Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation

MD NAKHLA RAFI, Concordia University, Canada
DONG JAE KIM, DePaul University, USA
AN RAN CHEN, University of Alberta, Canada
TSE-HSUN (PETER) CHEN, Concordia University, Canada
SHAOWEI WANG, University of Manitoba, Canada

Automatic software fault localization plays an important role in software quality assurance by pinpointing faulty locations for easier debugging. Coverage-based fault localization is a commonly used technique, which applies statistics on coverage spectra to rank faulty code based on suspiciousness scores. However, statistics-based approaches based on formulae are often rigid, which calls for learning-based techniques. Amongst all, *Grace*, a graph-neural network (GNN) based technique has achieved state-of-the-art due to its capacity to preserve coverage spectra, i.e., test-to-source coverage relationships, as precise abstract syntax-enhanced graph representation, mitigating the limitation of other learning-based technique which compresses the feature representation. However, such representation is not scalable due to the increasing complexity of software, correlating with increasing coverage spectra and AST graph, making it challenging to extend, let alone train the graph neural network in practice. In this work, we proposed a new graph representation, *DepGraph*, that reduces the complexity of the graph representation by 70% in nodes and edges by integrating the interprocedural call graph in the graph representation of the code. Moreover, we integrate additional features—code change information—into the graph as attributes so the model can leverage rich historical project data. We evaluate *DepGraph* using Defects4j 2.0.0, and it outperforms *Grace* by locating 20% more faults in Top-1 and improving the Mean First Rank (MFR) and the Mean Average Rank (MAR) by over 50% while decreasing GPU memory usage by 44% and training/inference time by 85%. Additionally, in cross-project settings, *DepGraph* surpasses the state-of-the-art baseline with a 42% higher Top-1 accuracy, and 68% and 65% improvement in MFR and MAR, respectively. Our study demonstrates *DepGraph*'s robustness, achieving state-of-the-art accuracy and scalability for future extension and adoption.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Fault Localization, Debugging, Graph Neural Networks

ACM Reference Format:
Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE, Article 86 (July 2024), 23 pages. https://doi.org/10.1145/3660793

1 INTRODUCTION

Locating and fixing software faults is a time-consuming and manual-intensive process. A prior study [15] found that more than 70% of the software development budgets are for software testing

Authors' Contact Information: Md Nakhla Rafi, Concordia University, Montreal, Canada, nakhla054@gmail.com; Dong Jae Kim, DePaul University, Chicago, USA, djaekim086@gmail.com; An Ran Chen, University of Alberta, Edmonton, Canada, anran6@ualberta.ca; Tse-Hsun (Peter) Chen, Concordia University, Montreal, Canada, peterc@encs.concordia.ca; Shaowei Wang, University of Manitoba, Winnipeg, Canada, Shaowei.Wang@umanitoba.ca.

ACM 2994-970X/2024/7-ART86
https://doi.org/10.1145/3660793

and debugging. As software projects continue to grow in complexity and scale, the need for efficient and accurate fault localization techniques further increases. Hence, to assist developers and reduce debugging costs, researchers have proposed various fault localization techniques [2, 23, 26, 30, 38, 40] to expedite the debugging process.

Traditional fault localization techniques, such as spectrum-based fault localization (SBFL), analyze the coverage of the passing and failing test cases to identify the potential locations of the fault that triggers the test failure. SBFL techniques are based on the intuition that a code element that is covered by more failing and fewer passing test cases is more likely to be faulty. In past decades, researchers have proposed different SBFL techniques, such as *Ochiai* [1], by crafting various formulas to rank the code elements based on the test coverage results. However, one major limitation is that such formulas may not generalize well to various faults and projects [22, 48, 52, 63].

Due to recent advances in machine learning and deep learning, there is a surge in learning-based fault localization techniques [23, 24, 26, 40, 57, 58]. These techniques enhance SBFL's potential by training models to rank the likelihood of faulty code elements, which often provide better accuracy and generalizability compared to traditional techniques. Learning-based techniques can combine various metrics in addition to test coverage to improve fault localization accuracy. For instance, *FLUCCS* [40] and *DeepFL* [23] incorporate both test coverage and code structure metrics into training deep learning models for fault localization, and showed significant improvement over traditional SBFL techniques.

In recent years, Graph Neural Networks (GNN)-based fault localization techniques have shown promising results. GNN-based techniques [30, 38] represent the code structure as a graph, in which the coverage information is represented as edges between different types of nodes (e.g., method nodes or line nodes). For example, as one of the earliest works, *Grace* [30] uses test cases and nodes in the abstract syntax tree (AST) as nodes in the graph representation. *Grace* constructs edges between test nodes and statement nodes to represent the dynamic coverage information in the graph. However, there are limitations in this graph representation. By only examining the test coverage and AST nodes, the graph representation is missing the caller-callee information. Moreover, the graph only contains structural code information, but as shown in previous studies [9, 40], historical code evolution information can also be valuable for fault localization.

In this paper, we introduce *DepGraph*, a novel GNN-based fault localization technique integrating interprocedural method calls and historical code evolution in the graph representation. In particular, *DepGraph* leverages a *Dependency-Enhanced Coverage Graph* to enhance the code representation. We first constructed a unified graph representation based on the code structure, interprocedural method calls, and test coverage. Different from prior graph representations, *dependency-enhanced coverage graph* is able to eliminate edges among methods that do not have call dependencies; thus, reducing noise in the graph. We then add code churn as attributes to the nodes in the graph, providing historical code evolution information to the GNN.

We evaluated *DepGraph* on the widely-used Defects4j (V2.0.0) benchmark [19], which contains 675 real-world faults from 14 open-source Java projects. Our results show that *DepGraph* outperforms both *Grace* and *DeepFL*, the state-of-the-art learning-based fault localization technique [23, 30]. Compared to *Grace*, $DepGraph_{w/o\ Code\ Change}$ (equivalent to *Grace* + *dependency-enhanced coverage graph*) locates 13% more faults at Top-1 and achieves over 40% improvements in both Mean First Rank (MFR) and Mean Average Rank (MAR). By further integrating code change data (i.e., code churn and method modification count) as additional features, *DepGraph* can locate 23 additional faults within Top-1 compared to $DepGraph_{w/o\ Code\ Change}$. This underscores the significance of refining graph representation and combining it with information from software development history. Furthermore, we evaluated the computing resources that can be reduced by adopting the *dependency-enhanced coverage graph*. Our results show that *Dependency-Enhanced Coverage Graph* dramatically reduces

the graph's size by 70% and minimizes GPU memory consumption by 44%, showing potential directions on adopting GNN-based techniques on larger projects. Finally, in cross-project settings, *DepGraph* significantly surpasses *Grace*, demonstrating a 42% improvement in Top-1 accuracy.

The paper makes the following contributions:

- We proposed a novel GNN-based technique, *DepGraph*, which incorporates call dependency and code evolution information in the graph representation of a project. Our findings show that *DepGraph* improves Top-1, MFR, and MAR by 20%, 55%, and 52%, respectively, compared to *Grace* [30].
- Compared to $DepGraph_{w/o\ Code\ Change}$, adding code change information further improve Top-1 by 7%. Future studies should consider integrating additional information into the graph to improve fault localization results.
- Adopting the *Dependency-Enhanced Coverage Graph* helps reduce both GPU memory usage (from 143GB to 80GB) and training/inference (from 9 days to 1.5 days). Future studies should consider compacting the graph representation to reduce the needed resources to train and run GNN-based FL techniques.
- Adopting *dependency-enhanced coverage graph* and code change information can help locate 10% to 26% additional faults compared to *Grace*, without missing the faults that *Grace* could locate. We also find that the additional faults that *DepGraph* can locate are related to the method interactions, loop structures, and call relationships, which further shows the importance of our *dependency-enhanced coverage graph*.
- In cross-project settings, *DepGraph* and $DepGraph_{w/o\ Code\ Change}$, significantly outperform *Grace*, with $DepGraph_{w/o\ Code\ Change}$ showing a 23% increase in Top-1 accuracy and *DepGraph* achieving a even higher improvement at 42%. *DepGraph* trained in cross-project settings has similar or even better results compared to the baseline techniques that are trained using data from the same project.

Paper Organization. Section 2 discusses related work. Section 3 provides a motivating example. Section 4 describes our technique, *DepGraph*. Section 5 presents the experiment results. Section 6 discusses the threats to validity. Section 7 concludes the paper.

2 RELATED WORK

Spectrum-based fault localization. Spectrum-based fault localization (SBFL) [1, 2, 18, 46] leverages statistical formulas to compute the suspiciousness of each code element (e.g., method) based on the test results and program execution. The intuition behind SBFL is that the code elements covered by more failing tests and fewer passing tests are more likely to be faulty. While SBFL has been widely studied, their effectiveness is still limited in practice [20, 53]. Several prior studies [9, 10, 45, 55] have proposed leveraging new information, such as code changes [9, 45] or mutation information [10, 55], to improve the accuracy of SBFL. However, the computed suspiciousness scores still heavily rely on code coverage and may be generalized well to other faults or systems.

Learning-based fault localization. Recently, there has been extensive research effort on leveraging learning-based methods to enhance the capabilities of SBFL [23, 24, 26, 40, 57, 58]. These techniques learn and derive the suspiciousness scores by learning from historical faults. Researchers have proposed using various machine learning techniques for fault localization, such as using radial basis function networks [47], back-propagation neural networks [49], multi-layer perceptron neural networks [58], and convolutional neural networks [3, 26, 58]. Some learning-based FL techniques integrate the suspiciousness scores computed by existing SBFL approaches with other relevant metrics. For instance, *FLUCCS* [40] combines SBFL-derived scores with other metrics, like code complexity and code history data. *CombineFL* [63] and *DeepFL* [23] combine features from diverse

dimensions, including spectrum-based, mutation-based [12, 32, 36], and information retrieval [59] scores to enhance fault localization accuracy. Le et al. [21] implemented a multi-modal technique that integrates information retrieval with program spectra for fault localization.

Due to recent advances in graph neural networks (GNNs), researchers have proposed using graphs to represent source code for learning-based fault localization [30, 38, 39, 54]. Xu et al. [54] apply GNNs to capture the source code context, representing fault subtrees as directed acyclic graphs for detecting prediction. *AGFL* [39] employs vector transformations of the abstract syntax tree (AST) nodes to represent structural code information as graphs. *Grace* [30] constructs the nodes and edges in the graph using test cases and code nodes from the program. Then, *Grace* combines the constructed graph with test coverage, adding dynamic test execution information to the graph. *GNET4FL* [38] follows a similar code representation but uses the test results directly as node attributes in the AST nodes and uses GraphSAGE [16], a more advanced GNN architecture. *GMBFL* [50] combines graph representation learning with Mutation-Based Fault Localization methods, utilizing a Gated Graph Attention Neural Network (GGANN) to extract features from the graph for identifying faulty program entities.

