ABSTRACT

Coverage-based fault localization has been extensively studied in the literature due to its effectiveness and lightweightness for real-world systems. However, existing techniques often utilize coverage in an oversimplified way by abstracting detailed coverage into numbers of tests or boolean vectors, thus limiting their effectiveness in practice. In this work, we present a novel coverage-based fault localization technique, Grace, which fully utilizes detailed coverage information with graph-based representation learning. Our intuition is that coverage can be regarded as connective relationships between tests and program entities, which can be inherently and integrally represented by a graph structure: with tests and program entities as nodes, while with coverage and code structures as edges. Therefore, we first propose a novel graph-based representation to reserve all detailed coverage information and fine-grained code structures into one graph. Then we leverage Gated Graph Neural Network to learn valuable features from the graph-based coverage representation and rank program entities in a listwise way. Our evaluation on the widely used benchmark Defects4J (V1.2.0) shows that Grace significantly outperforms state-of-the-art coverage-based fault localization: Grace localizes 195 bugs within Top-1 whereas the best compared technique can at most localize 166 bugs within Top-1. We further investigate the impact of each Grace component and find that they all positively contribute to Grace. In addition, our results also demonstrate that Grace has learnt essential features from coverage, which are complementary to various information used in existing learning-based fault localization. Finally, we evaluate Grace in the cross-project prediction scenario on extra 226 bugs from Defects4J (V2.0.0), and find that Grace consistently outperforms state-of-the-art coverage-based techniques.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
Fault Localization, Graph Neural Network, Representation Learning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION

Fault localization (FL) [19, 34, 39, 54, 63, 70, 74, 78] aims to diagnose buggy program entities (i.e., classes, methods, or statements) fully
Among them, coverage-based fault localization has been intensively proposed by numbers of tests). Moreover, some information (e.g., bug compressed coverage representation in SBFL (i.e., summarizing coverage) is utilized in existing SBFL, which inherently suffers from the same issues of the techniques directly adopt suspiciousness scores generated by existing SBFL and other information. Among them, coverage-based fault localization has been extensively studied in the literature due to its effectiveness and lightweightness for real-world systems [85].

Spectrum-based fault localization (SBFL) [9, 33, 68], one of the most popular coverage-based FL techniques, identifies buggy program entities by statistically analyzing coverage of failed and passed tests. In particular, existing SBFL represents coverage by the numbers of failed and passed tests covering each program entity, and regards the entities covered by more failed tests and less passed tests as more suspicious. Although widely adopted for its simplicity and efficiency, SBFL has limited effectiveness in practice [72], which results from two major drawbacks in its design. (1) SBFL utilizes coverage in an oversimplified way by abstracting it into the number of covering tests for each program entity, which may ignore detailed coverage information. (2) SBFL considers coverage as the only input information, which cannot always infer the actual causal relationships between program entities and faulty behaviours, or distinguish program entities with similar coverage.

To address the limitations in traditional spectrum-based fault localization, recently learning-based fault localization techniques have been proposed, which leverages advanced machine/deep learning techniques to (1) utilize coverage more exhaustively (i.e., learning-to-represent), or (2) integrate coverage with extra information more intelligently (i.e., learning-to-combine). In particular, learning-to-represent techniques [69, 71, 81, 82] summarize coverage by a finer-grained representation (i.e., a boolean vector for each test), based on which various learning approaches are further applied to learn causal relationships between test coverage and test outcome. However, such a representation can still be imprecise, since it treats each program entity equally and analyzes each test independently. In addition, these techniques still consider coverage as the only input, which suffers from the same issue of single information source as SBFL. Orthogonal to more exhaustive coverage utilization, learning-to-combine fault localization techniques [41, 42, 62, 85] learn to integrate coverage with extra information by using suspiciousness scores computed by existing SBFL and other information as features. For example, FLUCCS [62] learns to rank program entities based on suspiciousness scores of existing SBFL, code complexity, and code history; similarly, the latest learning-to-combine technique DeepFL [41], utilizes neural networks [49, 55] to combine suspiciousness scores of spectrum-based FL and mutation-based FL [50, 52, 78], code complexity, and text similarity. However, these techniques directly adopt suspiciousness scores generated by existing SBFL, which inherently suffer from the same issues of the compressed coverage representation in SBFL (i.e., summarizing coverage by numbers of tests). Moreover, some information (e.g., bug reports and code change history) used in these techniques cannot be always available, while other information (e.g., mutation) can be very time-consuming to collect [85], limiting their applications in practice. In summary, although achieving substantial improvement, existing learning-based fault localization techniques still fail to well address the limitations in coverage-based fault localization.

