph.

The first sub-layer is designed to extract the initial statement-level feature $\mathbf{X}^0$ from the raw text data of each statement. We tokenize each statement, use the camel case nomenclature to split each token, and remove the unimportant punctuation such as braces, commas and quotation marks. Then, word2vec is used to embed each token

term. After that, a 1D-CNN with d filters is employed to incorporate the semantic meaning of tokens, followed by max-pooling. The filters should slide within each statement and stop when they meet the end of the statement, respecting the atomicity of the statement in semantics explicitly. In this sub-layer, we focus on the semantic meaning of a single statement, thus all the statements are processed individually.

After extracting the initial statement-level feature $\mathbf{X}^0$, the second sub-layer further exploits the program structure to enhance the satement-level feature. To explicitly consider the *flowing nature* of CFG, we particularly design a flow-based GRU to transmit the semantics of statements from the entry node to the exit nodes along the execution paths. In the rest part of this subsection, we assume that G is *connected*, which means there is only one entry node v_{entry}, and v_{entry} can reach all the other nodes in the CFG. Otherwise, the flow-based GRU will process each connected component independently. Here comes three definitions:

Definition 1 *(InNode)* For an arbitrary node $p \in V$, the set $InNode(p)$ is defined as $InNode(p) = \{q|(q, p) \in E\}$.

Definition 2 *(OutNode)* For an arbitrary node $p \in V$, the set $OutNode(p)$ is defined as $OutNode(p) = \{q|(p, q) \in E\}$.

Definition 3 *(ReachableFrom)* For an arbitrary Node $p \in V$, the set $ReachableFrom(p)$ is defined as:

- $q \in InNode(p) \Rightarrow q \in ReachableFrom(p)$,
- $r \in ReachableFrom(q) \wedge q \in ReachableFrom(p) \Rightarrow r \in ReachableFrom(p)$.

For an arbitrary node p, any node $q \in ReachableFrom(p)$ indicates that there is an execution path (i.e., a *walk* in the CFG) in the source code, and the correponding statement q is executed before statement p. So the statement q may affect the semantics of the statement p. Therefore, when enhancing the feature of the node p, the semantic information of all the nodes in $ReachableFrom(p)$ should be taken into consideration.

The information transmission process of flow-based GRU is inspired by the Breadth First Search (BFS) algorithm. The process has T time steps in total, where T is a hyperparameter and will be discussed later. Let $\mathbf{z}_l^t$ denotes the hidden state of node v_l, and $Act^t \subseteq V$ denotes the *activated* nodes at time step t. At the beginning, only the entry node v_{entry} is *activated,* and all the hidden states $\mathbf{z}_l^0$ are initialized as $\mathbf{0}$. For each time step t, each *activated* node p aggregates non-zero hidden states from all its *InNode(p)* (Eq. 5) and updates its node feature $\mathbf{x}_p^t$ and hidden state $\mathbf{z}_p^t$ with GRU (Eq. 6). Inactivated nodes will remain their node features and hidden states the same as the last time step (Eq. 6).

$$\widetilde{\mathbf{z}}_p^t = AVG\left(\left\{\mathbf{z}_q^{t-1} | q \in InNode(p) \wedge \mathbf{z}_q^{t-1} \neq \mathbf{0}\right\}\right). \tag{5}$$

$$\mathbf{x}_p^t, \mathbf{z}_p^t = \begin{cases} GRU\left(\mathbf{x}_p^{t-1}, \widetilde{\mathbf{z}}_p^t\right), & p \in Act^t, \\ \mathbf{x}_p^{t-1}, \mathbf{z}_p^{t-1}, & otherwise. \end{cases} \tag{6}$$

The *OutNode* of *activated* nodes will become *activated* nodes at the next time step:

$$Act^t = \begin{cases} \{v_{entry}\}, & t = 1, \\ \bigcup_{p \in Act^{t-1}} OutNode(p), & otherwise. \end{cases} \tag{7}$$

Intuitively, each hidden state indicates an execution path starting from the entry node. It is worth noticing that as long as one *InNode* is activated at the last time step (i.e., a new execution path comes), all the non-zero hidden states from *InNodes* (whether they are activated or not at the last time step) will be aggregated to update the node feature and hidden state. If a node has never been activated, the hidden state will remain 0, which will not affect the update procedure. The advantage of doing so is that we no longer need to confirm how long each execution path is and all the possible execution paths will be aggregated simultaneously when the last never-been-activated *InNode* is activated (i.e., the longest path reaches). Therefore, cFlow can comprehensively consider the inherent correlations among different execution paths.