Although GNN-based FL techniques have shown promising results compared to other FL techniques, there are some limitations in prior studies. The graph representation in prior studies may not accurately represent the code structure, and the information in the graph is only constrained to using code structure or test execution. Hence, in this paper, we propose *DepGraph*, which integrates interprocedural call dependency graph to enhance the code representation and adds software evolution information (i.e., code changes) to the nodes in the graph. Our findings show that *DepGraph* can improve the state-of-the-art GNN-based techniques (i.e., *Grace*) by over 50% in mean average rank and in mean first rank, as well as improving the accuracy of identifying the most relevant faults at the top. In addition, our graph representation can reduce the computing resources by over 44% (total GPU memory reduced from 143GB to 80GB) and the training time by over 80% (from 9 days to 1.5 days). Our findings open up potential future directions and highlight the importance of code representation and adding additional information to the graph.

3 MOTIVATION

In this section, we highlight the limitations of an existing graph representation of the code employed by existing GNN-based FL techniques [30, 38]. In *Grace* [30], the code is first represented as many small graphs, where the root nodes are the MethodDeclaration nodes in the abstract syntax tree (AST) for both source code methods and test cases. Other nodes in a graph are the remaining AST nodes in the method/test case, such as IfStatement or ReturnStatement, connected by edges that represent the dependencies of the AST nodes. The graph is then integrated with the test coverage result, where the root nodes (i.e., methods or test cases) are linked to form a larger graph if they have a coverage relationship in test execution. Using such a code representation for training GNN models overcomes the limitations associated with existing fault localization approach in several ways: (1) It overcomes rigid formula-based (e.g., *Ochiai*) SBFL technique by automatically learning to localize faults [63], (2) it overcomes some learning-based techniques like *DeepFL* [23] that pre-processes rich coverage information into single vector, which loses topological coverage relationships [30], and finally, (3) it also preserves rich AST information of the code, thereby preserving the structural integrity of the code.

While the graph representation in *Grace* has achieved state-of-the-art results, there are still several challenges with the current approach. We use a real bug, Lang-62, from the widely-used benchmark Defects4J (V2.0.0) [19] for illustration. Figure 1 shows Lang-62's test code and source code, and their corresponding coverage and call graph representations. Firstly, as shown in Figure 1, the graph representation of code coverage only depicts the test-to-source method edge relationship,

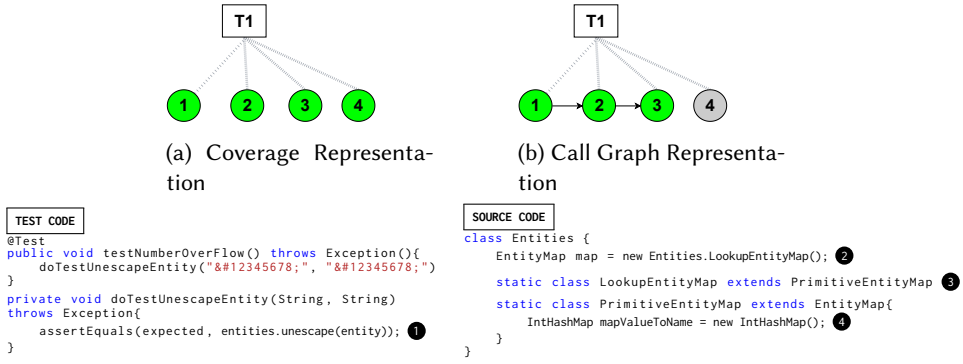

Fig. 1. Comparing graph representation for coverage and call graph for Lang-62. The green nodes correspond to the code statements (1–4). ***(a)*** shows a coverage graph linking tests to source code statements without considering the method-level call graph. ***(b)*** considers the method-level call graph, eliminating node 4 since it was not reachable in the call graph.

lacking the source-to-source edge relationship (i.e., the call graph relationship among source code methods). This omission may result in the loss of critical information about fault propagation in the source code, which could be helpful for enhancing fault localization results. Secondly, representing the entire code coverage as a graph may introduce inaccurate information into the candidate list for fault localization. For example, as shown in Figure 1, while the test case *"testNumberOverflow"* covers all source code entities (1-4), if we look at the source code, there is an absence of call relationship from (3) to (4). Although (4) is covered and an *IntHashMap* object is initiated, there is no evidence that the object is invoked during execution. Consequently, code elements like *"IntHashMap"* might be incorrectly included in the graph, even if they are not pivotal to the fault[1].

The motivating example shows that test coverage alone does not accurately portray the actual method calls within a program. Including such nodes and edges in the graph representation will increase complexity and noise in the graph representation learning process, making the GNN model less accurate and longer to train. Moreover, in addition to the structural information in the code, prior studies have found that historical information (e.g., code churn) [9, 40, 45] has a statistically significant relationship with faults. Hence, incorporating code change metrics may further enrich the graph representation by providing the evolution aspect of the code. We hypothesized that constructing the coverage graph based on test-to-source method relationships derived from the static call graph and incorporating code evolution information in the graph can further enhance GNN-based fault localization results. Below, we discuss our approach, *DepGraph*, in detail.

4 APPROACH

We present *DepGraph*, a novel fault localization technique based on graph neural networks. Figure 2 presents an overview of *DepGraph*. We implement *DepGraph* and conduct the experiment at the method-level by following prior studies [23, 30, 38], since debugging faults at the class level lack precision for effective localization [20] and pinpointing at the statement level can be overly detailed and miss important context [37]. Given the source and test code of a project, *DepGraph* constructs the abstract syntax tree (AST) for all the source methods. Then, *DepGraph* constructs an inter-procedural call graph to capture the relationships between different method calls. The AST and the call graph are then merged to form a graph representation of the code. After getting the code coverage, we integrate the information on the covered code statement into the graph, pruning the

[1]We provide the details of such statistics in our online repository [4]

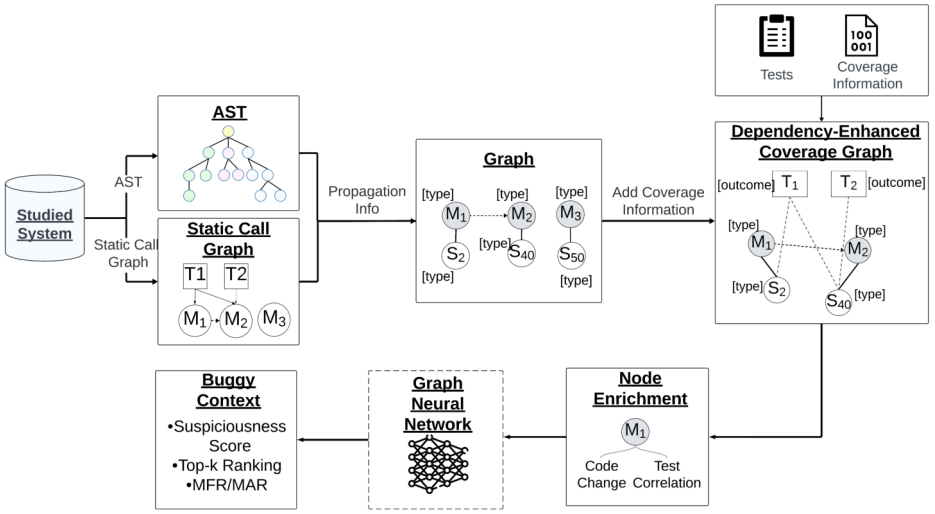

Fig. 2. Overview of *DepGraph*. The term type denotes the AST node's type (e.g., M1 is a MethodDeclaration, and S2 is an IfStatement). Tests, such as T1, include test outcomes (e.g., pass or fail).

AST vertices and the related methods that are not covered in the tests. To enhance the information in the graph and better identify faulty methods, we further integrate additional information (e.g., code churn) into the graph. Finally, the graph is fed to a Gated Graph Neural Network (GNN) model for training to locate potential faulty methods. Below, we discuss *DepGraph* in detail.

4.1 Dependency-Enhanced Coverage Graph

We represent the code, its call graph, and code coverage as a *Dependency-Enhanced Coverage Graph*. As illustrated in Figure 3, we represent the source methods, statements, and individual test methods as vertices (also referred to as nodes). The nodes are then connected by three different types of edges: 1) code edges, which connect method nodes to statement nodes and among the statement nodes; 2) method call edges, where there is a call dependency between two methods; and 3) coverage edges, where individual methods and their corresponding statements are connected to the tests that cover them. Below, we discuss our graph construction steps.

4.1.1 Source Code Graph Construction From Abstract Syntax Trees. *DepGraph* first constructs a graph representation of the code based on the Abstract Syntax Tree (AST) by following prior work [27, 28, 30, 44], which found that AST is an effective representation of code as inputs for graph neural network. Let $G = (V, E)$ be the graph representation of a program p under a test method $t \in T$, where T is the entire test suite, and V represents methods and E represents edges between V. For each V in program p, we derive an AST, denoted V_{ast}. The nodes in these ASTs typically include constructs such as MethodDeclaration, IfStatement, ReturnStatement, and VariableDeclaration, among others. However, not every node in the AST is of equal significance for fault localization. The inclusion of token-level vertices, which represent the finest granularity elements in the AST (e.g., individual operators, literals, and identifiers), introduces overhead without adding considerable information. Thus, we refine the AST by removing these token-level nodes and their corresponding edges, preserving only statement and method declaration-level nodes. This is represented as: $G_{AST'} = G_{AST} - G_{ASTtoken} = (V_{AST'}, E_{AST'})$, where $V_{AST'}$ represents set of $\{V_{MethodDeclaration}, V_{ASTstatement}\}$, and $E_{AST'}$ represent edges between $V_{AST'}$. Every method is translated into a root node, $v_m \in V_M$,

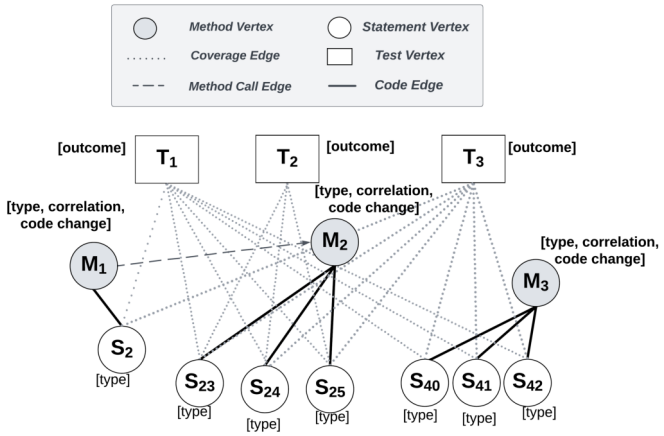

Fig. 3. An example of the *Dependency-Enhanced Coverage Graph* representation in *DepGraph*.

which represents the method's *MethodDeclaration* node. Under this root node, statement nodes $v_s \in V_S$ are structured to capture the code's hierarchy.