In this work, we present a novel coverage-based fault localization technique, GRACE, which leverages Graph-based representation learning to fully utilize coverage information. The intuition in GRACE is that coverage can be regarded as connective relationships between tests and program entities, which can be inherently and integrally represented by a graph structure: with tests and program entities as nodes, while with coverage and code structures as edges. Therefore, we first propose a novel graph-based representation to reserve all detailed coverage information and fine-grained code structures into one graph. Then we leverage Gated Graph Neural Network (GGNN) [43] to learn helpful features from the graph-based coverage representation, and to rank program entities in a listwise way. Different from traditional machine learning and neural networks which often preprocess graph structured data to a simpler representation before learning [17, 25], GGNN can directly analyze graph structured information with all topological relationships reserved [27, 60], and has a prominent capability in graph analysis, enabling more powerful fault localization.

We evaluate GRACE on the widely used benchmark [35] Defects4J (V1.2.0), which contains 395 real-world bugs from six open-source Java projects. Our results show that GRACE significantly outperforms state-of-the-art coverage-based fault localization techniques including Ochiai [10], CNNFL [81], FLUCCS [62], and DeepFL [41]. For example, GRACE localizes 195 bugs within Top-1 whereas the compared techniques can at most localize 166 bugs within Top-1. We further investigate the impact of each component and find that: (1) the default listwise ranking is the most effective ranking loss function; (2) the default fine-grained code structures with detailed coverage information can also positively contribute to GRACE; (3) representing tests by oversimplified numbers as prior work significantly degrades fault localization. In addition, GRACE can further boost state-of-the-art learning-to-combine fault localization, DeepFL, by integrating suspiciousness scores of GRACE as extra features for DeepFL, localizing 225 bugs within Top-1, the best learning-based fault localization results on Defects4J (V1.2.0) to our knowledge. This indicates that GRACE learns essential features from coverage, which are complementary to various information used in existing learning-based FL. Finally, we evaluate GRACE in the cross-project prediction scenario on extra 226 bugs from the latest version of Defects4J benchmark, i.e., Defects4J (V2.0.0). Our results show that GRACE consistently outperforms state-of-the-art coverage-based fault localization techniques on the new benchmark, indicating general effectiveness of our approach.

This paper makes the following contributions:

- A novel graph-based coverage representation that integrally reserves all detailed coverage information by representing program entities, tests, their coverage relationships, and fine-grained code structures into one unified graph. This coverage representation is general and could be applied to other problems using code coverage as inputs (e.g., regression test prioritization and reduction).
• A novel GGNN-based FL technique GRACE that leverages Gated Graph Neural Network (GGNN) to fully analyze the proposed graph-based coverage representation and to rank suspicious program entities in a listwise way.

• An extensive evaluation on two versions of well-established Defects4J benchmarks in both within-project and cross-project prediction scenarios. The results demonstrate the effectiveness and general applicability of the proposed approach. Our replication package is available at [8].

2 BACKGROUND AND RELATED WORK

Since our work leverages graph-based representation learning to boost coverage-based fault localization, in this section, we discuss the closely related work in traditional coverage-based fault localization (Section 2.1) and learning-based fault localization (Section 2.2).

2.1 Spectrum-Based Fault Localization

Spectrum-based fault localization (SBFL) [10, 11, 33, 56, 57, 68, 75], one of the most popular coverage-based FL techniques, calculates suspiciousness scores (probability of being faulty) of each program entity based on the number of failed/passed tests that cover it. The basic intuition in SBFL is that program entities covered by more failed tests and less passed tests are more likely to be faulty. More specifically, given a buggy program, the test suite (with at least one failed test), and coverage information, SBFL first abstracts coverage information into the number of tests covering each program entity $e$, including the number of failed tests covering $e (f_e)$ or not covering $e (n_f)$, and the number of passed tests covering $e (p_e)$ or not covering $e (n_p)$. Based on these numbers, SBFL further leverages ranking formulae, e.g., Ochiai [10], DStar [68], and Tarantula [33], to calculate suspiciousness scores for each program entity. For example, Ochiai computes the suspiciousness score of the program entity $e$ as $Ochiai(e) = \frac{(n_f + n_p)(f_e + p_e) - f_e p_e}{(n_f + n_p)(f_e + p_e) - f_e p_e}$.

Although widely adopted for simplicity and efficiency, SBFL has been shown to have limited effectiveness in practice [72]. In particular, traditional SBFL suffers from two major drawbacks. (1) **SBFL utilizes coverage in an oversimplified way**, which summarizes coverage by the number of tests. Such a compressed representation ignores detailed coverage information that may be essential for fault localization. (2) **SBFL considers coverage as the only input information**, which fails to distinguish program entities with similar coverage. In addition, coverage alone cannot always help infer the actual causal relationships between program entities and faulty behaviours.

