# FFL: Fine-grained Fault Localization for Student Programs via Syntactic and Semantic Reasoning

Thanh-Dat Nguyen [†],Thanh Le-Cong[‡], Duc-Minh Luong[‡], Van-Hai Duong[§],
Xuan-Bach D. Le[†], David Lo [¶], Quyet-Thang Huynh[‡]
[†]The University of Melbourne
[‡]Hanoi University of Science and Technology
[§]Independent Researcher
[¶]Singapore Management University

***Abstract*—Fault localization has been used to provide feedback for incorrect student programs since locations of faults can be a valuable hint for students about what caused their programs to crash. Unfortunately, existing fault localization techniques for student programs are limited because they usually consider either the program's syntax or semantics alone. This motivates the new design of fault localization techniques that use both semantic and syntactical information of the program.**

**In this paper, we introduce FFL (Fine grained Fault Localization), a novel technique using syntactic and semantic reasoning for localizing bugs in student programs. The novelty in FFL that allows it to capture both syntactic and semantic of a program is three-fold: (1) A fine-grained graph-based representation of a program that is adaptive for statement-level fault localization; (2) an effective and efficient model to leverage the designed representation for fault-localization task and (3) a node-level training objective that allows deep learning model to learn from fine-grained syntactic patterns. We compare FFL's effectiveness with state-of-the-art fault localization techniques for student programs (NBL, Tarantula, Ochiai and DStar) on two real-world datasets: Prutor and Codeflaws. Experimental results show that FFL successfully localizes bug for 84.6% out of 2136 programs on Prutor and 83.1% out of 780 programs on Codeflaws concerning the top-10 suspicious statements. FFL also remarkably outperforms the best baselines by 197%, 104%, 70%, 22% on Codeflaws dataset and 10%, 17%, 15% and 8% on Prutor dataset, in term of *top-1*, *top-3*, *top-5*, *top-10*, respectively.**

***Index Terms*—Fault Localization, Programming Education, Graph Neural Network**

## I. Introduction

Fault localization is the problem of identifying faulty locations in source code which leads to erroneous behaviors triggered when running a test suite. Due to large variability of faulty causes, it is a challenging problem to narrow down the possible root causes from the triggered failure. This is especially hard for students, where they may have little familiarity with the programming language for identifying faulty locations as well as root causes.

Unfortunately, most of existing methods focusing on the real-world programs are not optimized to be effective in the student programs due to the differences between the former and the latter. As an example, current state-of-the-art techniques [20], [21], [23] aim to locate bugs only on the method level, making them hard to be used for student programs, since most of student programs would consists only of a few methods. This motivates a research direction of fault localization which focuses on students programs, as proposed by recent studies [2], [8], [9], [14]. By providing hints of potential bugs' locations, fault localization techniques give useful instructions to students. Indeed, the user studies reported by Edmision et al., [8] have shown that fault localization techniques enable students to make improvements on their code from submission to submission, as well as supporting students to spend less time achieving the maximum score in overall.

Fault localization techniques for student program usually fall into one of the following categories: learning-based approaches and spectrum-based approaches. The former leverage historical data and a deep learning models to learn how to localize bugs in student programs. Meanwhile, the latter leverage spectrum-based fault localization techniques [1], [4], [17], [39], that is widely used for industrial-scale programs, to output the suspiciousness of code statements based on analysis on the coverage data of failed/passed tests.

The learning-based approaches [14], [15], while have shown capability on learning syntactic patterns from historical data, ignore semantic information of programs such as test coverage or execution traces. Meanwhile, the spectrum-based approaches [1], [4], [17], [39] only consider test coverage as the most effective input and ignore the information from source code. This motivates the new design of fault localization techniques that can utilize both syntactic and semantic information present in student programs.

In this paper, we introduce FFL (Fine grained Fault Localization), a novel technique using syntactic and semantic reasoning for localizing bugs in student programs. FFL utilizes both syntactic and semantic information of the program for fault localization via our new design of three main components: (1) *graph-based representation*, namely *syntax-coverage* graph, of a program that comprises syntax and program semantic information via Abstract Syntax Tree and detailed coverage of given tests into one graph; (2) *an effective and efficient deep learning model* (i.e., graph neural network (GNN)) which is able to naturally deal with graph-based representations; and (3) a *node-level training objective* that allows deep learning model to learn from fine-grained syntax patterns.

Compared to other learning-based fault localization techniques (e.g., NBL [14], DeepFix [15]), our technique is highly customizable by design, enabling an easy inclusion of both syntactic and semantic information of a program. DeepFix [15] represents a student program as a sequence of tokens and uses Recurrent Neural Network (RNN) to develop deep learning model to fix syntactic (compilation) errors. This type of errors is different from our method's aim of fixing logical errors that fail on certain test cases. Low et al. [23] aim for method-level fault localization and thus leverage statement-level AST to represent syntactical information. Our aim is more aligned to NBL [14], in which we both address the problem of fault localization of student programs. The main characteristic of student programs is that they are often small in size and errors often lie in sub-statement level (e.g., small AST nodes). Thus, the goal here is to accurately pinpoint error locations at fine-grained level. NBL [14] converts each program's AST into an adjacency-list-like representation, train a Convolutional Neural Networks to predict test cases' outcome, and leverage neural attribution techniques to obtain each line's suspiciousness score. Incorporating richer semantic information such as test coverage into these representations is difficult because it requires to encode structural information into a single sequence or list of sequences. It can be seen that for both of these representations, it is difficult to incorporate semantic information such as code coverage and can suffer the loss of code structure information.

To address the aforementioned problem, a potential solution is by using a graph-based coverage representation. For example, Lou et al. [23] represents both program statements and their coverage relationships to test cases in one unified graph. However, we observe that student errors often lies on sub-statement-level e.g., if statement conditions, logical operators or type cast operators, as reported by Tan et al. [33] (see Codeflaws site for additional details[1]). Hence, these recently introduced graph-based program representations which usually focus on method-level, may be not suitable for our focus, i.e., student program. Thus, we leverage fine-grained syntax representation, i.e., AST-node level instead of statement-level in conjunction with code coverage as semantic information by connecting each node belonging to a statement to test cases which cover the statement. We design a graph neural network (GNN) along with a node-level training objective to extract features from the proposed representation. FFL learns to detect the combination of intra-syntax and inter syntax-coverage pattern that is likely to be faulty. By this way, our approach can effectively capture fine-grained patterns for localizing bugs in student-written programs.

Given a student program and a set of test cases (passing and failing), FFL works in two main phases. In the first phase of input preparation, we leverage the program syntax (i.e., the Abstract Syntax Tree (AST)) and augment this syntax representation with code coverage information using test cases, resulting in a new representation, which we call *syntax-coverage graph*. In the second phase, we first leverage a graph neural network to predict suspiciousness scores at AST node level (i.e., the probability of each AST node being modified to fix bugs). We then aggregate AST-node-level results to obtain the statement-level faulty score.