More formally, for each method, we present $V_{MethodDeclaration}$ as the root node, and we represent edge $e \in E$ such that $e : V_{ASTstatement} \rightarrow V_{MethodDeclaration}$. This implies that statement nodes are directly associated with their corresponding method node. Moreover, statement nodes are also interlinked with other statement nodes based on the hierarchical and sequential nature of the AST. This can be represented as: $e' : v_{s_i} \rightarrow v_{s_j}$, where v_{s_i} and v_{s_j} are distinct nodes in V_S, and e' signifies their connection.

The above-mentioned representations ensure that the constructed graph captures the hierarchical structure of the code through the AST while maintaining the semantic structures and dependencies between different parts of the code. Figure 4 gives an example where a method named `unescape` is transformed to a graph representation following the above technique. This statement-level AST representation, as opposed to individual AST nodes, significantly reduces the number of nodes in the AST, hence reducing the amount of memory consumption. Finally, for each $V_{ASTstatement}$, we assign a set of attributes denoted as $attr(V_{ASTstatement})$, which consists of the preserved AST node types. To determine these types, we adopt classifications from Javalang [42]. This approach recognizes 13 distinct node types, including common constructs like IfStatement and ReturnStatement. Recognizing and categorizing these syntactic constructs provides critical insights, particularly useful in tasks like fault localization. The list of the preserved nodes is publicly available online [4].

4.1.2 Enhancing the Graph with Interprocedural Call Graph Analysis. To pinpoint the faulty locations, it is essential to understand the flow of execution and the relationships between different parts of the code, as this information is crucial in identifying where faults may originate. While the AST offers a structured representation of the code, it has limitations in capturing comprehensive interactions between different methods. Consequently, it provides an incomplete view of the program's semantic flow. For example, as shown in Figure 3, ASTs only represent code structures, they cannot identify the flow of execution, such as whether there exists a method call between M2 and M3. To address this limitation, we introduce interprocedural call graph analysis, which gives a clearer picture of the calling dependencies among methods. This analysis extends the structural representation provided by ASTs to show how methods interact.

To better illustrate this analysis, let us define the interprocedural call graph as $G_{call} = (V_{call}, E_{call})$. In this graph, V_{call} represents all the methods, and E_{call} shows the static method calls. If method

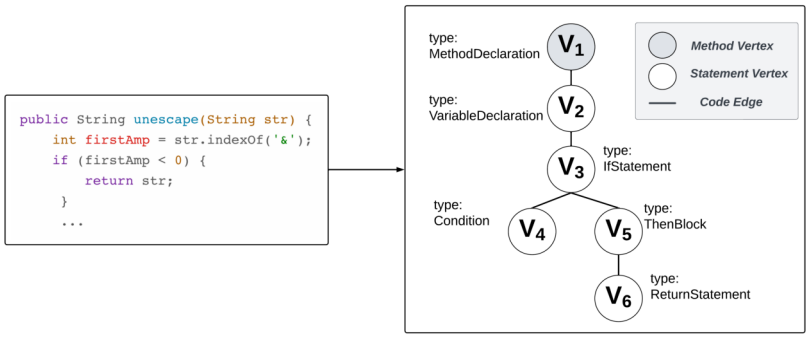

Fig. 4. Code Representation for the method unescape() based on the abstract syntax tree.

v_{m1} calls method v_{m2}, we represent this with an edge: $e_{call} = \langle v_{m1}, v_{m2} \rangle$, where $e_{call} \in E_{call}$. This captures how methods interact, both within and between classes. We further integrate the interprocedural call graph with the AST-based graph, resulting in the combined graph:

$$G_{integrated} = (V_{AST'} \cup V_{call}, E_{AST'} \cup E_{call}),$$

which merges the detailed structure from the ASTs with the method calls from the static call graph. As shown in Figure 3, the integrated graph shows that there is no method call between M2 and M3. Hence, the interprocedural call graph provides a better understanding of how different methods interact with each other.

4.1.3 Enhancing Static Graph with Dynamic Code Coverage Information. Given the integrated graph ($G_{integrated}$) from the previous step, we further integrate the dynamic code coverage information: test case T and the statements (i.e., V_S) covered by T. For a method m in the faulty program and the test suite T, let $C[m, t]$ denote the set of statements in m that are covered by test t, where $t \in T$. We denote code coverage through a set of edges, E_{cov}, between the code statement nodes V_S and test nodes V_T which can be expressed as: $E_{cov} = \{\langle v_s, v_t \rangle | s \in C[m, t], t \in T\}$, where v_s corresponds to the code statement s within V_S, and v_t represents the test t in V_T. As an example, as shown in Figure 3, test node T1 is connected to statement nodes S2, S23, S24, S25, S40, S41 and S42, which shows that T1 covers these statements from methods M1, M2, and M3. Similarly, test node T2 is connected to the statement nodes S23, S24, and S25 from method M2.

To provide further context, we associate each test node v_t with an attribute, attr(v_t), indicating its outcome (i.e., pass (✓) or fail (×)). Moreover, following the technique from *Grace*, to scale the graph, we exclude nodes v_m and v_s that are not covered by the failing tests. However, as discussed in Section 3, it still does not scale well due to the size of the coverage spectra, i.e., which contains methods unhelpful for fault localization. Hence, we use interprocedural static call graph from Section 4.1.2 to prune the graph by further verifying the static reachability of method calls $e_{call} = \langle v_{m1}, v_{m2} \rangle$, where $e_{call} \in E_{call}$. The final unified coverage is defined as, $G_{integrated} = (V_{AST} \cup V_{call} \cup V_T \cup V_S, E_{AST} \cup E_{call} \cup E_{cov})$.

4.2 Enhancing the Dependency-Enhanced Coverage Graph with Additional Graph Attributes

One major advantage of graph neural networks is their ability to integrate and process node attributes, thereby enriching the model's understanding and improving its learning capabilities [51]. By including these additional attributes in the unified coverage graph, we aim to enhance the precision and efficacy in pinpointing faulty statements. We utilize attributes like Test Correlation,

which captures textual similarities between method names and failed test cases, and Code Change Information, providing insights into code modifications and their historical context. Below, we provide a detailed description of these attributes.

Test Correlation. Test correlation information has been widely recognized in previous studies as a valuable feature for enhancing fault localization [23, 30, 60]. Test correlation is based on the idea that methods that have a greater textual similarity with the failed test method are more likely to be faulty. We measure the textual similarity between the names of the source method and test method using the Jaccard distance [35] by following prior work [23, 30]. We split the words based on camel cases as a preprocessing step prior to computing the similarity score. More formally, given method name (w_m) and failed test method name (w_t), we define the similarity as: $Similarity(w_m, w_t) = \frac{length(w_m \cap w_t)}{length(w_t)}$. The equation finds the ratio between the overlapping word tokens ($length(w_m \cap w_t)$) and the total number of tokens in the failed test name ($length(w_t)$). When multiple failed tests are present, it greedily selects the test method with the highest similarity score.

Code Change Information. Prior studies [9, 40, 45, 62] found that code changes provide valuable insights into fault proneness. Therefore, to improve the effectiveness of fault localization, we incorporate two metrics based on code changes: *Code Churn* and *Method Modification Count (MMC)*. Based on earlier research [33, 62], we define *Code Churn (CHURN)* as the total net change in lines (i.e., both additions and deletions) for a particular method over a given duration. Meanwhile, the *Method Modification Count (MMC)* is determined by the number of modifications made to the method within the same time frame. We calculate each metric using two-time intervals: (1) the entire method history and (2) recent history (six months before the faulty commit, as suggested by Zimmermann et al. [62]), and incorporate them into our graph. To obtain detailed change information, we employ the *git diff* command, capturing the code differences between the faulty commit and its preceding commits. To achieve a more detailed analysis, we identify all changed methods from the commits. Using JavaLang [42], we produce an AST for each method on our candidate list. From these ASTs, we extract the beginning and ending line numbers for every method. Subsequently, we verify if any lines within these ranges changed within our selected time intervals. Then, we use this information as method node attributes.

4.3 Constructing the Graph Neural Network Model

We deploy a Gated Graph Neural Network (GGNN) to rank potential faulty methods. The input for this model is the *Dependency-Enhanced Coverage Graph* discussed in Section 4.1, represented as $G(V, E)$, where V are the nodes and E are the edges. The nodes, V, comprise the method nodes, statement nodes, and test nodes, which can be expressed as $V = V_M \cup V_S \cup V_T$. Their interconnections are depicted by the adjacency matrix A. A connection between node V_i and V_j is denoted with $a_{ij} = 1$. The absence of a connection is marked as $a_{ij} = 0$. For the method nodes, each v in V carries an attribute $\gamma_v \in \Upsilon$ which includes node type (e.g., `If Statement`), Test correlation information, *Code churn* values, *Method modification* counts. On the other hand, the test vertices contain a test outcome attribute, indicating whether the test passed or failed. To ensure stable model performance, we normalize A with the formula $\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, where D represents a diagonal matrix. The detailed attribute sequence Υ along with the adjusted adjacency matrix $\hat{A}$ are then channeled as primary inputs into the GGNN for the fault detection process.

Gated Graph Neural Network (GGNN). The Gated Graph Neural Network (GGNN) [25], an advanced variant of GNNs, incorporates gating mechanisms such as Gated Recurrent Unit (GRU) [11] and Long Short-Term Memory (LSTM) [14] to capture intricate patterns over extended sequences. In *DepGraph*, we leverage a GGNN framework to distill essential features from the comprehensive coverage graph in order to localize suspicious statements.