2.2 Learning-Based Fault Localization

To address the limitations in traditional spectrum-based fault localization, learning-based fault localization [16, 41, 42, 62, 69, 71, 81, 82, 85] has also been extensively studied to leverage advanced machine/deep learning techniques to (1) utilize more detailed coverage information (i.e., learning-to-represent), or (2) integrate coverage with extra information more intelligently (i.e., learning-to-combine). In particular, learning-to-represent FL techniques learn suspiciousness scores from a finer-grained coverage representation [16, 82]. Different from existing SBFL summarizing coverage by the number of failed/passed tests, these techniques represent coverage of each test by a boolean vector that reserves its coverage relationships with each program entity. Given a test and its coverage vector $v$, the element $v_i$ shows whether the test covers the $i$th program entity. Learning approaches are then applied on the vectors to learn causal relationships between test coverage and test outcomes, based on which suspiciousness of program entities can further be inferred. Researchers have proposed to utilize various learning approaches, such as back propagation neural network [71], radial basis function network [69], multi-layer perceptrons [82], and convolutional neural network [81]. For example, CNNFL [81], the state-of-the-art learning-to-represent technique, leverages convolutional neural network [13] to facilitate coverage vector analysis.

Orthogonal to more exhaustive coverage utilization, learning-to-combine FL techniques learn to combine strengths of coverage and extra information by adopting suspiciousness scores computed by existing SBFL and other information as features. For example, FLUCS [62] adopts the suspiciousness scores of existing SBFL, code complexity, and code history as features; TraPT [42] leverages suspiciousness scores of existing SBFL and also mutation-based fault localization [50, 52, 78] as features; CombineFL [85] adopts suspiciousness scores computed by existing spectrum-based, mutation-based, slicing-based [12, 58], and information-retrieval-based fault localization [83] as features. Similarly, DeepFL [41], the state-of-the-art learning-to-combine fault localization technique, utilizes neural networks (e.g., recurrent neural network [49] and multi-layer perceptron [55]) to combine features of four dimensions, including suspiciousness scores of spectrum-based and mutation-based FL, code complexity, and text similarity.

Although achieving substantial improvement, existing learning-based FL techniques still fail to eliminate the limitations in traditional SBFL completely. For learning-to-represent techniques, representing coverage of each test as a vector can be imprecise, which treats all program entities equally and analyzes each test independently. Moreover, these techniques also suffer from the same issue of single information source as SBFL, since they consider coverage as the only input. For learning-to-combine techniques, adopting suspiciousness scores computed by existing SBFL inherently suffers from the same issues of the compressed coverage representation in SBFL (i.e., representing coverage as numbers of tests). Moreover, some information (e.g., bug reports and code change history) used in these techniques cannot be always available, while other information (e.g., mutation) can be rather time-consuming to collect [85], further limiting their applications in practice.

Different from existing coverage-based techniques, this work makes the first attempt to represent detailed coverage by graph structures and utilize Gated Graph Neural Network to directly cope with the proposed graph-based representation. In addition, we integrate coverage with lightweight information (i.e., fine-grained code structures) to boost coverage-based fault localization for the first time.

3 MOTIVATING EXAMPLE

To better illustrate the limitations in existing coverage-based fault localization, we further present a motivating example in this section. As shown in Table 1, we use a real bug Lang-47 from the widely-used benchmark Defects4J (V1.2.0) [35]. Lang-47 denotes
the 47th buggy version of Apache Commons Lang [1] in Defects4J (V1.2.0). Column “ID” and Column “Method signature” present the unique method identifier and signature, Column “Coverage” presents method-level coverage of both failed and passed tests, Column “SBFL” presents the suspiciousness scores of each method computed by Ochiai [10] (i.e., one of the most popular SBFL formula), and Column “Rank” presents the position of each method in the SBFL ranked list. This bug involves multiple buggy methods, i.e., appendFixedWidthPadRight and appendFixedWidthPadLeft (highlighted in grey) and triggers two test failures, i.e., ft1 and ft2. Figure 1 presents code snippets of the correct method m1 and the buggy method m2. For space limits, we present only methods and tests that are essential for fault localization.

```
public String getNullText() {
    return nullText;
}

public StrBuilder appendFixedWidthPadLeft(Object, int, char) {
    if (width > 0) {
        ensureCapacity(size + width);
        String str = (obj == null ? getNullText() : obj.toString());
        int strLen = str.length(); // bug
        ...
    }
```

Figure 1: Code snippets of m1 and m2

Unfortunately, as shown by the table, the traditional SBFL fails to localize neither buggy method within Top-1, since it always considers the method covered by more failed tests and less passed tests as more suspicious. For example, the correct method m1 and the buggy method m3 are both covered by three passed tests, but m1 is covered by more failed tests than m3. Therefore, SBFL is misled by its compressed coverage representation and inflexible ranking heuristics to make a wrong judgment. In fact, besides Ochiai, all the existing SBFL formula [42] share a same ranking intuition and thus all fail to rank the buggy method m3 before the correct method m1. Therefore, the learning-to-combine FL that directly adopts suspiciousness scores computed by existing SBFL formula would also suffer from the same issue as SBFL. For example, in our experiment (Section 6), DeepFL [41], the state-of-the-art learning-to-combine FL technique, fails to rank the buggy method within Top-1. As for learning-to-represent fault localization that represents coverage as vectors, the failed test ft1 and the passed test pt1 have the same coverage distribution and thus have identical coverage vectors. Therefore, it is challenging for the model to learn causal relationships between test coverage and test outcomes, which finally results in an incorrect ranked list. For example, in our experiment (Section 6), the representative learning-to-represent FL technique (i.e., CNNFL [81]) fails to identify the buggy methods neither.