We evaluated FFL on 2,136 buggy programs from the Prutor dataset [5] and 780 programs from Codeflaws dataset [33]. We compare FFL against the state-of-the-art spectrum-based and learning-based fault localization techniques for student programs, consisting of NBL [14], Tarantula [17], Ochiai [1] and DStar [38]. Experimental results show that FFL successfully localizes bugs for 84.6% out of 2,136 programs on Prutor and 83.1% out of 780 programs on Codeflaws when reporting the top-10 suspicious lines. FFL also remarkably outperforms the best baselines by 197%, 104%, 70%, 22% on the Codeflaws dataset and 26%, 17%, 22% and 38% on the Prutor dataset, in term of *top-1*, *top-3*, *top-5*, *top-10*.

In summary, our contributions include:

- We propose a novel technique, namely FFL, that is the first to combine syntactic and semantic information to automatically localize bugs in student programs.
- We propose *syntax-coverage graph* that can capture fine-grained syntax-semantic representation of programs at AST node-level.
- We design a graph-based deep learning model and a novel training objective to effectively and efficiently learn the proposed graph-based representation for ranking suspicious program statements.
- We conduct evaluations on two popular datasets of student programs. Experiment results show that the unique combination fine-grained syntactic and semantic inform-antion at AST node-level empowers FFL to achieve significant improvements over state-of-the-art baselines.

The remainder of this paper is structured as follows. Section II introduces background and related works on the fault localization and graph neural network, followed by Section III that presents our approach in detail. Section IV describes our experimental setup and our findings. Section V presents threats to validity of our approach. Finally, Section VI concludes and presents future work.

## II. Background & Related Work

### A. Fault Localization

**Problem formulation** In this work, we formulate the fault-localization in student programs as follows:

- **Input**: A student program and a set of failing and passing test cases.
- **Output**: Suspiciousness score indicating the likelihood of a statement being faulty.

To address this problem, we build a deep neural network model to classify whether each node in the AST of the student program source code is faulty. We take the output probability of this model to calculate suspiciousness score of each statement in the aforementioned program. Note that, different from repair-based feedback generation for student

[1]https://codeflaws.github.io/

program [35] our problem formulation does not require the existence of reference programs i.e., a correct implementation provided by teachers/tutors.

**Spectrum-based Fault Localization** Spectrum-based fault localization (SBFL) [1], [17], [18], [22], [24], [27], [40], one of the most popular FL techniques, which considers program entities (e.g., statements, methods, classes) executed by test cases. These techniques take a buggy program and coverage information of all tests as the input and return a ranked list of program entities according to their descending order of suspicious scores. These scores can be calculated by specific formulae, which mainly rely on: (1) the set of all failed/passed tests, i.e., $T_f/T_p$, (2) the set of failed/passed tests executing code element $e$, i.e., $T_f(e)/T_p(e)$, and (3) the set of failed/passed tests that do not execute code element $e$, i.e., $T_f(\bar{e})/T_p(\bar{e})$. For example, Ochiai formula can computes the suspiciousness score of the program entity $e$ as $\text{Susp}(e) = \frac{|T_f(e)|}{\sqrt{|T_f| \times (|T_f(e)| + |T_p(e)|)}}$. While SBFL has been widely adopted, recent studies [2], [7]–[9] proposed to apply SBFL for providing feedback about the root cause of failure in student programs. Edmison et al. [9] have demonstrated that this feedback help students find it easier to make improvements on their code, as well as spending less time overall achieving the maximum score on the instructor assessments. While these techniques have shown their usefulness in providing feedback for students, their effectiveness still needs to be further improved to localize bugs more accurately. Compared to their approaches, FFL supplements syntactic patterns learned from historical bugs using a novel graph-based learning technique to improve effectiveness in localizing bugs. Our experiments demonstrate that FFL outperforms the well-known SBFL techniques (i.e., Tarantula [17], Ochiai [1] and DStar [38]) by a significant margin (see details in section IV-C).

**Learning-based Fault Localization.** While Machine/Deep learning has recently emerged as a powerful framework in solving real-world problems, it can be adopted to improve the effectiveness of fault localization for student programs as pioneered by NBL [14]. The basic idea of NBL is to learn the potential faulty locations via frequent buggy patterns from historical bugs. Toward this, NBL first represents a program in the form of Abstract Syntax Tree; then, it converts the AST into an adjacency list-like representation via performing breadth-first traversal. Finally, NBL utilizes a Convolutional Neural Network (CNN), and neural prediction attribution [32] to predict bug locations. Compared to NBL, our approach designs graph-based representations, allowing FFL to easily include both program syntax and semantic information (i.e., code coverage). Furthermore, we also proposed to use Graph Neural Network (GNN), which demonstrated its superior effectiveness on rich-structured data like source code of programs [6], [11], [13], [41]. Less relevant to our work in this paper are recent research efforts in learning-based fault localization for real-world programs [3], [20], [21], [23], [31]. The current state-of-the-arts among these works, however, [21], [23] only focus on method-level fault localization and ignore statement-level fault localization. This makes it difficult to apply these techniques to student programs, which require a finer-grained localization at statement level due to their small size.

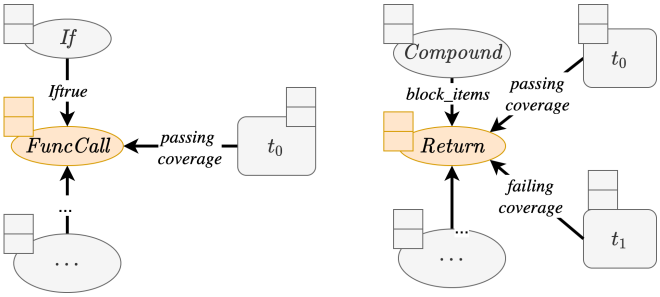



Fig. 1: GNN message passing illustration for two nodes. The rounded rectangular nodes represent test cases, and the ellipsis node represents AST nodes. The target node of message passing process (highlighted in yellow) takes into account its neighbor information as well as the edge type for updating it own hidden features.

### B. Graph Neural Network

Graph Neural Networks (GNNs) is family of a widely-used deep learning techniques for processing data represented by graph-structured data such as knowledge graphs [34], social networks [42] and images recognition [37]. The basic intuition of GNNs is that each node in a graph $G = (V, E)$ can be characterized in terms of: (i) its own features, (ii) the relations it has with its neighboring nodes, and (iii) the features of its neighbors. Toward this, GNNs learn representations of nodes via the *message passing process* [11], in which each node features are updated via a $message$ that gathers information from its own features and the neighbors' features. The message passing operation works with all nodes in parallel, updating each node feature of the graph as a result. By stacking $T$ consecutive message passing process, a $T$-layered GNNs is obtained, where each layer enriches graph node representations while allowing aggregation of features from an extra neighbor hop.