Embedding Layer. The embedding layer is designed to transform raw data attributes, represented by the sequence Υ, into a structured and enriched feature matrix, $\mathcal{X}$, of size $R^{|V|\times d}$, with d being the embedding dimension. This transformation aids in offering a better representation of the model's understanding. Test nodes and non-root code nodes are directly transformed into d-dimensional vectors. For primary code nodes that have more complex attributes, such as the AST node type, a two-step process is adopted. Initially, the AST node type is converted into a $d-1$-dimensional vector. This vector is then concatenated with the test correlation and code change information to complete its representation in the d-dimensional space. This procedure ensures a uniform d-dimensional representation for all vertices within the attribute matrix.

GGNN Iterative Process. Following a prior work [30], we apply five iterations in the GGNN layer. This process iteratively refines node representations, enhancing the model's ability to detect patterns and relationships, ultimately aiming for improved results. During the t^{th} iteration, every node updates its current state by incorporating information from both its neighboring vertices and the outcomes of previous iterations. The gated mechanism in the GGNN is implemented by utilizing the input and forget gates of the LSTM, guiding the propagation of cell states. Specifically, the cell state of node v in the t^{th} iteration is represented as $c_v^{(t)}$, and is initialized as $c_v^{(1)} = X_v$. The propagation of cell states from adjacent vertices in the previous iteration is described by:

$$a_v^{(t)} = \hat{A}^v : [c_1^{(t-1)T}; \ldots; c_{|V|}^{(t-1)T}]$$

The forget gates determine which portions of information to exclude from the cell states, while input gates dictate the new information from the current input $a_v^{(t)}$ to be integrated into the cell states. To counteract the vanishing gradient issue, we use the residual connections [17] with layer normalization [6] implemented between each pair of sub-layers.

Inference and Loss Functions in GGNN Analysis. After processing the graph through the GGNN, our primary goal is to deduce a suspiciousness score for each code statement. The transformation layers, specifically the softmax function, translate the refined node features into these scores. At a high level, we process the outcomes of the GGNN to generate a suspiciousness score, ranging between 0 and 1, for each candidate method. This score essentially quantifies the likelihood of a method being faulty. The last layer of the GGNN is equipped with a linear transformation that employs the softmax function. Specifically, consider a node, v_i. After the final iteration of GGNN, its output is represented by z_i. This output is transformed linearly to yield a real number y_i', which is articulated as $y_i' = Wz_i + b$. Here, $W \in R^{d\times 1}$ is the weight matrix guiding the transformation, and $b \in R$ is the bias term. Following this, we adopt a listwise strategy to rank the nodes. The outputs for all nodes undergo normalization using the softmax function, resulting in:

$$p(v_i) = \frac{exp\{y_i'\}}{\sum_{j=1}^{n} exp\{y_j'\}}$$

This gives $p(v_i)$ as the probability score that indicates the likelihood of the node v_i being associated with a fault. When it comes to defining the ranking loss function, our tool, *DepGraph*, leans heavily towards the listwise approach. It assesses lists as cohesive entities based on the arrangement of their elements. This approach resonates well with *DepGraph*'s philosophy of treating elements and their interrelationships as a coherent whole. The corresponding listwise loss function is:

$$L_{list} = -\sum_{i=1}^{n} g(v_i)\log(p(v_i))$$

Wherein, $g(v_i)$ represents the ground truth label for each method node $v_i \in V_M$, defined as 1 if v_i is faulty, and 0 otherwise.

Table 1. An overview of our studied projects from Defects4J v2.0.0. *#Faults*, *LOC*, and *#Tests* show the number of faults, lines of code, and tests in each system. *Fault-triggering Tests* shows the number of failing tests that trigger the fault.

Project	#Faults	LOC	#Tests	Fault-triggering Tests
Cli	39	4K	94	66
Closure	174	90K	7,911	545
Codec	18	7K	206	43
Collections	4	65K	1,286	4
Compress	47	9K	73	72
Csv	16	2K	54	24
Gson	18	14K	720	34
JacksonCore	26	22K	206	53
JacksonXml	6	9K	138	12
Jsoup	93	8K	139	144
Lang	64	22K	2,291	121
Math	106	85K	4,378	176
Mockito	38	11K	1,379	118
Time	26	28K	4,041	74
Total	675	380K	24,302	1,486

5 STUDY DESIGN AND RESULTS

In this section, we first describe the study design and setup. Then, we present the motivation, approach, and results of the research questions.

Benchmark Dataset. To answer the RQs, we conducted the experiment on 675 faults from the Defects4J benchmark (V2.0.0) [19]. Defects4J provides a controlled environment to reproduce faults collected from projects of various types and sizes. Defects4J has also been widely used in prior automated fault localization research [9, 30, 40, 56]. We excluded three projects, JacksonDatabind, JxPath, and Chart, from Defects4J in our study since we encountered many execution errors and were not able to collect the test coverage information. Table 1 gives detailed information on the projects and the faults that we use in our study. In total, we conducted our study on 14 projects from Defects4J, which contains 675 unique faults with over 1.4K fault-triggering tests (i.e., failing tests that cover the fault). The sizes of the studied projects range from 2K to 90K lines of code. Note that, since a fault may have multiple fault-triggering tests, there are more fault-triggering tests than faults.

Evaluation Metrics. According to prior findings, debugging faults at the class level lacks precision for effective location [20]. Alternatively, pinpointing them at the statement level might be overly detailed, omitting important context [37]. Hence, in keeping with prior work [5, 7, 23, 30, 43], we perform our fault localization process at the method level. We apply the following commonly-used metrics for evaluation:

Recall at Top-N. The Top-N metric measures the number of faults with at least one faulty program element (in this experiment, methods) ranked in the top N. In other words, the Top-N metric identifies the methods that are most relevant to a specific fault among the top-ranked N methods. The result from *DepGraph* is a ranked list based on the suspiciousness score. Prior research [37] indicates that developers typically only scrutinize a limited number of top-ranked faulty elements. Therefore, our study focuses on Top-N, where N is set to 1, 3, 5, and 10.

Mean Average Rank (MAR). For each faulty version, the Mean Average Rank (MAR) measures the average position of all faulty methods in a list. For each project, MAR finds the average of these positions across all its faulty versions. The lower the value, the better the ranking.

Mean First Rank (MFR). This metric determines the position of the first identified fault in the ranked list for every faulty version. The MFR for a project is then derived by taking the average of these positions across all of its faulty versions. A lower MFR means the faulty methods can be identified earlier in the list.

Implementation and Environment. To gather information about test coverage and calculate the result for baseline techniques, we use Gzoltar [8]. Gzoltar is a library for automatic debugging of Java applications. It provides techniques for test suite minimization and fault localization, which can help developers identify and fix software faults more efficiently. Gzoltar can also measure the code coverage of the test suite, which indicates how much of the source code is executed by the tests. In order to generate static call graphs, we used the "Java-Callgraph" tool [13]. The tool reads classes from a jar file, walks through their method bodies, and generates a caller-callee relationship table to analyze the Java bytecode to retrieve the calling relationship among methods. We then constructed a call graph based on the calling relationship. For generating AST, we parse the source code using the JavaLang toolkit [42]. To train *DepGraph*, we use a learning rate of 0.01 and an embedding size of 32 for all projects by following prior studies [23, 30]. To prevent under-fitting and over-fitting due to too few or too many epochs, we use 10 epochs according to previous work [23]. To reduce variations across the experiments, we fixed the random seed for all the runs. All experiments are conducted on a Linux server with 120G RAM, AMD EPYC 7763 64-Core CPU @ 3.53 GHz, and a 40G NVIDIA A100 GPU. We use PyTorch 2.0.1 for training and validating the GNN model [41]. The configuration values for our experiment can be found online [4].

5.1 RQ1: What is the Effectiveness of DepGraph in Fault Localization?

Motivation. In this RQ, we evaluate the localization accuracy of *DepGraph* and the impact of leveraging inter-procedural call graph information. *DepGraph* contains two novel components: *Dependency-Enhanced Coverage Graph* and integrating additional node attributes (i.e., code change information) in the graph. Hence, we study the effect of each component separately and the combined effect on fault localization accuracy. The findings may also provide insights into the necessity of including each component in future GNN-based fault localization research.
Approach. We compare *DepGraph*'s fault localization accuracy with three baselines: ***Ochiai*** [1], ***DeepFL*** [23], and ***Grace*** [30] (which we refer to as ***GNN***).

Ochiai: Ochiai is a widely used spectrum-based fault localization technique because of its high fault localization accuracy [1, 10, 26, 30, 39, 45]. *Ochiai* is defined as:

$$Ochiai(a_{ef}, a_{nf}, a_{ep}) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{nf}) \times (a_{ef} + a_{ep})}}$$

where a_{ef} is the number of failed test cases that execute a statement, a_{nf} is the number of failed test cases that do not execute the statement, and a_{ep} is the number of passed test cases that execute the statement. Intuitively, *Ochiai* assigns a higher suspiciousness score to a statement that is executed more frequently by failing cases and less frequently by passing test cases. The result of *Ochiai* is a value between 0 and 1, with higher values indicating a higher likelihood that a particular statement is faulty. We then aggregate the scores for every statement in a method to get the method-level score.

DeepFL: DeepFL [23] is a deep-learning-based fault localization technique that integrates spectrum-based and other metrics such as code complexity, and textual similarity features to locate faults. It utilizes a Multi-layer Perceptron (MLP) model to analyze these varied feature dimensions. We follow the study [23] to implement *DeepFL* and include the SBFL scores from 34 techniques, code complexity, and textual similarities as part of the features for the deep learning model.

GNN (i.e., Grace): Grace is one of the first fault localization techniques based on graph neural networks (GNN) [30] and can be viewed as a direct adoption of GNN. Since *DepGraph* is also based on *GNN/Grace*, we study the effectiveness of different components in *DepGraph* over *Grace* and refer to *Grace* as *GNN*. *GNN* represents test cases as test nodes in the coverage graph and establishes

edges between these test nodes and statement nodes based on test coverage. However, compared to *DepGraph*, *GNN's* graph representation misses call graph info and direct edges between method nodes, which could capture fault propagation. This omission means that *GNN* does not capture potential fault propagation paths between methods, so the candidate method lists produced by *GNN* might include some unrelated methods as discussed in Section 3. Since *DepGraph* can be viewed as an enhanced version of *GNN* (incorporating a *dependency-enhanced coverage graph* and additional node attributes), our comparison with *GNN* effectively serves as an ablation study.