Figure 3 illustrates an example of how the flow-based GRU exploits the program structure to enhance the statement-level feature. The upper left is the source code and the upper right is the corresponding CFG, where each statement corresponds to a node and directed edges represent that two statement may be executed consequently. Only node 3 is activated at $t = 3$, thus hidden state of $InNode(3)$ is aggregated to update $\mathbf{x}_3^3$ and $\mathbf{z}_3^3$ according Eqs. 5 and 6, and other nodes remain their features and hidden states. At $t = 4$, the $OutNodes(3)$ (i.e., node 4 and 5) become activated nodes and update their features and hidden states. However, node 4 has never been activated before $t = 4$, thus the hidden state is $\mathbf{0}$ and node 5 only aggregates the hidden state from node 3. At $t = 5$, node 5 is still activated since it is the *OutNode* of node 4 and node 4 is activated at $t = 4$. At this moment, node 5 is able to aggregate hidden states from all the $InNode(5)$, which means two different execution paths (1, 2, 3, 5) and (1, 2, 3, 4, 5) will be combined here and the correlation between them will be considered.

For each time step, the flow-based GRU only steps forward one node along the execution path. Therefore, the maximum time steps T determines how far the semantic information will be transmitted. In order to ensure that each node is able to receive semantic information from all its *ReachableFrom* nodes, an upper bound of T is provided in Proposition 1.

Proposition 1 *Given the control flow graph* $G = (V, E, X)$ *and the entry node* s*. For* $\forall p, q \in V$*, if* $p \in ReachableFrom(q)$*, there is a walk* $|(s = v_1, v_2, \dots, v_{k_1} = p, v_{k_1+1}, \dots, v_{k_1+k_2} = q)| \leq 2|V|$.

Proof To prove this proposition, we only need to prove that there is a walk $|(s = v_1, v_2, \dots, v_{k_1} = p)| \leq |V|$ and a walk $|(p = v_1, v_2, \dots, v_{k_2} = q)| \leq |V|$. Since G is connected and s is the entry node, it is obviously that $\forall v \in V \backslash \{s\}, s \in ReachableFrom(v)$. Otherwise, the statement will never be executed and can be ignored. Thus, we only need to prove that $\forall v \in V, w \in ReachableFrom(v)$ indicates that there exists a walk $|(v = v_1, v_1, \dots, v_k = w)| \leq |V|$. It is trivial by Definition 3. □

Proposition 1 gives a theoretical proof that any semantic information from *ReachableFrom(p)* takes at most 2|*V*| time steps to be transmitted to the node p, where |*V*| is the number of nodes in the CFG. In other words, when T is large