Formally, suppose we have a graph $G = (V, E)$ where $V$ is the set of nodes, and $E$ is the set of edges. Each node in $G$ retains a node feature $\mathbf{x}$, and each edge is assigned an edge feature $e$. More specifically, at layer $t$, each existing node $i$ with assigned feature $\mathbf{x}_i^{(t)}$ will be updated to new feature $\mathbf{x}_i^{(t+1)}$ as follows:

$$\mathbf{m}_i^{(t+1)} = \sum_{j \in \mathcal{N}(i)} f_{\text{mess}}(\mathbf{x}_i^{(t)}, \mathbf{x}_j^{(t)}, e_{ij}) \tag{1}$$

$$\mathbf{x}_i^{(t+1)} = f_{\text{upd}}\left(\mathbf{x}_i^{(t)}, m_i\right) \tag{2}$$

where $f_{\text{mess}}$ and $f_{\text{upd}}$ is the message function and update function for $t$-th layer and $\mathcal{N}(i)$ is node $i$'s set of neighbor.

The message passing mechanism is applied to the GNN of FFL in order to aggregate information from both syntax and coverage neighbor nodes, 1 hop at a time for each node

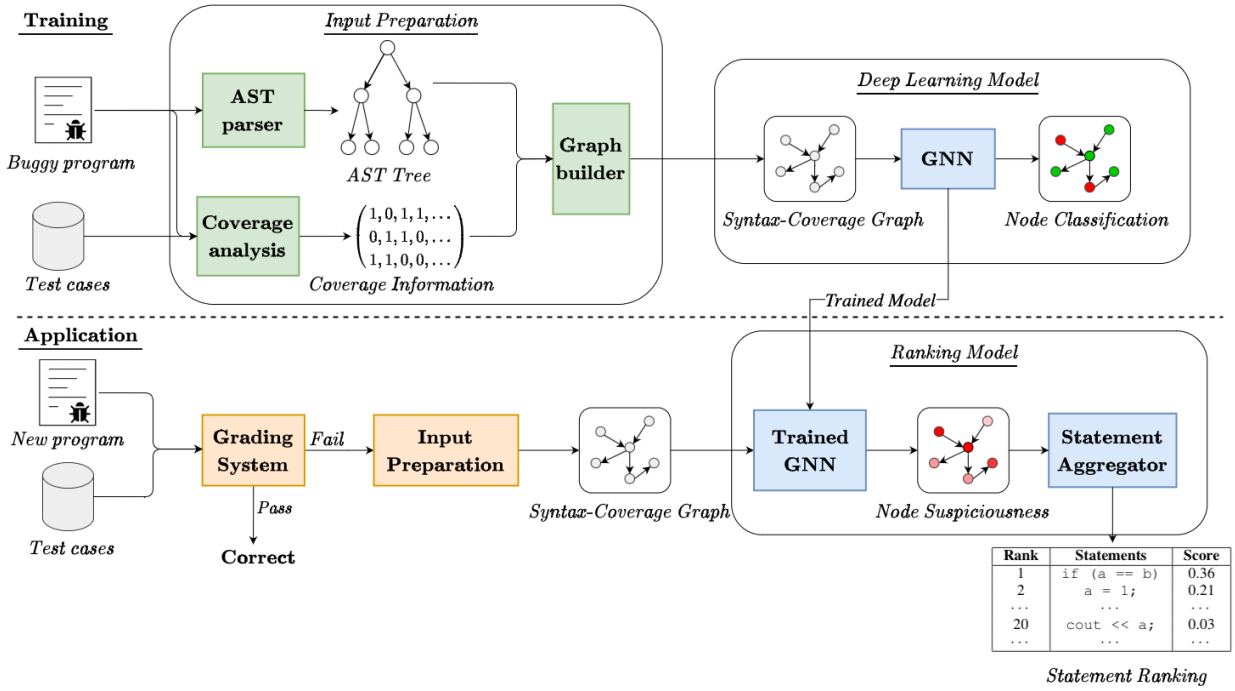



Fig. 2: The overview framework of FFL

in parallel as shown in Figure 1. At each message passing operations, every node takes into account the type of edge as well as information of neighboring nodes to update its own hidden representation.

## III. Our Approach

In this section, we introduce a deep learning model over syntactic and semantic features for fault localization in student programs. Toward this goal, we describe a graph-based representation comprising of both program syntax (i.e., Abstract Syntax Tree) and semantic features (i.e., coverage information) with a tailored graph neural network (GNN) to identify buggy locations based on this representation. The key idea is that by leveraging the graph-based representation, we can treat fault localization problems as node classification on a graph, in which we predict whether each node in a graph is erroneous. Furthermore, while aiming for statement-level fault localization is a straightforward option, we hypothesize that a node-level feedback signal would boost model performance. Thus, we design a training objective based on node-level AST differencing as the label for graph node classification.

### A. Syntactic and Semantic Program Representation

GNN is widely known for its effectiveness in dealing with structured data. However, in order to leverage the power of GNN, designing an expressive representation of the input data is crucial.

To achieve this, we propose to use both syntax and semantic information to build a graph-based representation of input programs that empowers GNN to learn effectively.

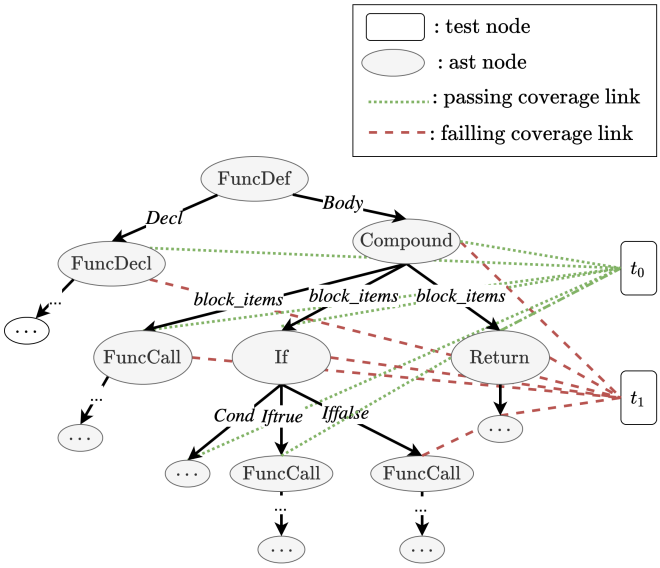



Fig. 3: Syntax-Coverage graph: Program syntactic is represented by AST while the semantic is represented via coverage information

Finally, we obtain a representation that combines both syntactic and semantic information, namely *syntax-coverage graph*. We lay out the details of our syntax-coverage graph construction below.