Based on the example, several observations can be made at this point. (1) Coverage relationships among all program entities and all tests should be represented and analyzed in an integral way. Existing coverage-based FL compresses coverage as numbers or vectors, which actually regards program entities or tests as equivalent individual instances and ignores the diversity embedded in their topological coverage relationships. For example, some infrequent coverage relationships (e.g., tests covering few program entities or program entities covered by few tests) may be more helpful by reducing the search space of fault localization. In the example, the passed tests pt1 and pt2 cover almost all the methods (i.e., 5 out of 6), which can provide very limited hints to identify buggy methods compared to the other passed tests covering less methods (e.g., pt3 covering 3 methods). (2) Code structures can be helpful information for coverage-based fault localization. Although both the correct method m1 and the buggy method m2 are covered by the failed test ft1, i.e., all statements listed in Figure 1 are covered by ft1), coverage on statements of different types may be not equally important for fault localization. For example, the correct method m1 is simply structured with only one return statement, which can be covered by any test executing m1; whereas, the buggy method m2 has nested structures with if statements, which are more challenging to be covered for a test executing m2 and thus may provide richer features on program behaviours. In addition, code structures (e.g., abstract syntax structures) are lightweight information and always available with low collection costs compared to various other information (e.g., mutation or bug reports) used in existing learning-based FL.

### 4 APPROACH

Inspired by observations above, in this work, we present a novel coverage-based fault localization technique, Grace, which fully exploits coverage information via graph-based representation learning. More specifically, given a buggy program, its test suite, and coverage information, Grace identifies buggy program entities by two phases. First, to reserve detailed coverage information, Grace novelty represents all program entities, tests, their coverage relationships, and fine-grained code structures into one graph (Section 4.1); then, Grace leverages Gated Graph Neural Network [43] to learn key features from the proposed graph-based representation and to rank suspicious program entities in a listwise way (Section 4.2).

#### 4.1 Graph-Based Coverage Representation

In this section, we formally define the proposed graph-based coverage representation, which regards program entities and tests as nodes, and coverage relationships and code structures as edges in one graph, i.e., **unified coverage graph**. For better illustration, we first describe three key sub-components in the graph, including: (1) code representation showing how to represent code structures with code nodes and code edges, i.e., Definition 4.1; (2) test representation
showing how to represent tests with test nodes, i.e., Definition 4.2; and (3) coverage representation showing how to represent coverage relationships as coverage edges in the graph, i.e., Definition 4.3.

**Definition 4.1. Code representation.** Given a method \( m \) in the buggy program and its abstract syntax tree (AST) \(^1\), its statement-level abstract syntax tree \( G_{\text{ast}}^{m} \) is a subgraph of its original AST, containing only statement- and block-level nodes and their corresponding edges (i.e., the token-level nodes are excluded). \( G_{\text{ast}}^{m} = (V_{\text{ast}}^{m}, E_{\text{ast}}^{m}) \), where \( V_{\text{ast}}^{m} \) and \( E_{\text{ast}}^{m} \) denote code nodes and code edges. In particular, for each code node \( v_c \in V_{\text{ast}}^{m} \), \( \text{attr}(v_c) \) denotes a set of its attributes. \( v_{\text{root}}^{m} \) denotes the root code node in \( G_{\text{ast}}^{m} \).

Instead of adopting a complete AST, we use statement-level AST to represent code structures. In fact, statement- and token-level representations are equally informative in terms of the widely statement coverage information, but the latter significantly enlarges scales (i.e., the number of nodes and edges) and substantially increases unnecessary computation costs. As for node attributes, we consider AST node type and test correlation for code nodes. In particular, AST node type (e.g., if statement and return statement) effectively distills the syntactic type of each node, which can be helpful for fault localization. Test correlation represents the textual similarity between code nodes and failed tests, which has been inspired by information retrieval-based fault localization [83] that identifies buggy code based on the textual similarity between source files and bug reports. These two attributes aim to reserve syntactic and semantic features of the buggy program respectively, and we would further describe their detailed construction in Section 4.2.1.

**Definition 4.2. Test representation.** Given the test suite \( T \), test nodes \( V_T \) refer to a set of all tests in \( T \). In particular, for each test node \( v_t \in V_T \), its node attribute \( \text{attr}(v_t) \) is its test outcome, i.e., \( \text{attr}(v_t) \in \{X, \checkmark\} \).

We represent each test as an individual test node and distinguish failed and passed tests by using their outcomes as node attributes, i.e., \( \checkmark \) or \( X \).

**Definition 4.3. Coverage representation.** Given a method \( m \) in the buggy program and the test suite \( T \), statement-level coverage \( C[m, t] \) denotes a set of statements in \( m \) that are covered by the test \( t \). We represent coverage as a set of edges \( E_{\text{cov}} \), between code nodes \( V_{\text{ast}}^{m} \) and test nodes \( V_T \), i.e., \( E_{\text{cov}} = \{< v_c, v_t | c \in C[m, t], t \in T \} \), where \( v_c \) denotes the corresponding code node of the statement \( c \) in \( G_{\text{ast}}^{m} \) and \( v_t \) denotes the corresponding test node of the test \( t \) in \( V_T \).