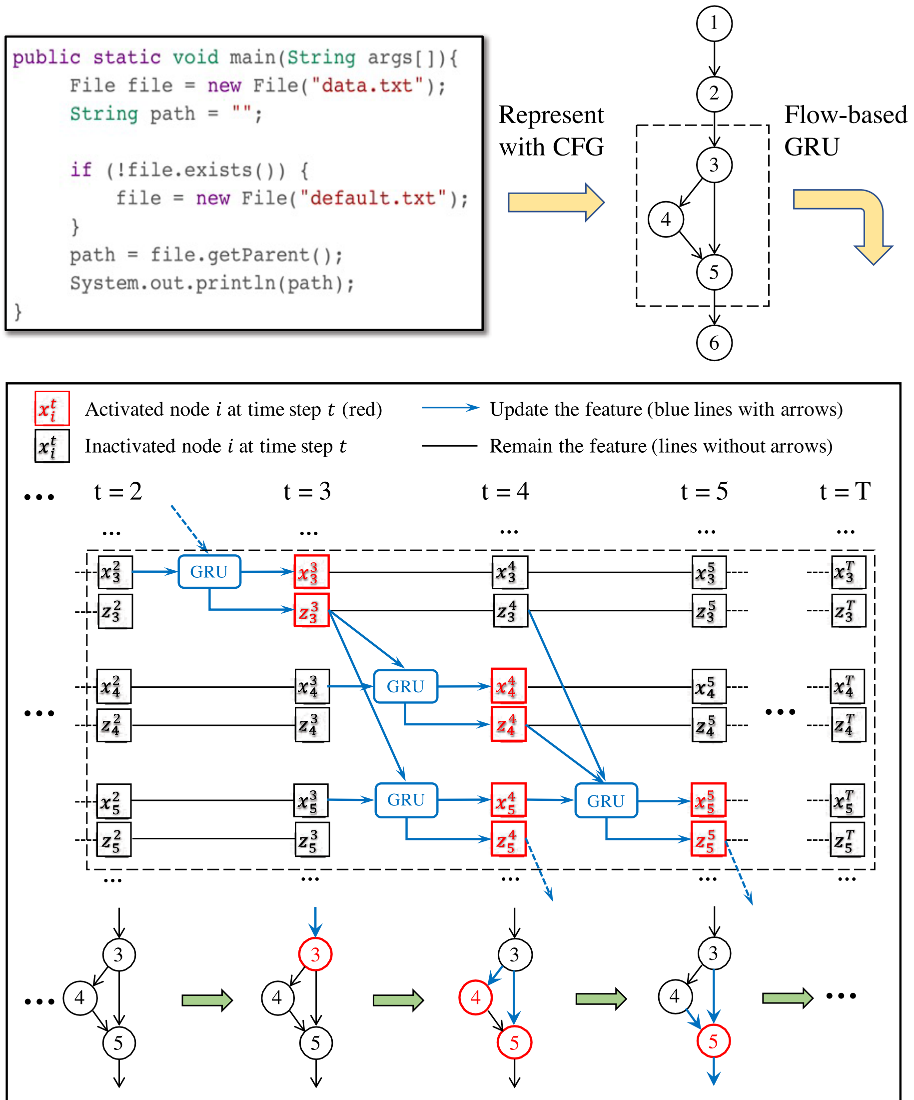

Fig. 3 An example of flow-based GRU. For each time step, activated nodes (red) aggregate hidden states from *InNode* and update node features and hidden states with GRU (lines with arrows). Their *OutNodes* will become activated at the next time step. Inactivated nodes remain node features and hidden states (lines without arrows). It is worth noticing that the hidden state of never been activated nodes (e.g., $\mathbf{z}_4^3$) will not be aggregated (Color figure online)

enough, flow-based GRU ensures that each node can receive semantic information from all the *ReachableFrom* nodes.

We further provide the pseudo code of flow-based GRU in Algorithm 1. Flow-based GRU takes the CFG, the initial node features and the total time step as input. Line 1-2 are the initial part, and the update equations for activated and inactivated nodes are listed in line 6-7 and line 9-10, respectively. Line 13-17 merge the *OutNodes* of all the activated

nodes to generate the activated nodes at the next time step. In the end, flow-based GRU outputs the enhanced node features.

After enhancing the statement-level feature in the second sub-layer, the third sub-layer is designed to merge all the statement-level features into the code-level feature. In order to preserve the original semantics of statements, we concentrate the initial statement-level features $\mathbf{X}^0$ and the enhanced statement-level features $\mathbf{X}^T$. Then, a normal GRU with average-pooling is employed to extract the code-level feature:

Algorithm 1 Flow-based GRU

Input: $G = (V, E)$: the control flow graph; $\mathbf{X}^0 \in \mathbb{R}^{|V|\times d}$: the initial node feature matrix; T: the total time step
Output: $\mathbf{X}^T$: the enhanced node feature matrix

```
1: Initialize the hidden states Z^0 with 0.
2: Initialize the activated nodes array Act = {s} , where s is the entry node.
3: for t = 1 to T do
4:    for each node p ∈ V do
5:       if p ∈ Act then
6:          z̃_p^t = AVG({z_q^{t-1} | q ∈ InNode(p) ∧ z_q^{t-1} ≠ 0})
7:          x_p^t, z_p^t = GRU(x_p^{t-1}, z̃_p^t)
8:       else
9:          x_p^t = x_p^{t-1}
10:         z_p^t = z_p^{t-1}
11:      end if
12:   end for
13:   NextAct = {}
14:   for Node p ∈ Act do
15:      NextAct = NextAct ∪ OutNode(p)
16:   end for
17:   Act = NextAct
18: end for
19: return X^T = (x_1^T, x_2^T, ..., x_{|V|}^T)
```

$$\mathbf{u}^c = \frac{1}{|V|} \sum_{i=1}^{|V|} GRU([\mathbf{x}_i^T;\mathbf{x}_i^0]). \tag{8}$$

After being processed by the source code feature extraction layer, the extracted source code feature $\mathbf{u}^c$ will be fed into the cross-language fusion layer together with the bug report feature $\mathbf{u}^r$ for further fusion and prediction (mentioned in subsection 2.1).

3 Experiments

We have conducted comparative evaluation of cFlow against state-of-the-art bug localization methods, based on four widely-used real-world software projects. This section includes the summary of our experimental setup, the evaluation result, and a brief analysis of the result.

3.1 Data sets

The data sets used in the experiment are extracted from four real-world software projects. All these projects are open source and ground truth correlations between bug reports and

Table 1 The statistic information of the four real-world data sets

Project	# source files	# bug reports	# total matches
Platform	6125	5016	18055
PDE	5330	2612	10721
JDT	10845	5060	14408
AspectJ	6908	368	1522

source code files can be extracted from bug tracking system (Bugzilla) and version control system (Git) (Fischer et al. 2003).

Eclipse Patform[1] defines the set of frameworks and common services that make up Eclipse infrastructures. The first data set *Platform* is the "UI" component of it. Plug-in Development Environment[2] is a tool to create and deploy Eclipse plug-ins. We use the "UI" component as the second data set, which is denoted as *PDE*. Java Development Tools[3] is an Eclipse project for plug-ins support and development of any Java applications. The third data set *JDT* is the "UI" component of it. The AspectJ[4] project is an aspect-oriented extension to the Java programming language, which is the last data set and denoted as *AspectJ*.