*1) Syntactic Representation via Abstract Syntax Tree:* In the syntax-coverage graph, we incorporate program syntax by leveraging AST as a subgraph in our representation. In detail,

given an abstract syntax tree in the form of a graph $G_{ast} = (V_{ast}, E_{ast})$, where each node $v \in V$ represents a node in the abstract syntax tree, a directed edge $(v_i, v_j) \in E_{ast}$ exists for every parent-child in the AST, we incorporate every node in the abstract syntax tree as a part of our syntax-coverage graph (i.e. $V_{ast} \subseteq V_H$ and $E_{ast} \subseteq E_H$). As an illustration, AST nodes are represented as ellipse-shaped nodes in Figure 3.

*2) Semantic Representation via coverage information:* As another part of the syntax-semantic representation, we include semantic features via test coverage information. Given a test suite $T = \{t_1, t_2, t_3, \ldots t_{N_T}\}$ where $t_i$ for $i \in 1..N_T$ is a test case, we represent each test case as a node in the syntax-coverage graph. In order to enrich the syntax-coverage representation, we obtain the outcome for each of these test cases by running the test suite and embed the outcome of into their corresponding edges between their test nodes and the covered AST nodes: For the example in Figure 3), since test $t_0$'s outcome is *passed*, its connections towards the covered AST nodes are *passing* edges (dotted edges), for the failed test $t_1$, the edges will be of *failing* type (dashed edges in Figure 3). This aids the GNN model in distinguishing test case types (i.e. *passed* or *failed*) and aggregating this information towards its covered AST node. Subsequently, this representation allows FFL learning to make use of both faulty syntax patterns and syntax-coverage patterns that frequently presents in the dataset.

### B. Proposed Model

An overview of FFL's architecture is shown in Figure 2. FFL works in two phases. In the training phase, it learns a deep learning model to determine whether each AST node in the syntax-coverage graph is faulty. The training data consists of a set of historical bugs, consisting of a buggy program, a set of tests (failing and passing), and ground truth bug locations. The training phase of FFL consists of two main steps:

- **Input Preparation.** (Section III-B1) FFL first uses AST parser and coverage analysis tool to produce AST tree and coverage information of program. Then, it uses a graph builder to construct syntax-coverage graph of input program.
- **Node classification via GNN.** (Section III-B2 and III-B3) FFL takes the syntax-coverage graphs and the ground truth locations to train a graph neural networks that determines whether a syntax-coverage graph's AST node is faulty. This model is the overall output of the training phase that is passed to the deployment phase.

In the deployment phase (Section III-B4), FFL takes as input a set of test cases (including both failing and passing tests) and a buggy program, and constructs syntax-coverage graphs through input preparation. After that, FFL uses the pre-trained model to produce a suspiciousness score for each node of syntax-coverage graphs. Then, FFL computes the suspiciousness score of a statement by aggregating the score of each node that belongs to the statement. Finally, FFL produces a ranked list of statements that are likely responsible for the failing test cases.

*1) Input Preparation:* Given a buggy program and a set of tests (passing and failing), FFL constructs syntax-coverage graph $G$ as follows. We first parse the buggy program by `pycparser` [2] to obtain AST representations $G_{ast}$ for the program. Simultaneously, we run the buggy program over the given tests and perform coverage analysis by using `gcov` [3] to obtain coverage information.

Note that, `gcov` only provides coverage information at statement level. Hence, we associate the line-level coverage information obtained by gcov to the AST nodes by connecting each node in a statement to test cases that cover the statement. Then, FFL connects $G_{ast}$ of each program into one graph by including test nodes and coverage edges according to coverage information. Finally, we annotate each node and edge with its attributes in the graph. As mentioned in Section III-A, we annotate nodes and edges of AST following its type generated by `pycparser`. Meanwhile, test nodes and coverage edges types are determined based on test outcomes. More specifically, the edge will be of *passing* type if the test outcome is passed; otherwise, the edge will be of *failing* type.

*2) Graph Neural Network Architechture:* Given heterogeneous syntax-coverage graph $G_H = (V_H, E_H)$ where $V_H = V_{ast} \cup V_{test}$ and $E_H = E_{ast} \cup E_{cov}$. We proposed a GNN that takes input as $G_H$ and provides a prediction label for each node $v \in V_{ast}$. As input representation of GNN, we use an *embedding layer* to retrieve numerical feature representation of each node and each edge. This input is then fed through several *message passing layers*, with each layer updating each node feature. We take each node's representation output of the last layer and calculate node label prediction. Finally, we aggregate the node-level predictions to retrieve statement-level suspicious scores from them.

**Embedding layer.** Each node in the syntax-coverage graph should be assigned with a corresponding node type $t \in T_{nodes}$ where $T_{nodes}$ is the set of all node labels (e.g. $FuncCall, If, Test$ etc. ), which we leverage to obtain the numerical feature of each node in the input graph. In detail, each node type $t$ is encoded using an one-hot vector $\mathbf{x}_t \in \{0,1\}^{|\mathcal{T}_{nodes}|}$, where each dimension in the vector is set to 1 if the node is of corresponding type. We stack these node features to obtain the feature matrix $\mathbf{X}_H^0 \in \{0,1\}^{|V_H| \times |\mathcal{T}_{nodes}|}$. Finally, we apply a linear transformation to $\mathbf{X}_H^0$ to obtain hidden representation $\mathbf{H}^{(0)} \in \mathbb{R}^{F^{(0)}}$ where $F^{(0)}$ is a chosen hyper-parameter for embedding. Each row of $\mathbf{h}_i^{(0)} = \mathbf{H}_{i,:}^{(0)}$ corresponds to a node's embedding.

**Message-passing layer.** Given node embedding, we leverage graph input structure to update hidden node features through layer via widely-used message passing mechanism [11]. The specifically chosen message passing mechanism has to be flexible towards multiple types of edges. For this task, we choose R-GCN [30] which has been known for its capability in dealing with heterogeneous graph, its form is shown below:

[2]https://github.com/eliben/pycparser

[3]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{T}_{edges}}\sum_{j\in\mathcal{N}_i^{te}} \mathbf{W}_r^{(l)}\mathbf{h}_j^{(l)} + \mathbf{W}_0^{(l)}\mathbf{h}_i^{(l)} + \mathbf{b}^{(l)}\right) \tag{3}$$

Where $l$ is the layer index, $\sigma$ is ReLU activation function [26], $\mathbf{W}_r^{(l)}, \mathbf{W}_0^{(l)} \in \mathbb{R}^{F^{(l)}}, \mathbf{b}^{(l)}$ is layer $l$'s learnable parameter, $\mathbb{R}^{F^{(l)}}$ is a hyper parameter of choice and $\mathcal{N}_i^r$ is $1-$hop neighbor having relation of type $r$ with the node of target. Intuitively, each node feature is updated in parallel.