We represent coverage relationships as coverage edges between code nodes and test nodes. In the motivating example, Figure 2 illustrates code, test, and coverage representations for the method \( m_2 \). For clear illustration, we have not included all test nodes in the figure. By now, we have presented representations within each method. We further merge representations of all methods in the buggy program into one graph, i.e., unified coverage graph, as shown in Definition 4.4. The unified coverage graph integrally represents coverage information of the whole program, including nodes of two categories (i.e., code nodes and test nodes) and edges of two categories (i.e., code edges and coverage edges). Figure 3 illustrates the final representation for the entire program in the motivating example.

**Definition 4.4. Unified coverage graph.** Given the method set \( M \) in the buggy program, the test suite \( T \), and statement-level coverage \( C \), the unified coverage graph \( G \) of the buggy program is a graph including code, test, and coverage representations of all methods in \( M \). \( G = (V, E) \), where nodes \( V = \{ \bigcup_{m \in M} V_{\text{ast}}^{m} \cup V_T \} \) and edges \( E = \{ \bigcup_{m \in M} E_{\text{ast}}^{m} \cup \bigcup_{m \in M} E_{\text{cov}}^{m} \} \).

In fact, unified coverage graph is a general representation and could be applied to not only fault localization but also other problems that mainly use code, test, and coverage as inputs, such as coverage-based regression test prioritization [20, 44, 45, 47, 48, 59] and reduction [51, 76, 77]. In addition, provided with extra code/test related information, the unified coverage graph can be further extended by representing new information as additional node attributes, nodes, or edges based on the basic unified coverage graph.

![Figure 2: Representations for the method m2](image1)

![Figure 3: Unified coverage graph for Lang-47](image2)

### 4.2 Proposed Model

#### 4.2.1 Inputs

Given a buggy program, the test suite with at least one failed test, and the statement-level coverage information, we construct the unified coverage graph \( G \) as follows. (1) We first parse the buggy program by Javalang toolkit [5] to obtain AST representations for each method, based on which we further construct

---

\(^1\)In this work, we treat AST as an unweighted and undirected graph.
its statement-level AST $G^m_{ast}$ by removing token-level nodes and relevant edges. (2) We further connect $G^m_{ast}$ of each method into one graph by including test nodes and coverage edges according to coverage information. (3) Finally, we annotate each node with its attributes in the graph. As mentioned in Section 4.1, we annotate test nodes with test outcomes, and annotate code nodes with AST node type and test correlation. In particular, for AST node type, we adopt the node type generated by Javalang during AST parsing, including 13 types in total; for test correlation, currently we consider this attribute only for each root code node by calculating the textual similarity between its belonging method name and failed test names, since prior fault localization work has demonstrated the effectiveness of textual similarities between failed tests and buggy methods [41]. Intuitively, a method whose name has a higher textual similarity with failed test names is more likely to be faulty. Therefore, following prior work [84], we compute the textual similarity between two words as Equation 1, where $w_m$ and $w_t$ denote the method name and the failed test name, and $\text{len}(w_m \cap w_t)$ and $\text{len}(w_t)$ denote the number of their common tokens and the number of tokens in the failed test name after they are tokenized by CamelCase. In particular, we adopt the maximum value when there are more than one failed tests.

$$\text{cor}(w_m, w_t) = \frac{\text{len}(w_m \cap w_t)}{\text{len}(w_t)}$$

(1)

To further represent the constructed unified coverage graph in a suitable format for the graph neural network, we use an adjacency matrix $A$ to represent the graph structure, and use an attribute sequence $S$ to represent all node attributes in the graph. In particular, given the unified coverage graph $G = (V,E)$, the element $A_{i,v_j}$ in the adjacency matrix $A \in \{0, 1\}^{|V|\times|V|}$, represents whether node $v_i$ connects with node $v_j$. To avoid gradient vanishing/exploding issues caused by cumulative degrees in the matrix, we further normalize $A$ as $\tilde{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ [65], where $D$ denotes a diagonal matrix, i.e., $D_{v_i,v_i} = \text{deg}(i)^{-\frac{1}{2}}$, and $\text{deg}(i)$ is the cumulative degrees of node $v_i$. In the attribute sequence $S$, $S_{v_i}$ denotes the attributes of node $v_i$. In particular, for the test node and non-root code node that have only one attribute, it includes one token, which can be any of {X, Y, type}; and for the root code node that has two attributes, it includes one token (i.e., AST node type) and one float value (i.e., test correlation), which can be a two-tuple (type, cor).

By far, the unified coverage graph $G$ has been represented by the systematic adjacency matrix $\tilde{A}$ and the attribute sequence $S$, which are further fed to the neural network as its inputs.