DepGraph: In addition to studying the overall fault localization accuracy of *DepGraph*, we also construct sub-models to understand the effect of the enhanced graph and node attributes. Specifically, we build two versions of *DepGraph*: 1) $DepGraph_{w/o\ Code\ Change}$, which is *GNN + Dependency-Enhanced Coverage Graph* without the code change attributes; and 2) *DepGraph*, which incorporates both *Dependency-Enhanced Coverage Graph* and code change information.

Evaluating the Models: We follow prior studies [23, 30] and evaluate *DeepFL*, *GNN*, and *DepGraph* using leave-one-out cross-validation [23, 30]. Given one particular fault, we train the model using all other faults and evaluate the model using that one fault. The process is repeated for all the faults in a project.

Results. ***Across all the studied projects, $DepGraph_{w/o\ Code\ Change}$ provides 13%, 46%, and 42% improvement on Top-1, MFR, and MAR, respectively, when compared to GNN.*** Table 2 presents the fault localization result of *DepGraph*, $DepGraph_{w/o\ Code\ Change}$, and the baseline techniques. Out of 675 faults, $DepGraph_{w/o\ Code\ Change}$ identifies 336 faults in Top-1, which is 13% more than *GNN* (298 faults), 31% more than *DeepFL* (257), and, 177% more than *Ochiai* (121 faults). $DepGraph_{w/o\ Code\ Change}$ also generates an effective ranked list of potential faulty methods, capable of identifying 588/675 (87%) of the faults in the Top-10. We see a larger improvement in MFR and MAR when comparing $DepGraph_{w/o\ Code\ Change}$ to *GNN*, where the total MFR and MAR are improved by 46% and 42%, respectively. The result suggests that the *Dependency-Enhanced Coverage Graph* is effective in promoting faulty methods in the ranked list. Similarly, when comparing $DepGraph_{w/o\ Code\ Change}$ to *DeepFL*, there is a significant improvement in both MFR and MAR (65% and 62% better, respectively). Among the three techniques, *Ochiai* has the worst MFR and MAR (20.75 and 24.04), which shows that GNN-based techniques are more effective than traditional techniques such as *Ochiai*.

Across all the studied projects, incorporating the code change information to DepGraph improves the overall Top-1, MFR, and MAR by 7%, 16%, and 17%, respectively, compared to $DepGraph_{w/o\ Code\ Change}$. Although the relative improvement is less for Top-3. Top-5, and Top-10 (2.3%, 1.3%, and 1.5%, respectively), code change information is more effective at improving Top-1. *DepGraph* can identify 23 more faults in Top-1, representing a 7% improvement. Having faulty methods ranked earlier in the list is essential for developers' adoption of fault localization techniques [37], which further shows the importance of adding code change information. When looking at individual projects, adding code change information, in general, also gives the best results in all the studied projects. One exception is the Codec, where we see a decrease in Top-3, Top-10, MFR, and MAR after adding code change information. The reason may be that the number of faults in Codec is relatively small (18 faults), so missing one fault causes larger fluctuations in the localization result. Our findings show that code representation and the information in the graph play a significant role in improving the fault localization result. Future studies should adopt enhanced graph representation of the code, and consider combining SBFL techniques with other valuable information that can be mined from the software development history.

Table 2. A comparison of the fault localization techniques. For each project, we show the technique with the best MFR in bold (the lower the better). *DepGraph$_{w/o\ Code\ Change}$* shows the result after adopting *Dependency-Enhanced Coverage Graph*, and *DepGraph* shows the result of incorporating both *Dependency-Enhanced Coverage Graph* and Code Change Information. The number in the parentheses shows the percentage improvement over *GNN (i.e., Grace)* [30]. The best result is marked in bold.

Project (# faults)	Techniques	Top-1	Top-3	Top-5	Top-10	MFR	MAR
Cli (39)	Ochiai	3	5	10	18	15.711	18.272
	DeepFL	11	21	24	28	8.991	10.681
	GNN	14	24	26	30	7.861	9.903
	DepGraph$_{w/o\ Code\ Change}$	15 (7%)	22 (-8%)	26 (0%)	31 (3%)	5.973 (24%)	7.118 (28%)
	DepGraph	**17 (21%)**	24 (0%)	**27 (4%)**	**34 (10%)**	**5.105 (30%)**	**6.223 (32%)**
Closure (174)	Ochiai	20	39	70	72	98.652	110.348
	DeepFL	46	61	92	99	29.388	35.333
	GNN	51	78	102	121	12.854	14.814
	DepGraph$_{w/o\ Code\ Change}$	58 (14%)	97 (24%)	123 (21%)	**148 (22%)**	4.844 (62%)	7.911 (47%)
	DepGraph	**60 (18%)**	**99 (27%)**	**126 (24%)**	**148 (22%)**	**4.542 (65%)**	**7.306 (51%)**
Codec (18)	Ochiai	3	12	**17**	17	2.701	3.461
	DeepFL	5	10	12	16	2.742	4.803
	GNN	6	11	13	17	2.536	4.015
	DepGraph$_{w/o\ Code\ Change}$	**7 (17%)**	**12 (9%)**	14 (8%)	16 (-6%)	**2.412 (5%)**	**3.265 (19%)**
	DepGraph	**7 (17%)**	10 (-9%)	14 (8%)	16 (-6%)	3.111 (-23%)	4.327 (-8%)
Collections (4)	Ochiai	1	1	2	2	3.871	3.431
	DeepFL	1	1	2	2	1.512	1.519
	GNN	1	1	2	2	1.511	1.511
	DepGraph$_{w/o\ Code\ Change}$	**1 (0%)**	1 (0%)	**2 (0%)**	2 (0%)	1.511 (0%)	1.511 (0%)
	DepGraph	**1 (0%)**	**2 (100%)**	**2 (0%)**	2 (0%)	**1.445 (4%)**	**1.445 (4%)**
Compress (47)	Ochiai	5	12	17	29	20.106	23.275
	DeepFL	22	27	31	38	9.573	12.955
	GNN	23	29	34	42	5.383	6.987
	DepGraph$_{w/o\ Code\ Change}$	24 (4%)	32 (10%)	**36 (6%)**	42 (0%)	4.384 (19%)	5.209 (25%)
	DepGraph	**25 (9%)**	**33 (14%)**	**36 (6%)**	**45 (7%)**	**3.361 (38%)**	**4.245 (39%)**
Csv (16)	Ochiai	3	8	10	12	5.625	5.782
	DeepFL	7	8	9	11	5.623	5.971
	GNN	6	8	10	12	5.438	5.938
	DepGraph$_{w/o\ Code\ Change}$	**8 (33%)**	**9 (13%)**	10 (0%)	**13 (8%)**	5.362 (1%)	5.581 (6%)
	DepGraph	**8 (33%)**	**9 (13%)**	**12 (20%)**	**13 (8%)**	**4.813 (12%)**	**5.001 (16%)**
Gson (18)	Ochiai	4	9	9	12	9.177	10.183
	DeepFL	8	11	12	12	8.873	9.324
	GNN	11	13	14	15	6.471	6.755
	DepGraph$_{w/o\ Code\ Change}$	12 (9%)	14 (8%)	15 (7%)	15 (0%)	2.177 (66%)	2.471 (63%)
	DepGraph	**14 (27%)**	**15 (15%)**	**16 (14%)**	**16 (7%)**	**1.353 (79%)**	**1.765 (74%)**
JacksonCore (26)	Ochiai	6	11	13	14	9.789	16.754
	DeepFL	5	5	9	10	8.671	9.711
	GNN	9	13	14	15	6.471	6.755
	DepGraph$_{w/o\ Code\ Change}$	9 (0%)	14 (8%)	**15 (7%)**	**17 (13%)**	3.474 (29%)	4.509 (28%)
	DepGraph	**12 (33%)**	**15 (15%)**	**15 (7%)**	16 (7%)	**3.052 (38%)**	**4.015 (36%)**
JacksonXml (6)	Ochiai	0	0	0	0	59.2	59.2
	DeepFL	3	3	4	5	3.513	4.245
	GNN	3	3	4	5	2.401	2.401
	DepGraph$_{w/o\ Code\ Change}$	**4 (33%)**	**5 (67%)**	**5 (25%)**	**5 (0%)**	0.411 (83%)	0.411 (83%)
	DepGraph	**4 (33%)**	**5 (67%)**	**5 (25%)**	**5 (0%)**	**0.409 (83%)**	**0.409 (83%)**
Jsoup (93)	Ochiai	15	40	48	57	14.944	20.209
	DeepFL	33	39	46	49	10.23	11.444
	GNN	40	64	72	77	8.223	9.669
	DepGraph$_{w/o\ Code\ Change}$	50 (25%)	70 (9%)	77 (7%)	82 (6%)	4.022 (51%)	6.815 (30%)
	DepGraph	**53 (33%)**	**73 (14%)**	**78 (8%)**	**83 (8%)**	**3.023 (63%)**	**4.6174 (52%)**
Lang (64)	Ochiai	25	45	51	59	4.68	5.15
	DeepFL	42	53	55	57	2.833	3.08
	GNN	43	53	57	58	2.113	2.462
	DepGraph$_{w/o\ Code\ Change}$	45 (5%)	**55 (4%)**	58 (2%)	**61 (5%)**	1.564 (26%)	1.902 (23%)
	DepGraph	**48 (12%)**	**55 (4%)**	**60 (5%)**	**61 (5%)**	**1.153 (45%)**	**1.481 (40%)**
Math (106)	Ochiai	23	52	62	82	9.73	11.72
	DeepFL	52	81	90	95	3.95	4.911
	GNN	64	79	92	97	2.355	3.082
	DepGraph$_{w/o\ Code\ Change}$	67 (5%)	90 (14%)	96 (4%)	100 (3%)	1.185 (50%)	1.528 (50%)
	DepGraph	**72 (13%)**	**92 (16%)**	**97 (5%)**	**102 (5%)**	**1.115 (53%)**	**1.454 (53%)**
Mockito (38)	Ochiai	7	14	18	23	20.22	24.77
	DeepFL	10	18	23	26	13.541	17.001
	GNN	16	24	26	29	9.611	13.621
	DepGraph$_{w/o\ Code\ Change}$	20 (25%)	28 (17%)	**34 (31%)**	**34 (17%)**	2.361 (75%)	3.307 (76%)
	DepGraph	**21 (31%)**	**29 (21%)**	32 (23%)	**34 (17%)**	**2.194 (77%)**	**2.998 (78%)**
Time (26)	Ochiai	6	12	13	16	16.14	18.98
	DeepFL	12	15	18	20	12.722	13.754
	GNN	11	16	20	21	7.842	8.448
	DepGraph$_{w/o\ Code\ Change}$	16 (45%)	19 (19%)	20 (0%)	**22 (5%)**	3.321 (58%)	4.465 (47%)
	DepGraph	**17 (55%)**	**20 (25%)**	**21 (5%)**	**22 (5%)**	**3.044 (61%)**	**4.371 (48%)**
Total (675)	Ochiai	121	260	340	413	20.753	24.038
	DeepFL	257	353	427	468	8.726	10.338
	GNN	298	416	486	541	5.678	6.851
	DepGraph$_{w/o\ Code\ Change}$	336 (13%)	470 (13%)	534 (10%)	588 (9%)	3.049 (46%)	3.957 (42%)
	DepGraph	**359 (20%)**	**481 (16%)**	**541 (11%)**	**597 (10%)**	**2.562 (55%)**	**3.272 (52%)**

Adding the *Dependency-Enhanced Coverage Graph* to *GNN* improves Top-1 by 20% and MFR and MAR by more than 50%. Adding the code change information further improves Top-1 by 7%. Our findings emphasize improving graph representation and combining SBFL techniques with other information that can be mined from the software development history.