After passing through $L$ GNN message passing layers [11], we obtain $L$-th hidden feature matrix of each node: $\mathbf{H}^{(L)} \in \mathbb{R}^{F^{(L)}}$

**Output layer.** Each of node's classes probabilities are obtained by applying a linear transformation followed by class-wise softmax:

$$\mathbf{O} = \text{softmax}\left(\mathbf{W}_o\mathbf{H}^{(L)} + \mathbf{b}_o\right) \in [0,1]^{|V_H|\times C} \tag{4}$$

Where $\mathbf{W}_o$ and $\mathbf{b}_o$ are learnable parameters and $C$ is the number of classes. Each row $\mathbf{o}_i = \mathbf{O}[i,:] \in \mathbb{R}^C$ represent each node's probability of belong to each class. In our case $C$ is set to 3, with classes $0, 1, 2$ correspond to *unmodified, modified*, and *inserted* respectively.

*3) Training objective:* To prepare labeling for training, we make use of Gumtree fine-grained AST differencing algorithm [10]. Taking as input the source code of two versions, one buggy and one fixed source code, Gumtree output mapping between AST nodes of the source code to a fixed version. We determine the *unmodified, modified, inserted* nodes via the following procedure:

- **Modified nodes** are either removed or changed in content: In the former case, we search nodes having no correspondence from the buggy to fixed version whereas to identify the latter, we find nodes having its content changed between the buggy and fixed version.
- **Inserted nodes** are nodes that have no correspondence from the fixed version to the buggy version.

We perform automated annotation on the buggy source code version based on the aforementioned procedure. Inserted node is tricky to annotate due to its lack of appearance in the under-annotation buggy source AST; for this, we further use these annotations:

- For token/expression-level modification, we would take the parent that has not been modified in the buggy version and label the parent node as *insertion*.
- For statement-level insertion, we take the previous statement for annotation of *modification* the common parent to be annotated as *inserted*

With this, we obtain fine-grained annotation for training and evaluation (see Figure 4 for an illustration).

Since our final objective is statement-level suspicious score, a straightforward approach might be performing classification on statement nodes: where we classify whether each statement contains *modified* or *inserted element*. However, since we wish to learn fine-grained AST transformation, we instead adopt node-level classification to predict which of the aforementioned classes each node belong, then aggregate these detailed prediction to obtain statement-level prediction. We show the effectiveness of node-level classification in comparison with statement-level classification in Section IV-C.

In order to classify each node to one of the aforementioned classes, cross entropy objective is a popular objective function, which we leverage to learn node-level classification on each graph:

$$\mathcal{L}_{\boldsymbol{\theta}}(\mathbf{O}, \mathbf{L}) = -\sum_{\substack{c\in C \\ i\in 1..|V_H|}} \mathbf{L}_{i,c}\log \mathbf{o}_{i,c} \tag{5}$$

Where $\mathbf{L} \in 0, 1^{|V_H|\times C}$ is label annotation obtained from aforementioned annotation process for each node and $\mathbf{o}_{i,c}$ is the probability of node $i$ belonging to class $c$. For our main approach, class $c$ can be either *unmodified, modified* or *inserted* and $i$ indicate each node in AST Tree. For statement-level version used for comparison in Section IV-B, the class $c$ is either *unmodified* or *modified* and $i$ indicate each statement node in the AST tree. As mentioned, since node-level prediction is different from our final objective of statement-level fault localization, we introduce the process of obtaining statement-level fault localization from node-level prediction below.

*4) Applications:* After the training phase, users can leverage the trained FFL to determine statements that are likely to be responsible for the failure of their programs. Given a student program that is failing in the grading system (i.e. it fails at least one test case), FFL first constructs a syntax-coverage graph via *input preparation* and combines the trained GNN model's node level output wih a statement-level *Ranking Model* as follows to provide statement-level suspicious score:

- **Node suspiciousness**. In this step, FFL uses the trained GNN model from training phase to produce a suspiciousness score $score_i$ for a node $i$ of the syntax-coverage graph as follows:

  $$score_i = 1 - \mathbf{o}_{i,0} \tag{6}$$

  Here, $\mathbf{o}_{i,0}$ represents the probability that node $i$ should be *unmodified*.

- **Statement Prediction Aggregator**. Since our target is statement-level prediction, as our current GNN prediction targeted AST nodes in general (i.e. *expression/token* are also included), we introduce our method to convert from the AST-node prediction to the corresponding statement suspiciousness score. In detail, the suspiciousness score of a statement is calculated by taking the maximum of all nodes in its correspoding subtree $\mathcal{S}$ in AST. More formally, given a statement $s$ with correspoding subtree $\mathcal{S}$ in AST, we computed its suspiciousness score as follows:

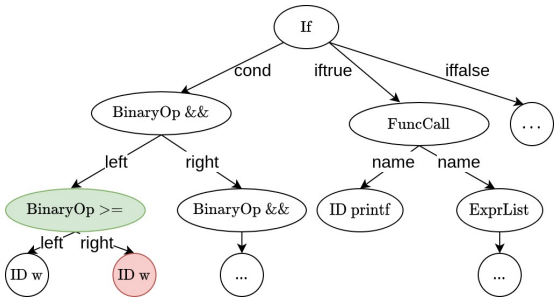
  $$Susp(s) = \max_{i\in\mathcal{S}}\{score_i\} \tag{7}$$

  Where $score_i$ is the node's suspicious score calculated in Equation 6.

Finally, we use the scores from *Statement Aggregator* to rank statements' suspiciousness.

```
1: #include <stdio.h>
2: int main(int argc, char* argv[]){
3:     int w;
4:     scanf("%d", &w);
5:     if (w>= w && w % 2 == 0)
6:         printf("YES");
7:     else
8:         printf("NO");
9:     return 0;
10: }
```

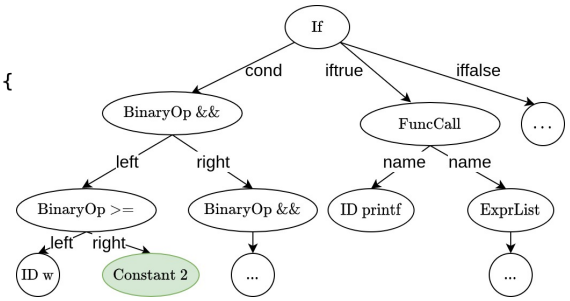
(a) Buggy code




(b) Corresponding AST of buggy version

```
1: #include <stdio.h>
2: int main(int argc, char* argv[]){
3:     int w;
4:     scanf("%d", &w);
5:     if (w>= 2 && w % 2 == 0)
6:         printf("YES");
7:     else
8:         printf("NO");
9:     return 0;
10: }
```

(c) Fixed Code




(d) Corresponding AST of fixed version

Fig. 4: Illustration example for fine-grained annotation.

## IV. Empirical Evaluations

### A. Experimental Methodology

*1) Dataset:* To evaluate the effectiveness of fault localization techniques in the student program, we use two benchmarks introduced in previous works [14], [33], which contain student-written programs for programming assignments from real-world tutoring systems, Prutor [5], and Codeforces [25]. The type of test provided in both 2 datasets is system-level testing, the tests are presented in the form of input-output examples. In order to construct the dataset, we leverage Gumtree [10] to capture the changes between the AST tree of buggy and fixed programs. Note that, since Prutor does not originally contain the buggy-fixed pairs as Codeflaws [33], we construct these pairs using the implementation[4] provided by [14]. The details of dataset distribution are shown in Table I.