4.2.2 Gated Graph Neural Network. Traditional machine learning and neural networks often handle graph structured data with a preprocessing phase to transform the graph to a simpler representation, during which important information may be lost [17, 25]; whereas Graph Neural Network (GNN) [27, 60], which can directly analyze graph structured information with all topological dependency reserved [27, 60], has gained increasing popularity in recent years. Gated Graph Neural Network (GGNN) [43, 61] is a variant of GNN that further includes gated units to preserve long-term dependencies, such as Gated Recurrent Unit (GRU) [23] and Long Short Term Memory (LSTM) [30]. Therefore, in GRACE, we design a GGNN model to learn key features in the unified coverage graph and to rank suspicious code nodes in the graph. Figure 4 shows the architecture of our model. We then describe key components as follows.

**Embedding layer.** The word embedding layer first encodes the attribute sequence $S$ into an attribute matrix $X \in \mathbb{R}^{|V|\times d}$, where $d$ denotes the embedding size. As mentioned above, the test node and non-root code node have only one attribute token, which can be directly embedded into a $d$-dimension vector. For the root code node with two attributes, we first embed AST node type (i.e., a token) into a vector of $d - 1$ dimensions, and then concatenate it with test correlation (i.e., a float value) into a $d$-dimension vector.

**GGNN layer.** We apply the GGNN layer by five iterations. In the $t$th iteration, for each node, we update its current states by incorporating information from its adjacent nodes and from the previous iterations. In GRACE, we implement the gated mechanism in GGNN by leveraging input gates and forget gates in LSTM to control the propagation of cell states. In particular, $c^{(t)}_v$ denotes the cell state for node $v$ in the $t$th iteration, and initially $c^{(1)}_v = X_v$. $a^{(t)}_v$ propagates cell states of all its adjacent nodes in the $t - 1$th iteration as shown in Equation 2.

$$c^{(t)}_v = \tilde{A}_v [c^{(t-1)}_1; \ldots; c^{(t-1)}_{|V|}]$$

(2)

In particular, forget gates decide what information to be excluded from cell states, i.e., Equation 3; input gates decide what new information from current input (i.e., $a^{(t)}_v$) to be included into cell states, i.e., Equation 4. Based on new and forgetting information, cell states can be updated as Equation 5, where $\odot$ denotes Hadamard product.

$$f^{(t)}_v = \text{sigmoid}(W_f a^{(t)}_v + b_f)$$

(3)

$$i^{(t)}_v = \text{sigmoid}(W_i a^{(t)}_v + b_i)$$

(4)

$$c^{(t)}_v = \text{tanh}(W_g a^{(t)}_v + b_g)$$

$$c^{(t)}_v = f^{(t)}_v \odot c^{(t-1)}_v + i^{(t)}_v \odot c^{(t)}_v$$

(5)
To avoid the problem of vanishing gradients, we further leverage residual connection [29] and layer normalization [14] between each of the two sub-layers.

4.2.3 Inference. The outputs after the computation of all GGNN iterations, are further fed to a linear transformation layer followed by a softmax activation. In particular, for node \( v_i \), \( z_i \) denotes its output of the last iteration in GGNN layer, which is further linearly transformed into a real number \( y'_i \) as Equation 6, where \( W \in \mathbb{R}^{d \times 1} \) and \( b \in \mathbb{R} \). In Grace, nodes are ranked in a listwise way, and thus we leverage softmax function to normalize the outputs of all nodes as Equation 7, where \( p(v_i) \) denotes the probability of node \( v_i \) being faulty. Since Grace targets at method-level fault localization, we consider only root code nodes in the inference phase, i.e., \( n \) is the number of root code nodes.

\[
y'_i = Wz_i + b \quad (6)
\]

\[
p(v_i) = \frac{\exp{y'_i}}{\sum_{j=1}^{n} \exp{y'_j}} \quad (7)
\]

4.2.4 Ranking Loss Function. Listwise, pairwise, and pointwise are three common loss functions that have been widely used in learning-to-rank techniques [22, 38]. Given a ranked list, listwise function evaluates the entire list based on the order of all elements. It inherently agrees with the intuition of Grace that represents and analyzes all elements and their relationships in an integral way. Therefore, Grace adopts listwise ranking as its default loss function, which can be computed as Equation 8. In particular, \( g(v_i) \) denotes the ground truth label for node \( v_i \), and \( p(v_i) \) denotes its inference results.

\[
L_{\text{list}} = - \sum_{i=1}^{n} g(v_i) \log(p(v_i)) \quad (8)
\]

In principle, Grace can also leverage the other two functions (i.e., pairwise and pointwise) for loss calculation, which can be computed as Equation 9. Pairwise function compares buggy nodes \( v^* \) and correct nodes \( v^* \) in pair while pointwise function computes loss for each node \( v_i \) as a binary classification problem. Different from listwise function, sigmoid activation function is used in the last layer instead of softmax, i.e., \( p(v_i) = \text{sigmoid}(y'_i) \). We would further investigate the impacts of loss functions in the detailed experiments.