5.2 RQ2: How Much Computing Resource Can Be Reduced By Adopting the Dependency-Enhanced Coverage Graph?

Motivation. Due to the complexity of graph data, training and using graph neural networks (GNNs) require a significant amount of memory (e.g., need to store the entire graph in GPU memory) and computation resources [61]. In the case of source code, the graph size grows exponentially due to having various control flows, which can make a graph extremely large for real-world software, making GNNs difficult and slow to train. Prior works [30, 34, 38, 39] in graph-based fault localization only utilize AST trees to represent the code. However, as we discussed in Section 3, representing the code using only the AST trees would miss the caller-callee information among the method nodes, potentially adding unnecessary edges among the nodes. In this RQ, we study how much resources can be saved when training the model using the *dependency-enhanced coverage graph*.

Approach. To answer this RQ, we evaluated several metrics for the generated graph of each project: number of nodes and edges, training/inference time, and GPU memory usage. Our comparison centers on two distinct phases of model configuration. The 'before' phase utilizes a graph representation similar to the *Grace (e.g., GNN)* model, which notably does not include inter-procedural call graph information, and serves as the baseline in our analysis. In contrast, the 'after' phase mirrors the approach employed in *DepGraph*, where we integrate inter-procedural call graph information into the graph representation (i.e., the *Dependency-Enhanced Coverage Graph*). We measure the response time by comparing the time that we start to train the model and the time that the training is done using both the baseline and *dependency-enhanced coverage graph* representations. For GPU memory, we examine and report the memory usage when it is stabilized (e.g., once the graph is loaded in the GPU memory and the model training starts).

Result. ***Adopting the dependency-enhanced coverage graph not only improves the localization results as shown in RQ1, but it can also reduce the number of nodes and edges by 69.32% and 74.93%, respectively, and the GPU memory usage by 44%.*** Table 3 shows the graph size reduction before and after adopting the *dependency-enhanced coverage graph*. For all of the projects, the number of nodes and edges decreased significantly (69.32% and 74.93% across all projects). We find that such reduction can be even greater for larger projects. For instance, the number of nodes in the largest project Closure is around 728.8K when only using the AST information, but reduces significantly to around 137K after adopting *dependency-enhanced coverage graph* (81% reduction). There were also around 100 million edges initially and they were reduced to around 10.5 million (89% reduction). A similar trend can be observed in other large projects, such as Jsoup, where the node count is reduced from around 131K to 61K (53.31% reduction), and Math, where the reduction is even greater (74.10%). For smaller projects, the reduction in nodes and edges may be less (e.g., Codec has 25.46% and 30% reduction in nodes and edges), but overall, adopting *dependency-enhanced coverage graph* can significantly reduce graph sizes and improve localization.

The reduction in graph size also contributes to a significant improvement in GPU memory usage. We see an improvement ranging from 15% to 81% in GPU memory usage across all the studied projects. When summing up the usage across all the studied projects, adopting *dependency-enhanced coverage graph* reduces the memory usage from 143.7GB to 80GB (44% improvement). For larger projects such as Closure, we can reduce more than 10GB of GPU memory (from 35.6GB to 23.9GB) in the training process. Reducing the GPU memory usage is significant for the practicability and

Table 3. Comparisons of graph sizes, training/inference time, and GPU memory usage during training before and after adopting the *dependency-enhanced coverage graph*. The numbers in the parentheses show the percentage improvement. *Total* shows the aggregated results from all the studied projects.

Project	#Nodes		#Edges		Training(s)		Inference(s)		GPU Memory Usage(GB)	
	Before	After	Before	After	Before	After	Before	After	Before	After
Cli	17.8K	12.1K (32.31%)	782.7K	229.6K (70.67%)	2.2K	491 (79%)	118	22 (81%)	3.5	1.9 (44%)
Closure	728.8K	137.3K (81.16%)	100.2M	10.5M (89.48%)	579.6K	102K (82%)	5.2K	556 (89%)	35.6	23.9 (33%)
Codec	3.4K	2.5K (25.46%)	46.4K	32.5K (29.99%)	223	40 (82%)	14	6 (57%)	1	0.8 (20%)
Compress	30.2K	14.9K (50.65%)	876.1K	244.8K (72.06%)	2.1K	703 (67%)	53	33 (38%)	4.5	1.9 (57%)
Csv	5.8K	3.5K (39.93%)	215.0K	85.2K (60.40%)	383	172 (55%)	27	12 (56%)	2.7	0.9 (66%)
Gson	19.6K	12.6K (35.71%)	2.4M	775.9K (68.23%)	2.9K	675 (77%)	162	38 (77%)	7.4	2.9 (60%)
JacksonCore	14.7K	7.1K (51.65%)	1.4M	216.7K (84.22%)	2.2K	429 (81%)	173	62 (64%)	5.7	2.1 (81%)
JacksonXml	2.9K	1.0K (64.22%)	163.0K	23.4K (85.64%)	79	19 (75%)	12	5 (57%)	2.3	1.3 (44%)
Jsoup	131.4K	61.3K (53.31%)	22.3M	4.4M (80.29%)	137K	21K (84%)	1.3K	232 (82%)	23	11.1 (51%)
Lang	18.1K	9.3K (48.75%)	435.6K	142.2K (67.35%)	1.7K	758 (57%)	52	29 (44%)	1.8	1.2 (34%)
Math	140.9K	36.5K (74.10%)	3.3M	1.1M (67.26%)	25K	7.3K (71%)	234	95 (59%)	18	10.3 (43%)
Mockito	63.3K	28.0K (55.69%)	10.7M	521.2K (95.14%)	31.4K	1K (97%)	1K	37 (97%)	19.3	5.2 (73%)
Time	99.1K	65.3K (34.15%)	4.9M	4.2M (14.89%)	10.4K	5K (52%)	329	220 (33%)	18.5	15.7 (15%)
Total	1.27M	391.4K (69.32%)	47.54M	11.91M (74.93%)	796K	140K (82%)	9K	1.3K (85%)	143.7	80 (44%)

adoption of all GNN-based techniques since a high-end GPU such as NVIDIA A100 only has 40GB or 80GB of memory. Hence, reducing memory usage can make GNN-based techniques easier to adopt on larger projects.

The model training and inference time are reduced by 82% and 85%, respectively, after adopting the dependency-enhanced coverage graph. The training time for the GNN model depends on the size of the project. However, in our experiment, the training time is significant, with a total training time of over 9 days for all the projects (796K seconds) on an NVIDIA A100 GPU before adopting the *dependency-enhanced coverage graph*. For larger projects like Closure, the training time took almost one week. Such a long training time may hinder the adoption of GNN-based techniques in general and is not limited only to fault localization. After adopting the *dependency-enhanced coverage graph*, we noticed a significant reduction in the training time, where it took only one day to train a model for Closure (82% reduction). Across all the studied projects, we find that the total training time is reduced by 82%, where the improvement is at least 50% or more for individual projects. Although inference is faster than training, the total inference time still takes 2.5 hours (9K seconds). In particular, the inference time is 1.4 hours for Closure. After adopting the *dependency-enhanced coverage graph*, the total inference time is reduced by 85%, from 2.5 hours to 20 minutes (1.3K).

Our findings highlight the challenges in adopting GNN-based techniques (not just fault localization) in practice while also showing potential future directions. Graph neural networks (GNNs) typically compute an entire adjacency matrix along with the embedding for all nodes. This process can be notably resource-intensive, both in terms of memory and computational time, as found in our study and also highlighted in prior work [31]. In this paper, we introduced a more compact and more accurate graph representation that substantially reduces the overall size of the graph. This reduction, in turn, has led to a great decrease in both training and inference time, and memory usage. As demonstrated in RQ1, our approach does not compromise fault localization accuracy yet it actually improves the accuracy. Future studies should consider improving the practical aspects of GNN-based fault localization by further improving the graph representation and perhaps incorporating other graph pruning techniques.

Adopting the *Dependency-Enhanced Coverage Graph* helps significantly reduce the graph size by 70% and GPU memory usage by 44%. We also find that the total training time is reduced from 9 days to 1 day. Our findings highlight the computation issue with using GNN techniques for software engineering tasks in general and provide potential future research directions.

5.3 RQ3: Does DepGraph Locate Different Sets of Faults Compared to GNN?

Motivation. In this RQ, we further evaluate what kind of faults *DepGraph* is able to detect. Specifically, whether *DepGraph* can locate an additional set of faults, or does it detects new faults and misses some previously-located faults. Understanding such changes in the located faults can help us understand why our approach works and fails for certain faults and the effect of *dependency-enhanced coverage graph* and code change information. The findings may give us valuable insight into improving GNN-based FL techniques in the future.

Approach. We conduct two experiments on the two representative code representations (*Dependency-Enhanced Coverage Graph* and the representation used in GNN (i.e., Grace) [30]), and the code change information we added: (1) we analyze the degree of overlap between the faults located by different code representation at Top-N, and additionally show whether there are overlaps in the located faults; and (2) from these overlapping and non-overlapping faults, we perform an empirical study to analyze what kind of faults each code representation is capable of locating at Top-N. For a given fault, we extract their faulty statements and analyze their (a) AST node type and (b) fix information, to identify the relationship between code presentation and fault location.