TABLE I: The statistics of datasets."# Programs" represents the number of buggy programs in each datasets. "# KLOC" and "# Test" correspond to average size of the program and number of test cases, respectively.

| Dataset | #Programs | #KLOC | #Test |
|---|---|---|---|
| **Prutor** | 6,171 | 25 | 8 |
| **Codeflaws** | 3,902 | 36 | 43 |

**Prutor** was collected from an introductory programming course at the Indian Institute of Technology, Kanpur, India using a programming tutoring system called Prutor [5]. The dataset contains 6,171 buggy programs across these 29 algorithmic implementation tasks with a total of 231 tests. Each program in Prutor datasets contains about 25 lines of code on average.

**Codeflaws** was extracted from submissions in Codeforces [25], a well-known programming contest website. The dataset contains 3,902 buggy programs across these 1,284 algorithmic implementation tasks with 43 tests on average. Each program in Codeflaws datasets contains about 36 lines of code on average.

*2) Evaluation Metrics:* In this paper, we use *top-n* which counts the number of bugs successfully localized within *top-n* position of the resultant ranked list as our evaluation metric, which is also commonly used in prior works [3], [14], [20], [21] following findings of Parnin and Orso [29] that programmers will only inspect the first few positions in a ranked list of potentially buggy statements. Followings prior studies [14], [21], we report *top-1*, *top-3*, *top-5* and *top-10*. Note that if two statements share the same suspicious score, we break the tie. Higher is better for this metric.

*3) Experimental Settings:* We implement the proposed model by DGL [36] library and Python programming language. The model is trained and evaluated on an NVIDIA GTX 1080 Ti GPU with 11GB of graphics memory. We train the model from scratch using Adam optimizer with the learning rate of 0.0001. We randomly selected 60% samples for the Codeflaws dataset as the training set, 20% samples as the evaluation set, and 20% for validation. For Prutor dataset, we follow settings of NBL [14], which use 2,136 samples as evaluation set and 4,035 samples for the training set and

[4]https://bitbucket.org/iiscseal/nbl

validation.

### B. Research Question

**RQ1: How effective is FFL?** In this research question, we evaluate how effectively *FFL* successfully localize bugs for 2,136 program in Prutor datasets and 780 programs in Codeflaws datasets, computing *top-n* with $n \in \{1, 3, 5, 10\}$.

**RQ2. How does FFL compare to previous approaches?** In this research question, we compare *FFL* to four previous techniques, including:

- *NBL* [14]: the current state-of-the-art of bug localization on student program.
- *Ochiai* [1], *Tarantula* [17] and *DStar* [38]: well-known bug localization techniques for industrial-scale programs.

**RQ3: How efficient is FFL?** In this research question, we measure the average running time needed for *FFL* to output a ranked list of statements for a given bug.

**RQ4: How effective is our approach using fine-grained representation and loss function at AST node level?** In this research question, we study the effect of representation and loss function on the overall performance of our approach. In particular, we evaluate the use of statement-level and node-level represention, as well as, statement-level and node-level loss functions. Note that, by default, our approach uses node-level representation and loss function. For all experiments in this study, we use the same model described in Section III-B2, which consists of a single linear encoding layer followed by 5 R-GCN layers.

### C. Findings

*1) RQ1: Overall Effectiveness:* As shown in Table II, FFL is able to localize bug for more than 80%, concerning the top-10 suspicious lines per program, for both evaluation datasets: Codeflaws and Prutors. In detail, FFL successfully localizes 83.1%, 67.6%, 46.2% and 31% out of 780 bugs of Codeflaws datasets in terms of average *top-10*, *top-4*, *top-3* and *top-1* positions, respectively; whereas the results for Prutor are 84.6%, 64.7%, 51.6%, 29.6% respectively.

TABLE II: Overall effectiveness of FFL on Codeflaws and Prutor datasets in term of $top - n(\%)$.

| Dataset | top-10 | top-5 | top-3 | top-1 |
|---|---|---|---|---|
| **Codeflaws** | 83.1 | 67.6 | 46.2 | 31 |
| **Prutor** | 84.6 | 64.7 | 51.6 | 29.6 |

We emphasize that there exists significant differences in distribution between the two datasets: In particular, Prutor contains about 6,200 student programs for 29 programming tasks, which is equivalent to 213.8 programs per task. Meanwhile, Codeflaws only provide 2.7 programs per task (3,902 student programs for 1,428 programming tasks), nearly 50 times lower than Prutor. Moreover, Prutor only provide 8 tests per program while each program of Codeflaws contains 43 tests on average.

Despite of these differences, we note that FFL's performance remains stable while consistently outperforms the varied performance of the baselines as proven in **RQ2**.

> **Answers to RQ1:** FFL has promising performance in bug localization on student programs. FFL is able to localize at least one bug for 84.6% on Prutor and 83.1% on Codeflaws when considering the top-10 suspicious statements.

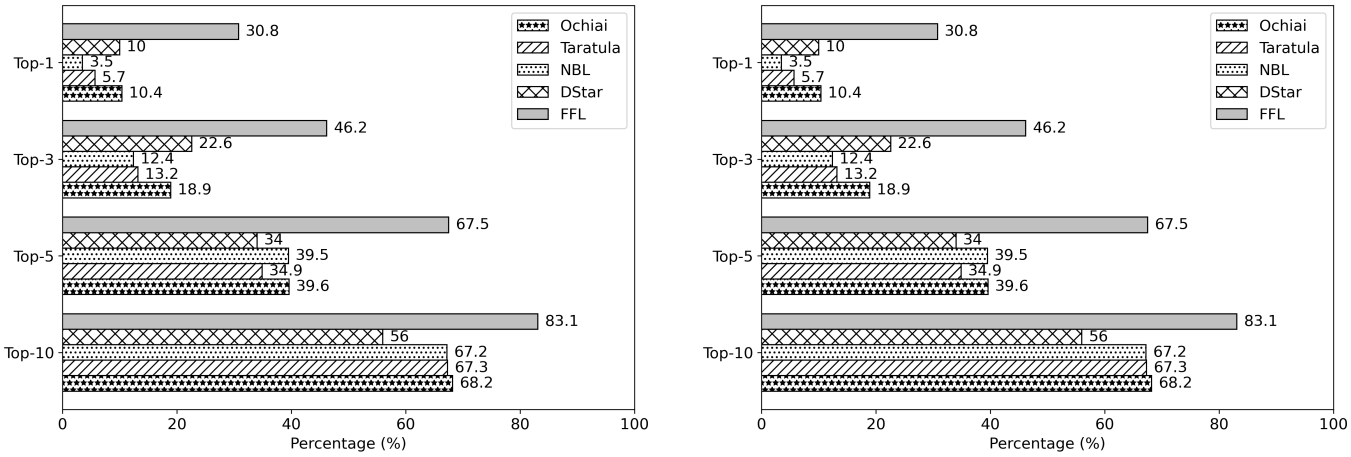
*2) RQ2: Comparison with Baselines:* Figure 5a and 5b show the effectiveness of FFL and four baselines on the two evaluation datasets: Codeflaws and Prutors. Among the techniques, FFL outperforms baselines in all metrics.