\[
\begin{align*}
L_{\text{pair}} &= \sum_{i \neq j} \sum_{v^*} \max(0, 1 - g(v_i) p(v_j) + g(v_j) p(v_i)) \\
L_{\text{point}} &= - (g(v_i)) \log(p(v_i)) + (1-g(v_i)) \log(1-p(v_i))
\end{align*}
\quad (9)
\]

5 EXPERIMENT DESIGN

5.1 Research Question

- **RQ1:** Effectiveness of Grace. How does Grace perform compared to state-of-the-art coverage-based fault localization techniques?
- **RQ2:** Impact analysis of Grace components.
  - **RQ2a:** Impact of ranking loss function. How does the ranking loss function impact the effectiveness of Grace?
  - **RQ2b:** Impact of code representation. How does the code representation impact the effectiveness of Grace?
  - **RQ2c:** Impact of test representation. How does the test representation impact the effectiveness of Grace?
- **RQ3:** Integrating with other information. Can Grace further boost state-of-the-art learning-based fault localization techniques that use various information?
- **RQ4:** Cross-project effectiveness on Defects4J (V2.0.0). How does Grace perform in the cross-project prediction scenario on the new benchmark Defects4J (V2.0.0)?

### Table 2: Benchmark information

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>#Bug</th>
<th>#Test</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lang</td>
<td>65</td>
<td>2,245</td>
<td>22K</td>
</tr>
<tr>
<td></td>
<td>Math</td>
<td>106</td>
<td>3,602</td>
<td>85K</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>27</td>
<td>4,130</td>
<td>28K</td>
</tr>
<tr>
<td></td>
<td>Chart</td>
<td>26</td>
<td>2,205</td>
<td>96K</td>
</tr>
<tr>
<td></td>
<td>Closure</td>
<td>133</td>
<td>7,927</td>
<td>90K</td>
</tr>
<tr>
<td></td>
<td>Mockito</td>
<td>38</td>
<td>1,366</td>
<td>23K</td>
</tr>
<tr>
<td></td>
<td>Defects4J (V1.2.0)</td>
<td>395</td>
<td>21,475</td>
<td>344K</td>
</tr>
<tr>
<td></td>
<td>Cli</td>
<td>39</td>
<td>361</td>
<td>4K</td>
</tr>
<tr>
<td></td>
<td>Codec</td>
<td>18</td>
<td>850</td>
<td>10K</td>
</tr>
<tr>
<td></td>
<td>Collections</td>
<td>4</td>
<td>1,286</td>
<td>65K</td>
</tr>
<tr>
<td></td>
<td>Compress</td>
<td>47</td>
<td>73</td>
<td>12K</td>
</tr>
<tr>
<td></td>
<td>Csv</td>
<td>16</td>
<td>257</td>
<td>2K</td>
</tr>
<tr>
<td></td>
<td>Gson</td>
<td>18</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>JacksonDatabind</td>
<td>112</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>JacksonCore</td>
<td>26</td>
<td>692</td>
<td>31K</td>
</tr>
<tr>
<td></td>
<td>JacksonXml</td>
<td>6</td>
<td>160</td>
<td>6K</td>
</tr>
<tr>
<td></td>
<td>Jsoup</td>
<td>48</td>
<td>530</td>
<td>14K</td>
</tr>
<tr>
<td></td>
<td>JsPath</td>
<td>22</td>
<td>401</td>
<td>21K</td>
</tr>
<tr>
<td></td>
<td>Defects4J (V2.0.0)</td>
<td>226</td>
<td>4,610</td>
<td>165K</td>
</tr>
</tbody>
</table>

5.2 Benchmark

We perform our experiments on the widely used benchmark Defects4J [35], which contains hundreds of reproducible real bugs from a wide range of projects. The benchmark currently has two versions: an original version Defects4J (V1.2.0) and a recently released version Defects4J (V2.0.0) [28] with extra bugs. To our knowledge, existing fault localization work uses only the original version Defects4J (V1.2.0) for evaluation. In our study, we evaluate our approach and state-of-the-art fault localization techniques not only on the original version (i.e., from RQ1 to RQ3) but also on the latest version (i.e., RQ4) for the first time.

Table 2 shows detailed information of the benchmark. Columns “ID” and “Name” present the short name and full name of each subject; Column “#Bugs” presents the number of bugs in each subject; Columns “Loc” and “#Test” present the number of lines and tests in the HEAD version of each subject. Note that the first 45 bugs in Jsoup and all bugs in Gson/JacksonCore (highlighted in gray) fail to be reproduced. Thus we exclude subjects Gson and JacksonCore and use the remaining 48 bugs for Jsoup. In total, our experiments are conducted on all 395 bugs from Defects4J (V1.2.0) and 226 additional bugs from Defects4J (V2.0.0).
5.3 Independent Variables