The statistics of the four data sets are shown in Table 1. All of them have been widely used in many previous bug localization studies (Huo et al. 2016; Lam et al. 2015; Zhou et al. 2012). Specifically, as suggested in Kochhar et al. (2014), we filtered those *fully localized* bug reports from the data set, that is, the names of all corresponding buggy source files have already been contained in the bug report. For such bug reports, they no longer need a machine learning model to automatically locate the buggy source files.

3.2 Baseline methods and experiment settings

To evaluate the effectiveness of cFlow, we compare against the following state-of-the-art bug localization methods:

- *Buglocator (Zhou et al.* 2012*)* A classicial information retrieval (IR) based bug localization method, which employs a revised vector space model (rVSM) to represent the bug report and the source code into vectors from the lexical perspective, and identifies potential buggy files by measuring the lexical similarity between bug reports and source files.
- *LS-CNN (Huo and Li* 2017*)* A state-of-the-art deep learning based bug localization method, which only considers the sequential nature of source code. LS-CNN employs a LSTM network to enhance the statement-level feature without considering more complex program structures like branches and loops. The network structure of LS-CNN is equivalent to cFlow without the flow-based GRU.
- *CG-CNN (Huo et al.* 2020*)* A state-of-the-art deep learning based bug localization method, which learns the semantic features from the CFG of the source code. CG-CNN first exploits the structrual information from the CFG to enhance the statement-level

[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/index.php

Table 2 MAP of the compared methods on all data sets

Method	Platform	PDE	JDT	AspectJ	Avg.
Buglocator	0.337	0.352	0.279	0.221	0.297
LS-CNN	0.413	0.439	0.403	0.487	0.436
CG-CNN	0.433	0.476	0.448	0.507	0.466
cFlow-GAT	0.426	0.456	0.415	0.464	0.440
cFlow	**0.459**	**0.489**	**0.468**	**0.529**	**0.486**

The best performance on each data set is boldfaced

Table 3 MRR of the compared methods on all data sets

Method	Platform	PDE	JDT	AspectJ	Avg.
Buglocator	0.394	0.415	0.331	0.268	0.352
LS-CNN	0.501	0.551	0.492	0.601	0.536
CG-CNN	0.526	0.593	0.549	0.617	0.571
cFlow-GAT	0.509	0.574	0.503	0.573	0.540
cFlow	**0.552**	**0.616**	**0.570**	**0.637**	**0.594**

The best performance on each data set is boldfaced

feature with DeepWalk (Perozzi et al. 2014) model. After that, CG-CNN decomposes the CFG into different execution paths and averages all the representations without considering the inherent correlations between different paths.
- *cFlow-GAT* A variant of cFlow, which utilizes a three-layer graph attention network (Veličković et al. 2018) to directly embed the CFG without considering the flowing nature.

To compare with these baselines, we follow the same hyper-parameter settings suggested in their studies. For hyper-parameters in cFlow, the window sizes of the convolutional filters are set as 3,4,5 (with 100 filters of each size). Batch normalization is applied after the fusion layer to prevent over-fitting. Adam (Kingma and Ba 2014) is employed to optimize parameters in cFlow. For each training, we first divide the training data into the training set and the validation set with the ratio of 8:1. Then, all the training hyper-parameters are determined in terms of the performance on the validation set, such as batch size, total time steps T, number of epochs. After that, the model will be retrained using all the training data and the best hyper-parameters.

3.3 Performance evaluation

We consider four evaluation metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), AUC and Top-k Rank, which have been widely-used in previous bug localization studies (Huo and Li 2017; Huo et al. 2020; Zhou et al. 2012). For each data set, 10-fold cross validation is used and the average performances are reported.

The evaluation result in terms of MAP and MRR are listed in Tables 2 and 3, respectively. The performance on each data set is boldfaced only if the model outperforms other