Results. ***Adopting dependency-enhanced coverage graph helps locate 8.8% to 13.6% additional faults compared to GNN. By further adding code change information, we can locate 10% to 26% additional faults.*** In our analysis, we compare the faults that can be located using different graph representations: *GNN*, $DepGraph_{w/o\ Code\ Change}$ (with call dependency), and *DepGraph*. As shown in Figure 5, we observe that $DepGraph_{w/o\ Code\ Change}$ is able to locate 77 additional new faults at Top-1 that could not be located by *GNN* (14.7% new faults). At Top-3 we see that $DepGraph_{w/o\ Code\ Change}$ can still locate 73 additional faults compared to *GNN* (13.6 % new faults), 63 additional faults (9.6%) at top-5, and 54 additional faults (8.8%). These results show that as we increase Top-N, $DepGraph_{w/o\ Code\ Change}$ can locate almost all the faults that *GNN* can locate, but at the same time, $DepGraph_{w/o\ Code\ Change}$ can locate many additional faults. Similarly, when adding code change information, *DepGraph* is able to locate more faults, and at the same time, miss fewer faults that were previously detected when using *GNN*. Moreover, at Top-10 we find that *DepGraph* can locate almost all the faults that *GNN* located, except for one. The findings show the robustness of adding additional information to the graph. Adding *dependency-enhanced coverage graph* provides mainly benefits in locating additional faults and does not miss the previously-located faults.

Compared to GNN, the additional faults that DepGraph locates are related to method-to-method relationship. The findings show the importance and effectiveness of our dependency-enhanced coverage graph. While our *DepGraph* can locate more faults compared to *GNN*, there are still a few faults that our technique cannot locate (and vice versa). We manually analyze this discrepancy to help us understand whether our techniques are superior in locating certain types of faults over *GNN*. Upon analyzing faults in Lang, we find that *DepGraph* may be more effective in locating faults that are associated with the loop structures like *"for"* or *"while"* loops or control statements. For instance, in Lang, *DepGraph* found eight unique faults, while *GNN* found only two. When we checked those two faults located by *GNN*, we observed that neither fault had loop or control type statements. Moreover, out of the eight faults that were located by *DepGraph*, four were related to fixing *"for"* or *"while"* loop. Additionally, *DepGraph* exhibits peculiarity in locating faults associated with method-call relationships. For example, in JacksonCore, *DepGraph*

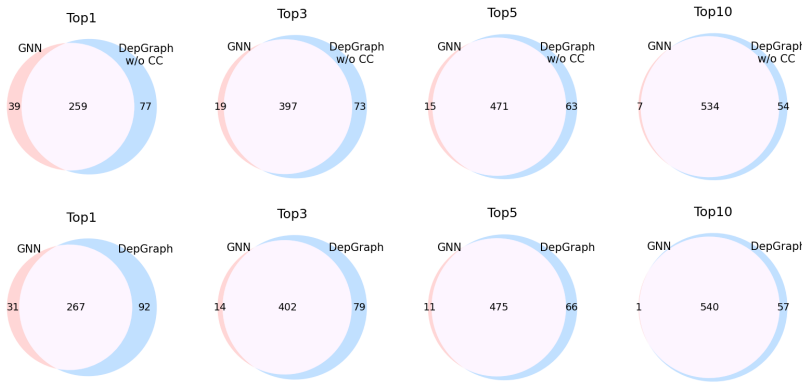

Fig. 5. Overlaps between the faults that *DepGraph* and *GNN (i.e., Grace)* locate in Top-1, 3, 5 and 10. The overlapping regions contain a number of faults that have the same ranking in both of the techniques. The non-overlapped regions contain faults that were uniquely located by each technique. We consider both *DepGraph*, and, *DepGraph w/o CC (Code Change)*.

located two additional faults compared to *GNN*. One of the two faults was related to an incorrect method-to-method call between the two classes. Hence, the analysis shows that enhancing graph representation with call graph improves the ability of our *DepGraph* to locate certain types of faults that *GNN* cannot locate.

Adopting *dependency-enhanced coverage graph* and code change information helps locate 10% to 26% additional faults compared to *GNN*. *DepGraph* detects faults related to method-to-method relationships, loop structures, and method-call relationships, which *GNN* cannot locate.

5.4 RQ4: What is the Cross-Project Fault Localization Accuracy?

Motivation. In real-world scenarios, a project might not have enough historical data to train a fault localization model. Hence, in this RQ, we explore the fault localization accuracy of *DepGraph* in a cross-project prediction scenario and compare the results against *GNN*.

Approach. We train *DepGraph* using the data from one specific project and apply the trained model to the remaining 13 projects. We repeat the process for every project. Then, for each project, we calculate the average fault localization accuracy of all the models trained on the data from other projects. For example, to evaluate the cross-project result on Math, we train 13 models separately using other projects and apply the models to Math. We then calculate the average fault localization accuracy across these 13 models. This ensures the models do not have any data leakage issues. We used GNN (i.e., *Grace*) as the baseline and trained the models by following the same procedure. We only included *GNN* in this RQ since it exhibited superior accuracy compared to other baselines as we found in RQ1.

Results. ***Both DepGraph and DepGraph$_{w/o\ Code\ Change}$ outperform the baseline GNN model in cross-project fault localization. Our models trained in cross-project settings even have higher or comparable accuracy compared to the baselines that were trained in same-project settings.*** Table 4 shows the cross-project fault localization results (i.e., cross-project setting). Across all the projects, *DepGraph (cross-project)* achieved a Top-1 of 279.463, and *DepGraph$_{w/o\ Code\ Change}$ (cross-project)* achieved a Top-1 of 241.693; both are noticeably better than the baseline *GNN (cross-project)* that achieved a Top-1 of 196.956 (42% and 23% better, respectively). Particularly, *DepGraph*

Table 4. A comparison of the cross-project fault localization accuracy. In the last row, the number in the parentheses shows the percentage improvement over *GNN* (i.e., *Grace*) [30]. The best result is marked in bold.

Project (# faults)	Techniques	Top-1	Top-3	Top-5	Top-10	MFR	MAR
Cli (39)	GNN	9.25	14.917	17.5	21.667	15.156	17.694
	$\text{DepGraph}_{\text{w/o Code Change}}$	9.846	17.462	21	26.846	8.13	9.511
	DepGraph	**13.083**	**19.583**	**22.417**	**28.417**	**6.702**	**8.011**
Closure (174)	GNN	32.456	63.986	78.492	94.432	24.611	29.011
	$\text{DepGraph}_{\text{w/o Code Change}}$	38.231	70.231	92.538	115.923	17.55	22.039
	DepGraph	**41.545**	**73.111**	**97.736**	**116.672**	**14.753**	**21.555**
Codec (18)	GNN	5.333	10.5	12.083	14.917	4.093	6.21
	$\text{DepGraph}_{\text{w/o Code Change}}$	5.923	**11.385**	**13**	15.462	3.208	**4.489**
	DepGraph	**6.667**	10.667	12.583	**16.083**	**2.99**	5.549
Collections (4)	GNN	1.25	1.5	1.833	1.833	2.958	2.958
	$\text{DepGraph}_{\text{w/o Code Change}}$	1.154	1.538	1.923	2	1.192	1.192
	DepGraph	**1.667**	**2**	**2**	**2**	**0.25**	**0.25**
Compress (47)	GNN	14.833	23.833	28	33.417	11.406	13.686
	$\text{DepGraph}_{\text{w/o Code Change}}$	17.846	27.846	32.077	37.769	6.494	7.73
	DepGraph	**21.417**	**31.833**	**36.417**	**41.333**	**4.619**	**5.696**
Csv (16)	GNN	5.583	7.75	9.167	10.75	10.578	11.083
	$\text{DepGraph}_{\text{w/o Code Change}}$	6	**8.615**	**10.769**	**12.923**	**5.947**	**6.106**
	DepGraph	**6.75**	8.417	10.5	12.5	7.203	7.451
Gson (18)	GNN	7.417	10.167	11.75	13.167	13.784	14.23
	$\text{DepGraph}_{\text{w/o Code Change}}$	**9.231**	11.615	12.692	14.308	**4.195**	**4.509**
	DepGraph	7.917	**11.667**	**13.5**	**15**	6.475	6.739
JacksonCore (26)	GNN	0.917	3.167	3.667	5.417	41.851	47.374
	$\text{DepGraph}_{\text{w/o Code Change}}$	7.154	12.462	13.462	**15.385**	**4.822**	**6.171**
	DepGraph	**8.667**	**12.667**	**13.833**	14.75	8.246	9.823
JacksonXml (6)	GNN	0.75	1.667	1.917	3	25.733	25.733
	$\text{DepGraph}_{\text{w/o Code Change}}$	1.538	2.615	3.077	4.154	4.215	4.215
	DepGraph	**2**	**3.333**	**4.333**	**4.833**	**2.15**	**2.15**
Jsoup (93)	GNN	25.667	42.833	53.417	61.667	29.87	36.572
	$\text{DepGraph}_{\text{w/o Code Change}}$	36.231	50.154	61.462	69.846	11.028	13.879
	DepGraph	**43.25**	**60.583**	**68.917**	**77.5**	**4.766**	**7.503**
Lang (64)	GNN	34.583	48.25	52.583	57.333	3.035	3.584
	$\text{DepGraph}_{\text{w/o Code Change}}$	37.462	50.077	54.538	60	1.935	2.241
	DepGraph	**42.667**	**53.583**	**57.75**	**60.75**	**1.507**	**1.795**
Math (106)	GNN	41	60.583	69.5	82.25	8.452	9.302
	$\text{DepGraph}_{\text{w/o Code Change}}$	46.923	71.077	82.615	94.538	2.922	3.297
	DepGraph	**55.25**	**82.917**	**91.917**	**99.417**	**1.748**	**2.103**
Mockito (38)	GNN	10.417	18	20.167	24.083	21.514	28.657
	$\text{DepGraph}_{\text{w/o Code Change}}$	13.385	25.231	28.846	32.538	3.788	4.852
	DepGraph	**17.5**	**27.083**	**30.917**	**33.167**	**2.618**	**3.474**
Time (26)	GNN	7.5	11.75	14.333	17	19.587	22.058
	$\text{DepGraph}_{\text{w/o Code Change}}$	10.769	14.692	16.154	18.385	10.893	11.752
	DepGraph	**11.083**	**15.333**	**17.417**	**20**	**8.948**	**10.034**
Total (675)	GNN	196.956	318.903	374.409	440.933	16.616	19.154
	$\text{DepGraph}_{\text{w/o Code Change}}$	241.693 (23%)	375 (18%)	444.153 (19%)	520.077 (18%)	6.166 (63%)	7.285 (62%)
	DepGraph	**279.463 (42%)**	**412.777 (29%)**	**480.237 (28%)**	**542.422 (23%)**	**5.213 (68%)**	**6.581 (65%)**

(cross-project) showed an improvement of 23% to 29% in Top-3, Top-5, and Top-10, and over 65% improvement in MFR and MAR, reflecting its higher fault localization accuracy. Although *DepGraph (cross-project)* achieved better fault localization accuracy compared to $DepGraph_{w/o\ Code\ Change}$ *(cross-project)*, $DepGraph_{w/o\ Code\ Change}$ *(cross-project)* is still significantly better than *GNN (cross-project)* across all evaluation metrics. The results highlight the importance of both our *Dependency-Enhanced Coverage Graph* and code change information in improving GNN-based fault localization techniques.