**FFL vs. Learning-based Fault Localization**. FFL outperforms NBL by 38%, 22%, 17% and 26% on Prutor dataset and 24%, 71%, 272% and 780% on Codeflaws dataset, in terms of average *top-10*, *top-5*, *top-3*, *top-1*. It can be seen that FFL shows an improvement of at least 20% on most metrics, especially our approach achieves 3 and 8 times higher top than NBL on Codeflaws dataset concerning *top-3* and *top-1* suspicious lines, respectively. Furthermore, the performance of NBL drops remarkably when switching from Prutor to Codeflaws. The reason behind this slide is the difference between two datasets (as discussed in RQ1). It shows that NBL is less effective on datasets where programs are diverse and almost different. Meanwhile, the effectiveness of FFL is almost stationary, showing that FFL is able to deal with various types of datasets due to the ability of capturing frequent buggy patterns.

**FFL vs. Spectrum-based Fault Localization**. The evaluation result shows that FFL perform stably in both datasets while spectrum-based techniques are only effective in Prutor dataset and achieve much more lower result in Codeflaws. Hence, although FFL only perform better than spectrum-based techniques by 8%, 15%, 17%, and 10% on Prutor dataset, our approach remarkably outperforms these techniques by 22%, 70%, 104% and 197% on Codeflaws dataset, in terms of *top-10*, *top-5*, *top-3*, *top-1*, respectively.

> **Answers to RQ2:** Overall, FFL outperform every baseline approach, including the state-of-the-art learning-based approach for student programs, NBL. As compared to the best baseline, our approach achieves an improvement of 197%, 104%, 70%, 22% on Codeflaws dataset and 10%, 17%, 15% and 8% on Prutor dataset, in term of *top-1*, *top-3*, *top-5*, *top-10*.

*3) RQ3: Efficiency:* The average time to output a ranked list of statements for a given program from Prutor is 0.009 seconds with a standard deviation of 0.001 seconds. Meanwhile, these numbers for Codeflaws are 0.0014 and 0.012, respectively. In particular, FFL required only 0.009 seconds for localizing bugs in the best case in both datasets. In the worst case, FFL also consumes less than 0.3 second (0.253 seconds). In conclusion,




(a) Results on Codeflaws

(b) Results on Prutor

Fig. 5: Comparison of FFL with four baseline techniques on the 2 datasets, in term of $top-n(\%)$

it can be seen that the inference time of FFL is reasonable in practice.

Table III shows the average inference time for FFL on two evaluation datasets: Prutor and Codeflaws.

TABLE III: Inference time of FFL (in seconds)

| Dataset | Mean | Std | Max | Min |
|---|---|---|---|---|
| **Prutor** | 0.009 | 0.001 | 0.010 | 0.009 |
| **Codeflaws** | 0.014 | 0.012 | 0.253 | 0.009 |

**Answers to RQ3:** FFL has reasonable inference time in practice with 0.009 and 0.014 seconds on average for localizing bugs from Prutor and Codeflaws, respectively. In the worst cases, FFL also consumes less than 0.3 second for prediction.

*4) RQ4: How effective is FFL using fine-grained representation and loss function at AST node level?:* As shown in Table IV, the model trained with node-level training objective outperforms the one trained with statement-level objective on evaluation metrics in both evaluation datasets.

It can be seen that, node-level training objective shows improvement of 32%, 8%, 20%, 13% on Codeflaws dataset in terms of *top-1*, *top-3*, *top-4*, *top-10*. On Prutor dataset, however, statement-level training objective nearly approximate node-level training objective in *top-5*, *top-1* and even outperformed in *top-3* for node-level AST and *top-10* for statement-level AST respectively. Additionally, statement-level AST performs much better in Prutor while exhibiting a decrease in recall on Codeflaws. The rationale behind this may be attributed to the ratio of sub-statement-level insertion/modification between the two datasets. On Codeflaws, we find the the number of sub-statement-level insertion and modification is $3,692$ over $3,902$ buggy-fixed program pairs (about $94$ percent of fixing is sub-statement level) while this ratio is slightly decreased to $4,863$ over $6,171$ buggy-fixed program pairs of Prutor (only $78$ percent of fixing is sub-statement level).

Overall, experiment results demonstrate that node-level representation and loss function help FFL better capture finer granularity of syntactic transformations and improve the performance in general.

**Answers to RQ4:** Representation and loss function at node level is empirically better than any combinations of those at statement level and contributed to the significantly better results of our approach in comparison with the best baselines.

## V. Discussion

### A. FFL performance on cases where almost all test fails

Since FFL does not require a correct implementation at runtime and instead uses pass/fail and coverage information in conjunction with the source code, we believe it would give further insight to FFL's applicability to evaluate the method's performance in the cases of which the program fails almost completely (i.e., where most of the test fails). Towards this assessment, we first collect the programs matching the criteria (i.e., the ratio of failing tests is over 90%) in the Codeflaws and Prutor dataset. The result is that there are no programs provided Codeflaws [33] dataset that meets this specification (i.e., most of the program would fail only 1-2 test cases over the total of 30-40 test cases) and a total of 36 programs matches the criteria in the Prutor [5]. The performance of the trained FFL on these program are *top-1*: $8.82\%$, *top-3*: $26.4\%$, *top-5*: $50\%$ and *top-10*: $88.2\%$ respectively. This hint that FFL might experience a performance drop in cases lacking positive coverage information associated with the source code, the

TABLE IV: Overall performance of FFL with statement-level and node-level training objective. $Baseline$ shows best results of our baseline. $Statement$ and $Node$ shows the results of statement-level and node-level training objective, respectively. The bold numbers denote the best result for each metric.

| Dataset | Metrics | Baseline | Statement Loss + Node-level AST | Statement Loss + Statement-level AST | Node Loss + Node-level AST |
|---|---|---|---|---|---|
| Codeflaws | top-1 | 10.4 (Ochiai) | 23.5 | 22.4 | **31** |
| | top-3 | 22.6 (Dstar) | 42.7 | 41.6 | **46.2** |
| | top-5 | 39.6 (Ochiai) | 56.4 | 53.2 | **67.6** |
| | top-10 | 68.2 (Ochiai) | 73.5 | 73.1 | **83.1** |
| Prutor | top-1 | 27 (Dstar) | 28.0 | 27.3 | **29.6** |
| | top-3 | 44.1 (NBL) | **52.5** | 50.9 | 51.6 |
| | top-5 | 58.8 (Dstar) | 65.4 | 64.7 | **67.6** |
| | top-10 | 78.6 (Dstar) | 84.6 | **86.7** | 84.6 |

comparable *top-10* recall might be due to the relatively smaller number of statements in these provided programs (21.805 lines of code on average) in comparison with the average 25 lines on the full dataset [5].