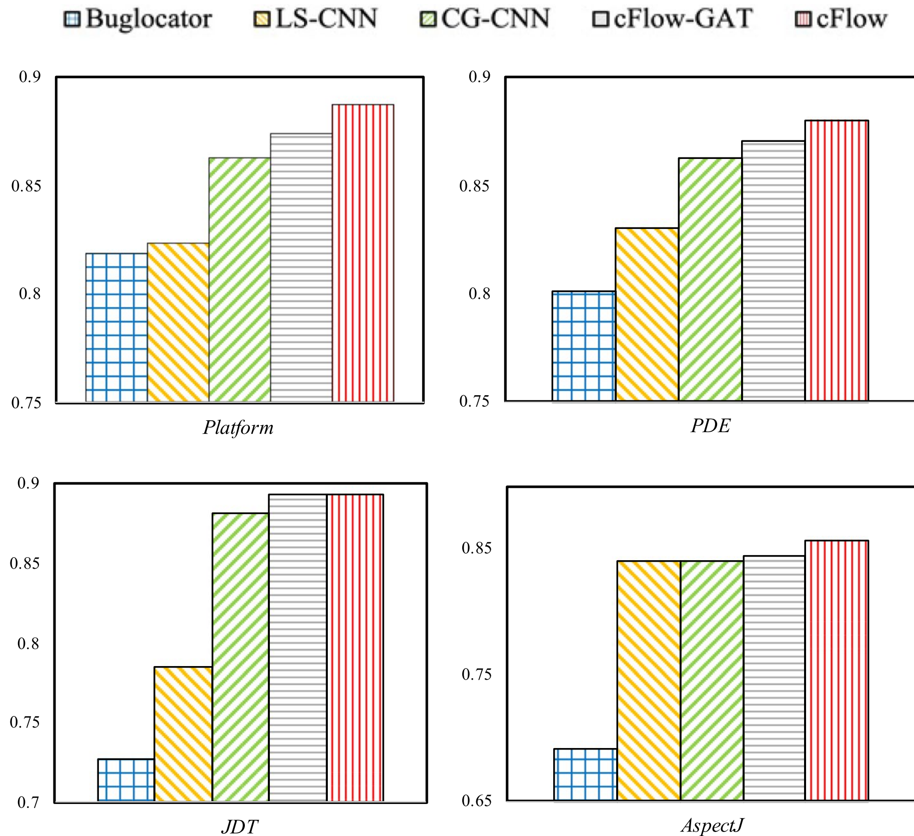

Fig. 4 AUC of the compared methods on all the data sets. cFlow achieves the best performance among all the baselines on all the data sets

baselines on each fold. It can be observed from Table 2 that cFlow achieves the best performance among all the baselines on all the four data sets in terms of MAP. Besides, cFlow achieves the best average MAP performance (0.486), which improves Buglocator (0.297) by 63.6%, LS-CNN (0.436) by 11.6%, CG-CNN (0.466) by 4.3%, cFlow-GAT (0.440) by 10.4%. The superiority of cFlow is significant.

Similar to the performance in terms of MAP, cFlow achieves the best performance among all the baselines on all the four data sets in terms of MRR. Also, cFlow achieves the best average MRR performance (0.594), which improves Buglocator (0.352) by 68.7%, LS-CNN (0.536) by 10.7%, CG-CNN (0.571) by 3.9%, cFlow-GAT (0.540) by 10.0%. This performance evaluation once again reflects the superiority of cFlow.

The performance evaluations in terms of AUC and Top-10 Rank are shown in Figs. 4 and 5, respectively. It can be noticed that cFlow outperforms all the baselines on all the data sets in terms of both AUC and Top-10 Rank. Higher AUC values mean that cFlow is able to rank the related buggy source files higher than other baseline methods, and higher Top-10 Rank values mean that cFlow is able to correlate the most buggy source files when the same number of potential files are examined.

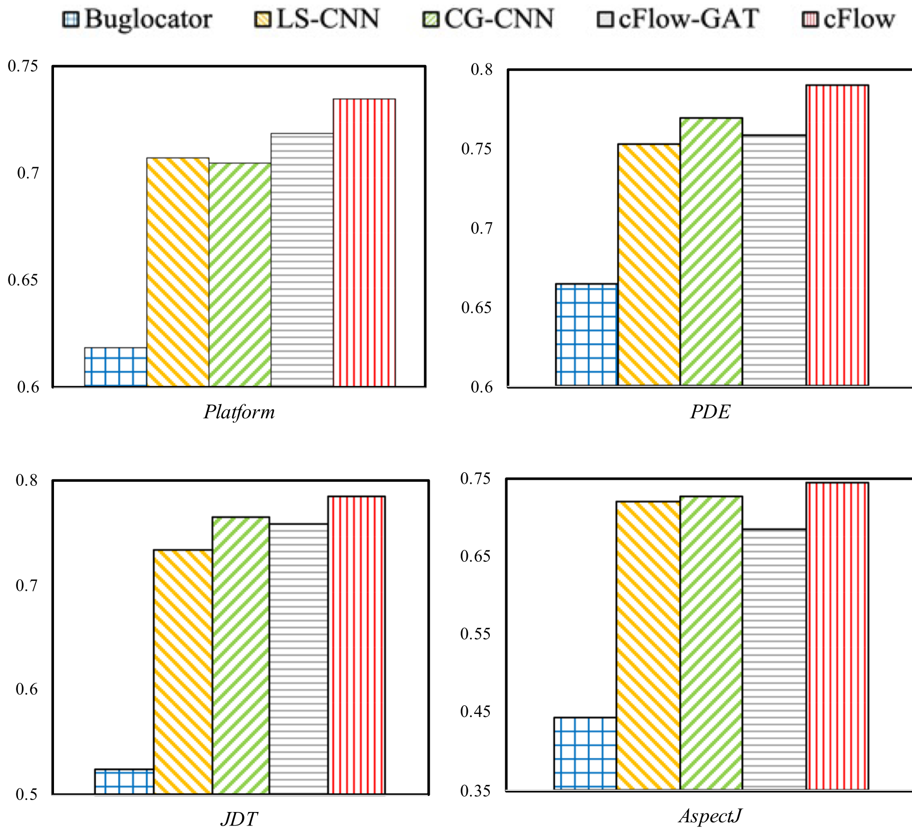

Fig. 5 Top-10 Rank of the compared methods on all the data sets. Higher metric value means better performance. It can be easily observed that cFlow outperforms all the baseline on all the data sets