Furthermore, *DepGraph's* cross-project fault localization results are comparable to or even superior to that of other baselines that were trained and tested using data from the same project (i.e., same-project setting). Notably, as shown in RQ1, *Ochiai*, and *DeepFL* identified 121 and 257 faults in the Top-1 position, respectively, while *DepGraph (cross-project)* achieved an average Top-1 of 279.463, better than *Ochiai* by 131% and *DeepFL* by 8.7%. We find that *DepGraph (cross-project)* and *GNN (same-project)* have similar fault localization results across all metrics, and *DepGraph (cross-project)* even has better MFR and MAR scores. Our results show the potential of using *DepGraph* in

a cross-project setting due to its superior result even when compared to other fault localization techniques that are trained using data from the same project.

In cross-project settings, *DepGraph* achieves a 42% higher Top-1 compared to *GNN*. We also find that *DepGraph* in cross-project settings has better or comparable accuracy compared to other baselines trained in the same-project setting.

6 THREATS TO VALIDITY

Threats to internal validity may arise from our technique implementations and experimental scripts. To address these, we have reviewed our code thoroughly and implemented it on state-of-the-art frameworks like Pytorch [41]. For *Grace* and *DeepFl*, we used the original implementations provided in prior work [23, 30]. Another internal threat can be our manual analysis in the RQ3. The first two authors discussed each disagreement and analyzed the faults independently to mitigate subjectivity. *Threats to external validity* may be tied to the benchmark used. We have countered this by testing on a popular benchmark Defects4J [19], featuring numerous real-world bugs and ensuring our techniques were evaluated on its latest version (V2.0.0). *Threats to construct validity* may lie in the measurements of our study. To mitigate this, we adopted the leave-one-out cross-validation approach for a more generalizable study outcome which is also used in some prior studies [29, 30]. We also perform cross-project evaluations to mitigate data leakage issues. Our results show that even when trained in the cross-project scenario *DepGraph* still consistently outperforms the state-of-the-art GNN technique.

7 CONCLUSION

In this paper, we introduced a new graph neural network (GNN) based fault localization technique, *DepGraph*. *DepGraph* extends existing state-of-the-art GNN-based technique, *Grace* [30], by developing *Dependency-Enhanced Coverage Graph*, a unique graph representation that includes interprocedural call graph and software evolution details. Unlike previous graph representations, *dependency-enhanced coverage graph* removes edges between methods lacking call dependencies, thereby reducing graph noise and size. Afterward, we add code churn to the node attributes, providing historical code evolution information to the GNN. Our evaluation on the Defects4j benchmark revealed that the *DepGraph* outperformed existing methods. Notably, *DepGraph* outperformed *Grace* [30] across multiple metrics, highlighting the impact of our enhanced graph representation. *DepGraph*'s enhanced graph structure, particularly when combined with code change data, showcased its effectiveness in identifying complex faults. Additionally, in cross-project settings, our approach demonstrated robust adaptability and significant performance improvements, underlining its potential in diverse software environments. Furthermore, with *Dependency-Enhanced Coverage Graph*, we managed to reduce the GPU memory usage from 143GB to 80GB and speed up the training time from 9 days to just 1.5 days. Future research might explore incorporating more data layers into the graph to refine fault localization accuracy and optimize training/inference resource allocation.

8 DATA AVAILABILITY

We have made our replication package available, which contains all the datasets and code available here: https://github.com/anonymoussubmission9/anonymous-submission [4].

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46.

https://doi.org/10.1109/PRDC.2006.18

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99. https://doi.org/10.1109/ASE.2009.25

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*. 1–6. https://doi.org/10.1109/ICEngTechnol.2017.8308186

[4] AnonymousSubmission9. 2023. Replication package and data. https://github.com/anonymoussubmission9/anonymous-submission.git GitHub repository.

[5] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[7] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2021. On the effectiveness of unified debugging: an extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 907–918. https://doi.org/10.1145/3324884.3416566

[8] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 378–381. https://doi.org/10.1145/2351676.2351752

[9] An Ran Chen, Tse-Hsun (Peter) Chen, and Junjie Chen. 2023. How Useful is Code Change Information for Fault Localization in Continuous Integration?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 52, 12 pages. https://doi.org/10.1145/3551349.3556931

[10] Zhanqi Cui, Minghua Jia, Xiang Chen, Liwei Zheng, and Xiulei Liu. 2020. Improving Software Fault Localization by Combining Spectrum and Mutation. *IEEE Access* 8 (2020), 172296–172307. https://doi.org/10.1109/ACCESS.2020.3025460

[11] Rahul Dey and Fathi M. Salem. 2017. Gate-variants of Gated Recurrent Unit (GRU) neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 1597–1600. https://doi.org/10.1109/MWSCAS.2017.8053243

[12] Arpita Dutta and Sangharatna Godboley. 2021. Msfl: A model for fault localization using mutation-spectra technique. In *Lean and Agile Software Development: 5th International Conference, LASD 2021, Virtual Event, January 23, 2021, Proceedings 5*. Springer, 156–173.

[13] Georgios Gousios. 2023. java-callgraph: A simple callgraph generator tool for Java. https://github.com/gousiosg/java-callgraph/tree/master GitHub repository.

[14] Alex Graves and Alex Graves. 2012. Long short-term memory. *Supervised sequence labelling with recurrent neural networks* (2012), 37–45.

[15] C. Hait and G. Tassey. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. DIANE Publishing Company.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[18] J.A. Jones, M.J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 467–477. https://doi.org/10.1145/581396.581397

[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[20] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[21] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 579–590.

[22] Tien-Duy B. Le, Ferdian Thung, and David Lo. 2013. Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization. In *2013 IEEE International Conference on Software Maintenance*. 380–383. https://doi.org/10.1109/ICSM.2013.52

[23] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/3293882.3330574

[24] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (oct 2017), 30 pages. https://doi.org/10.1145/3133916

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

[26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[27] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1461–1472. https://doi.org/10.1109/ICSE43902.2021.00131

[28] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. TeLL: log level suggestions via modeling multi-level code block information. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (, Virtual, South Korea,) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3533767.3534379

[29] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 75–87. https://doi.org/10.1145/3395363.3397351

[30] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3468264.3468580

[31] Hehuan Ma, Yu Rong, and Junzhou Huang. 2022. Graph Neural Networks: Scalability. *Graph Neural Networks: Foundations, Frontiers, and Applications* (2022), 99–119.

[32] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162. https://doi.org/10.1109/ICST.2014.28

[33] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. 284–292.

[34] Thanh-Dat Nguyen, Thanh Le-Cong, Duc-Minh Luong, Van-Hai Duong, Xuan-Bach D. Le, David Lo, and Quyet-Thang Huynh. 2022. FFL: Fine-grained Fault Localization for Student Programs via Syntactic and Semantic Reasoning. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 151–162. https://doi.org/10.1109/ICSME55016.2022.00022

[35] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[36] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628. https://doi.org/10.1002/stvr.1509

[37] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[38] Jie Qian, Xiaolin Ju, and Xiang Chen. 2023. GNet4FL: effective fault localization via graph convolutional neural network. *Automated Software Engineering* 30, 2 (2023), 16.

[39] Jie Qian, Xiaolin Ju, Xiang Chen, Hao Shen, and Yiheng Shen. 2021. AGFL: A Graph Convolutional Neural Network-Based Method for Fault Localization. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 672–680. https://doi.org/10.1109/QRS54544.2021.00077

[40] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[41] PyTorch Team. 2023. *PyTorch*. https://pytorch.org/

[42] Chris Thunes. 2023. javalang: Pure Python Java parser and tools. https://github.com/c2nes/javalang GitHub repository.

[43] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. 2021. Call Frequency-Based Fault Localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 365–376. https://doi.org/10.1109/SANER50967.2021.00041

[44] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[45] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[46] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[47] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2012. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Transactions on Reliability* 61, 1 (2012), 149–169. https://doi.org/10.1109/TR.2011.2172031

[48] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[49] W Eric Wong and Yu Qi. 2009. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* 19, 04 (2009), 573–597.

[50] Shumei Wu, Zheng Li, Yong Liu, Xiang Chen, and Mingyu Li. 2023. GMBFL: Optimizing Mutation-Based Fault Localization via Graph Representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 245–257. https://doi.org/10.1109/ICSME58846.2023.00033

[51] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[52] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 31 (oct 2013), 40 pages. https://doi.org/10.1145/2522920.2522924

[53] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 808–819. https://doi.org/10.1145/2884781.2884834

[54] Jiaxi Xu, Fei Wang, and Jun Ai. 2020. Defect prediction with semantics and context features of codes based on graph representation learning. *IEEE Transactions on Reliability* 70, 2 (2020), 613–625.

[55] Xuezheng Xu, Changwei Zou, and Jingling Xue. 2020. Every mutation should be rewarded: Boosting fault localization with mutated predicates. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 196–207.

[56] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*. 261–272.

[57] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1089–1113.

[58] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 445–455. https://doi.org/10.1109/SANER.2019.8668002

[59] Wei Zheng, Desheng Hu, Jing Wang, et al. 2016. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering* 2016 (2016).

[60] Jian Zhou, Hongyu Zhang, and David Lo. 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 (ICSE)*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210

[61] Zhe Zhou, Cong Li, Xuechao Wei, Xiaoyang Wang, and Guangyu Sun. 2023. GNNear: Accelerating Full-Batch Training of Graph Neural Networks with near-Memory Processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 54–68. https://doi.org/10.1145/3559009.3569670

[62] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.

[63] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.

Received 2023-09-28; accepted 2024-04-16