### B. Threats to validity

**Threats to internal validity** refer to possible errors in our implementation and experiments. To mitigate this risk, we have carefully checked our implementation to the best of our abilities. Moreover, we also use externally created datasets: Codeflaws and Prutors. Since these datasets are created by others, it reduces experimenter bias.

**Threats to external validity** correspond to the generalizability of our findings. We have evaluated our approach on 2,916 real-world student program from 2 well-known programming tutoring system: Codeflaws and Prutor. These evaluation dataset only includes C programs. In the future, we plan to further mitigate this threat by evaluating FFL on more bugs from other programming systems in various programming languages.

**Threats to construct validity** relate to the suitability of our evaluation metrics. To mitigate this threat, we make use of *top-n* ($n \in \{1, 3, 5, 10\}$), which is widely used in prior works in the field of fault localization [3], [14], [20], [21], [23] following findings by Parnin and Orso [29] which recommend the use of absolute ranks rather than percentages of program inspected.

## VI. Conclusion and Future Work

Providing feedback on student-written programs is an integral part of the programming tutoring system for programming. However, this task is tedious, error-prone, and time-consuming and requires a lot of effort from the teaching personnel [12]. Recent studies [9] have shown that fault localization is useful in providing feedback for students. Unfortunately, existing fault localization techniques for student programs are limited because they usually consider either the program's syntax or semantics alone. This motivates the new design of fault localization techniques that can utilize both syntactic and semantic information of programs. In this paper, we introduce FFL (Fine-grained Fault Localization), a novel technique using syntactic and semantic reasoning for localizing bugs in student program. To realize FFL, we first propose a novel program representation (that combines AST and program spectra), followed by a graph-based deep learning model and trained using a novel training objective. Our evaluation on 2,916 real-world student programs from two well-known programming tutoring systems has shown that FFL successfully localizes at least one bug for more than 83% programs when reporting the top-10 suspicious lines. FFL also remarkably outperforms the best baselines 197%, 144%, 70%, 22% on Codeflaws dataset and 26%, 17%, 22% and 38% on Prutor dataset, in term of *top-1*, *top-3*, *top-5*, *top-10*. The results hint that FFL's better performance benefited from the combination of the graph-based representation, i.e., syntax-coverage graph, in conjunction with the graph neural network and finally, fine-grained training objective based on node-level AST differencing while FFL execution time is empirically evaluated to be within a reasonable performance, hinting its practicality. Overall, the evaluations indicate that our approach may provide useful feedback for students about the root cause of the failures that they encounter.

In future work, we plan to improve FFL by extending the training datasets, incorporating student programsfrom other programming tutoring systems such as Hackerrank [16] or LeetCode [19] in addition to Prutor [5] and Codeforces [25]. Moreover, even though our trained model has achieved decent performance without the tuning of models' hyper-parameters (e.g., number of layers, hidden-dimensions, etc.), additional effort in selections of the hyper-parameters or leveraging graph neural networks oriented techniques [28], [43] may further improve task performance, model robustness and explainability. Additionally, we plan to extend FFL to localize bugs from other programming languages and in challenging scenarios such as the discussed case where the majority of test cases fails. Finally, we plan to extend this work for automated program repair. Specifically, it will be interesting to see if our graph-based representation and deep learning model can be adapted to the generative modeling of patches

**Dataset and Tool Release.** FFL's dataset and implementation are publicly available at https://github.com/FFL2022/FFL.


## Acknowledgement

We thank the anonymous reviewers for helpful comments. This material is based in part upon work supported by the Aus-

tralian Research Council's Discovery Early Career Researcher Award under project number DE220101057.

## REFERENCES


[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[2] E. Araujo, M. Gaudencio, D. Serey, and J. Figueiredo, "Applying spectrum-based fault localization on novice's programs," *Proceedings - Frontiers in Education Conference, FIE*, vol. 2016-Novem, 2016.

[3] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[4] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[5] R. Das, U. Z. Ahmed, A. Karkare, and S. Gulwani, "Prutor: A system for tutoring cs1 and collecting student programs for analysis," *arXiv preprint arXiv:1608.03828*, 2016.

[6] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[7] B. Edmison and S. H. Edwards, "Applying spectrum-based fault localization to generate debugging suggestions for student programmers," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2015, pp. 93–99.

[8] ——, "Experiences using heat maps to help students find their bugs: Problems and solutions," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 260–266.

[9] ——, "Turn up the heat!: Using heat maps to visualize suspicious code to help students successfully complete programming problems faster," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2020, pp. 34–44.

[10] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, vol. 85, no. 1. New York, NY, USA: ACM, sep 2014, pp. 313–324. [Online]. Available: https://dl.acm.org/doi/10.1145/2642937.2642982

[11] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[12] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, 2018.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[14] R. Gupta, A. Kanade, and S. K. Shevade, "Neural attribution for semantic bug-localization in student programs," in *NeurIPS*, 2019.

[15] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[16] HackerRank, "Hackerrank," http://hackerrank.com/, accessed: 2021-09-28.

[17] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[18] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 114–125.

[19] Leetcode, "Leetcode: The world's leading online programming learning platform," http://leetcode.com/, accessed: 2021-09-28.

[20] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[21] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[23] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676.

[24] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[25] M. Mizayanov, "Codeforces: The only programming contests web 2.0 platform," http://codeforces.com/, accessed: 2021-09-28.

[26] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[27] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[28] T.-D. Nguyen, T. Le-Cong, T. Nguyen H., X. B. D. Le, and Q. T. Huynh, "Towards the analysis of graph neural network," in *ACM/IEEE 44nd International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE '22. Association for Computing Machinery, 2022.

[29] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[30] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[31] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[32] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3319–3328.

[33] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[34] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool, "Multi-task learning for dense prediction tasks: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[35] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 481–495, 2018.

[36] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks," *arXiv preprint arXiv:1909.01315*, sep 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[37] X. Wang, Y. Ye, and A. Gupta, "Zero-shot recognition via semantic embeddings and knowledge graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6857–6866.

[38] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[39] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[40] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software*

*and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 449–456.

[41] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *International conference on machine learning*. PMLR, 2019, pp. 6861–6871.

[42] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[43] X. Zhang and M. Zitnik, "GNNGuard: Defending Graph Neural Networks against Adversarial Attacks," *Advances in Neural Information Processing Systems*, vol. 2020-Decem, no. NeurIPS, jun 2020.