Compared with the state-of-the-art bug localization models, Buglocator totally ignores the program structure, and LS-CNN only considers the long term sequential dependency. However, cFlow performs better since it exploits more complex program structures like branches and loops from the CFG of source code, indicating that exploiting the program structure improves the performance of bug localization. Although both cFlow-GAT and CG-CNN exploit the program structure from the CFG for feature learning, cFlow-GAT directly uses a GNN which ignores the *flowing nature* of CFG, and CG-CNN ignores the inherent correlation among different execution paths, resulting in only acceptable performances. Experimental results indicate that the *flowing nature* and the correlation among different execution paths are critical to the performance of bug localization.

Figure 6 provides a case study of the bug report and the corresponding source code. Statements that have been modified to fix the bug are in the red boxes. It can be observed from the source code that the variable “items” plays an important role in this bug, and statements containing the variable “items” are far away in the CFG. Interestingly, only cFlow can successfully locate the buggy file, since cFlow explicitly considers the *flowing nature* in feature learning. On the other hand, other baseline models failed to discover the impact of the variable “item” from these distant statements. Therefore, this case study once

Bug report: *Platform-92835*

Importance: P1 critical

Summary: [Viewers] retrieving projects from CVS creates duplicate resources in Navigator.

Description: Observed in I0420 and I0426, did not happen in M6 happens on solaris-motif, aix, and hpux, but NOT linux-motif or others. I could have the wrong component here, but I guessed UI since WorkbenchContentProvider seems to ultimately populate the Navigator view.

- start a new eclipse
- show the CVS Repositories view
- connect to dev.eclipse.org
- start retrieving the org.eclipse.swt and org.eclipse.swt.motif.linux.x86 projects
- click "Run In Background" and switch to the Navigator view
- the two projects will appear as they're retrieved from CVS (fine), but then at the end of Building there will be two duplicate projects added to the Navigator

My initial thought was that this must be a motif Tree bug, but I don't think it is because:
-> the problem does not happen on linux-motif
-> I inserted several println's in Tree and TreeItem apis, ran through the scenario, and noted that twice as many TreeItem creation apis are being invoked in the unix cases compared to linux-motif, so there seems to be some platform differentiation happening outside of swt

Source code: *AbstractTreeViewer.java*

```
private void createAddedElements(Widget widget, Object[] elements) {

        if(elements.length == 1){
                if (equals(elements[0], widget.getData()))
                        return;
        }

        ViewerSorter sorter = getSorter ();
        Item[] items = null;

if (sorter == null) {
           items = getChildren(widget);
          for (int i = 0; i < elements.length; i++) {
                for (int j = 0; j < items.length; j++) {
      if (items[j].getData().equals(elements[i])) {
        //refresh the element in case it has new children
        refresh(elements[i]);
         return;
      }
                }
          }
}


        //As the items are sorted already we optimize for a
        //start position
        int lastInsertion = 0;
        if(items == null)//Don't ask if we already have it
                items = getChildren(widget);

        for (int i = 0; i < elements.length; i++) {
                Object element = elements[i];
                int index;
                if(sorter == null)
                        index = -1;
                else{
                        lastInsertion = insertionPosition(items,sorter,lastInsertion, element);
                        //As we are only searching the original array we keep track of those positions only
                        index = lastInsertion + i; //Add the index as the array is growing
                        // Assume sorter is consistent with equals() - therefore we can
                        // just check against the item prior to this index (if any)
                        if (index >0 && index < items.length && items[index].getData().equals(element)) {
                                //refresh the element in case it has new children
                                refresh(element);
                                break;
                        }
                        if(index == items.length)
                                index = -1;
                }
                createTreeItem(widget, element, index);
        }
}
```

Fig. 6 An example of the bug report and the corresponding buggy file. Red boxes gives those statements that have been modified to fix the bug

again shows that explicitly considering the *flowing nature* is important in feature learning from the CFG for bug localization.

In summary, experimental results on widely-used real-world data sets indicate that cFlow outperforms the state-of-the-art bug localization methods and cFlow-GAT (a variant of cFlow that directly embeds the CFG with GNN) in terms of all the four commonly used metrics, demonstrating that exploiting the program structure from the CFG with respect to the *flowing nature* is beneficial for improving the performance of bug localization.

3.4 Ablation study

We have conducted the ablation study to show that our design for cFlow is effective. The key part of cFlow is the flow-based GRU (i.e., the second sub-layer of the source code feature extraction layer), which is designed to enhance the original statement-level feature by exploiting the program structure represented by the CFG.

In order to show that the enhanced statement-level feature is beneficial for bug localization, we compare cFlow with LS-CNN. cFlow shares a similar network structure with LS-CNN, except that LS-CNN directly merges the inital statement-level feature without employing the flow-based GRU to enhance it. Experimental results in Tables 2, 3, Figs. 4 and 5 show that cFlow outperforms LS-CNN in terms of all the metrics (MAP, MRR, AUC and Top-10 Rank, respectively) on all the four data sets, indicating that employing the flow-based GRU is beneficial and the enhanced statement-level feature is effective for improving bug localization.

4 Related work

In this section, we summarize the literature related to bug localization and deep software mining models.

4.1 Bug localization

Bug localization aims to automatically locate buggy source files according to the textual description in the bug report, which remains a big challenge in software maintenance. Traditional methods treat the source code as pure text and locate potential buggy source files by measuring the lexical similarity between the bug report and the source code. Lukins et al. (2008) use a Latent Dirichlet Allocation (LDA) model to represent source code and bug reports and locate potential buggy files by measuring their similarities. Zhou et al. (2012) propose a revised Vector Space Model (rVSM), where buggy files corresponding to similar historical bug reports are explored to improve the bug localization result. However, as pointed out by Huo et al. (2016), these traditional approaches ignore the rich program structure in source code. To overcome this, Huo et al. (2016) propose a NP-CNN model to generate high-level semantics by exploiting the local relationships among statements with the 1D-CNN. Huo and Li (2017) further exploit the long term sequential dependency of source code with the LSTM network. Youm et al. (2017) exploit the abstract syntax tree to classify the source code tokens into different categories for further analyzing the method information. However, this work only utilizes the token attribution to build the vector space model and does not fully exploit the tree structure. Recently, Huo et al. (2020) exploit more complex structures such as branches and loops from the CFG of source code, in which DeepWalk is employed to learn the representation of each statement, and then the source code is decomposed into different paths for multi-instance learning. Despite exploiting the structural information in the source code, some deep learning methods deal with the bug localization problem from other perspectives. Lam et al. (2017) combine an auto-encoder model with revised vector space model to deal with the lexical mismatch problem in traditional IR-based approaches. Rahman and Roy (2018) classify bug reports into three categories according to the quality and incorporates context-aware query reformulation for bug localization. Huo et al. (2019) propose the TRANP-CNN model for cross-project bug localization to face the problem of insufficient history data. Zhang et al. (2020) propose a KGBuglocator model, which exploits the interrelation information via code knowledge graph for bug localization.

4.2 Deep software mining models

In recent years, deep learning models are very popular and have achieved great success in various software mining tasks. White et al. (2015) first identify avenues to use deep learning models for mining software repositories. Li et al. (2019) employan attention-based network to learn the context-based code representation for improving bug detection. Shi et al. (2019) propose a specific network for code review, where an auto-encoder is employed to learn the revision feature from the original-new source code pair. Alon et al. (2019) design an attention-based neural network to represent arbitrary-sized code snippets into continuous distributed vectors. Zhang and Li (2019) exploit the contest between the plagiarist and the detector for software clone detection. Feng et al. (2020) propose a bimodal pre-trained

model CodeBERT, which achieves state-of-the-art performance on code search and code summarization.

Among all the deep learning models, mining from the source code graph representation has received extensive attention, especially in mining from the abstract syntax tree (AST). Mou et al. (2016) propose a tree-based CNN (TBCNN) model to learn the feature from the AST representation of source code for code functionality classification. Wei and Li (2017) propose a CDLH model for code clone detection, where a AST-based LSTM model is particularly employed for embedding the AST representation of source code. Allamanis et al. (2018) utilize ASTs with additional data flow edges to represent the program for variable-naming and variable-misuse problems and the feature representation of source code is learned with a gated graph neural network (GGNN) (Li et al. 2015). Zhou et al. (2019) propose a Devign model for vulnerability identification, where the source code is represented with a mixture graph containing AST edges, DFG edges, CFG edges and natural sequence edges and GGNN is applied to learn the node representation. Li et al. (2020) propose a DLFix model for automated program repair, where a tree-based RNN encoder-decoder model is employed to learn local contexts.

5 Conclusion and future work

In this paper, we claim that the *flowing nature* of control flow graph is essential and should be explicitly considered, and propose a novel model named cFlow for bug localization by exploiting the program structure represented by the CFG. cFlow employs a particularly designed flow-based GRU for feature learning from the CFG, where the *flowing nature* is explicitly considered by transmitting the semantics of statements along the execution paths. Experimental results on widely-used real-world software projects show that cFlow significantly outperforms the state-of-the-art bug localization methods, indicating that exploiting the program structure from the CFG with respect to the *flowing nature* is beneficial for improving the performance of bug localization.

For future work, it is interesting to exploit the program structure from the control flow graph for other software mining problems like clone detection or code summarization.

Acknowledgements This research was supported by NSFC(61751306).

Availability of data and material All the data sets are extracted from open-source software projects and the urls are provided in the paper.

Declarations

Conflict of interest This research was supported by NSFC(61751306). The authors have received research support from Nanjing University.

Code availability The source code of the model is not public available.

Ethics approval This paper is approved in ethics.

Consent to participate This paper is consented to participate.

Consent for publication This paper is consented for publication.

References

Allamanis, M., Brockschmidt, M., & Khademi, M. (2018). Learning to represent programs with graphs. *In International Conference on Learning* Representations.

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). *Code2vec: Learning distributed representations of code* Proc. ACM Program. Lang., 3.

Feng, Z., Guo, D., Tang, D.-Y., Duan, N., Feng, X.-C., Gong, M., Shou, L.-J., Qin, B., Liu, T., & Jiang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547. Association for Computational Linguistics.

Fischer, M., Pinzger, M., & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pp. 23–32.

Gay, G., Haiduc, S., Marcus, A., & Menzies, T. (2009). On the use of relevance feedback in ir-based concept location. In *2009 IEEE International Conference on Software Maintenance*, pp. 351–360.

Grover, A., & Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 855–864, San Francisco, California, USA. Association for Computing Machinery.

Huo, X., & Li, M. (2017). Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pp. 1909–1915, Melbourne, Australia.

Huo, X., Li, M., & Zhou, Z.-H. (2016). Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pp. 1606–1612, New York, USA.

Huo, X., Li, M., & Zhou, Z.-H. (2020). Control flow graph embedding based on multi-instance decomposition for bug localization. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, volume 34, pp. 4223–4230, New York, USA.

Huo, X., Thung, F., Li, M., Lo, D., & Shi, S.-T. (2019). Deep transfer bug localization. *IEEE Transactions on Software Engineering*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. In *Proceedings of the Conference on Empirical Methods in Natural Lanugage Processing*, pp. 1746–1751, Doha, Qatar.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kipf, T. N. & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Kochhar, P. S., Tian, Y., & Lo, D. (2014). Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 803–814, Vasteras, Sweden.

Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2015). Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 476–481.

Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2017). Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension*, pp. 218–229.

Li, Y., Wang, S.-H., & Nguyen, T. N. (2020). Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 602–614, New York, NY, USA. Association for Computing Machinery.

Li, Y., Wang, S.-H., Nguyen, T. N., & Van Nguyen, S. (2019). *Improving bug detection via context-based code representation learning and attention-based neural networks* Proc. ACM Program. Lang., 3.

Li, Y.-J., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015). Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Lukins, S. K., Kraft, N. A., & Etzkorn, L. H. (2008). Source code retrieval for bug localization using latent dirichlet allocation. In *2008 15th Working Conference on Reverse Engineering*, pp. 155–164.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space.

Mou, L.-L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. *Proceedings of the AAAI Conference on Artificial Intelligence, 30*(1).

Niepert, M., Ahmed, M., & Kutzkov, K. (2016). Learning convolutional neural networks for graphs. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 2014–2023. PMLR.

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 701–710, New York, NY, USA.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., & Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering, 33*(6), 420–432.

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

Shi, S.-T., Li, M., Lo, D., Thung, F., & Huo, X. (2019). Automatic code review by learning the revision of source code. *Proceedings of the AAAI Conference on Artificial Intelligence, 33*(01), 4910–4917.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. *In International Conference on Learning* Representations.

Wei, H.-H., & Li, M. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 3034–3040.

White, M., Vendome, C., Linares-Vasquez, M., & Poshyvanyk, D. (2015). Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 334–345.

Ye, X., Bunescu, R., & Liu, C. (2014). 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*, pp. 689–699, New York, NY, USA.

Youm, K. C., Ahn, J., & Lee, E. (2017). Improved bug localization based on code change histories and bug reports. *Information and Software Technology, 82,* 177–192.

Zhang, J.-L., Xie, R., Ye, W., Zhang, Y.-H., & Zhang, S.-K. (2020). Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, pp. 219–229, New York, NY, USA.

Zhang, Y.-Y., & Li, M. (2019). Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector. *Proceedings of the AAAI Conference on Artificial Intelligence, 33*(01), 5813–5820.

Zhou, J., Zhang, H.-Y., & Lo, D. (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering*, pp. 14–24.

Zhou, Y.-Q., Liu, S.-Q., Siow, J., Du, X.-N., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *In Advances in Neural Information Processing Systems, 32,* 10197–10207.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.