5.3.1 Compared techniques. In RQ1 and RQ4, we compare GRACE with the following state-of-the-art coverage-based fault localization techniques. (1) Spectrum-based fault localization. We compare GRACE with all 34 SBFL formulae studied in prior work [41] and present the best one (i.e., Ochiai [10]) in our results. (2) Learning-based fault localization. We also consider three representative learning-based fault localization for comparison, including the state-of-the-art learning-to-represent fault localization CNNFL [81], the representative learning-to-combine fault localization FLUCCS [62] based on machine learning, and the representative learning-to-combine fault localization DeepFL [41] based on deep learning. For CNNFL, since its source code is not available, we reimplement it strictly following the original paper [62]. For FLUCCS and DeepFL, we directly take their corresponding implementations from the DeepFL GitHub webpage [2]. Note that in this study we focus on coverage-based fault localization that includes only source code, tests, and coverage as inputs, while the original DeepFL technique includes mutation-based fault localization information which can be very time-consuming to collect (i.e., hours of online collection time per bug [85]). Therefore for a fair comparison with GRACE, we modify the original DeepFL implementation to exclude mutation-related features and keep the remaining three dimensions (i.e., spectrum-based fault localization information, code complexity, and text similarities) that can be derived from source code and coverage. We denote such a variant as DeepFL_{code} to differentiate with the original DeepFL (which additionally includes mutation-related features).

In RQ2, we consider the following variants of GRACE to analyze impacts of each component. (1) Ranking loss function. We consider GRACE with different ranking loss functions as mentioned in Section 4.2.4, by replacing the default loss function (i.e., listwise) with pairwise and pointwise loss respectively. For distinction, we denote these two variants as GRACE_{pair} and GRACE_{point}. Comparing the default GRACE with these variants can show the impact from ranking loss functions. (2) Code representation. We simplify current code representation (i.e., Definition 4.1) to investigate its contribution to GRACE. More specifically, instead of using code nodes, node attributes, and code edges to reserve fine-grained code structures, we use only one node with the number of containing statements to represent each method; in addition, coverage is adjusted to be edges between test nodes and method nodes. We denote the variant with such a coarse-grained code representation as GRACE_{code}-. (3) Test representation. We simplify current test representation (i.e., Definition 4.2) to investigate its contribution to GRACE. In particular, we remove test nodes and directly adopt the number of failed/passed tests that cover each code node as its extra node attributes. We denote the variant with such a coarse-grained test representation as GRACE_{test}-.

In RQ3, to investigate whether GRACE has learned novel features that are complementary to other information used in existing fault localization techniques, we further integrate GRACE with the state-of-the-art learning-to-combine technique, DeepFL [41]. In particular, we extend the original DeepFL with a fifth dimension of features, which are suspiciousness scores computed by GRACE. We denote such a variant of DeepFL as DeepFL_{GRACE}. Comparing DeepFL_{GRACE} with DeepFL, we can investigate the complementarity between GRACE and the other four feature dimensions used in DeepFL (i.e., suspiciousness scores of spectrum-based and mutation-based fault localization, code complexity, and textual similarity).

5.3.2 Experimental configurations. From RQ1 to RQ3, we perform within-project prediction by leave-one-out cross validation on bugs for each project. Following previous work [41], we split buggy versions in each project into two groups: one buggy version as testing data for prediction and all the remaining buggy versions in the same project as training data. Besides within-project prediction, in RQ4, we further perform cross-project prediction on the additional benchmark Defects4J (V2.0.0) by two-fold cross-validation. In particular, we use buggy versions of all six projects in Defects4J (V1.2.0) as training data, and randomly separate all buggy versions in Defects4J (V2.0.0) into two folds, which serve as testing set and validation set in turn.

5.4 Measurement

Following recent fault localization work [15, 40–42, 46, 62, 79], in this work, we perform fault localization at method level, because recent studies have shown that class-level fault localization is too coarse-grained to aid debugging while statement level might be too fine-grained to convey useful context information [36, 53]. We use the widely used measurements as follows [15, 40–42, 46].

Recall at Top-N. Top-N computes the number of buggy versions that have at least one buggy element localized within Top-N positions in the ranked list. Previous studies [53] have shown that developers inspect only a small number of buggy elements within top positions in the ranked list, e.g., 73.58% developers inspect only the Top-5 elements in the given list [36]. Therefore, following prior work [15, 41, 42, 46], we adopt Top-N (N=1,3,5).

Mean First Rank (MFR). For each buggy version, the first rank is the ranking of the first faulty element in the list. For each project, MFR calculates the mean of first ranks for all buggy versions.

Mean Average Rank (MAR). For each buggy version, the average rank is the average ranking of all faulty elements in the list. For each project, MAR calculates the mean of average ranks for all buggy versions.

Following previous work [15, 41, 42, 46], we use the worst ranking for the tied elements that have the same suspiciousness scores. For example, if a correct element and a buggy element are tied with each other and both ranked at kth position in the ranked list, we consider both of them are ranked at k + 1th.

5.5 Implementation

Data collection. We use ASM [18] and Java Agent [4] to instrument bytecode for coverage collection. For GRACE, we parse source code via Javalang toolkit [5] to construct AST. In line with prior work [41, 62], we use Jhawk [6], ASM [18], and Indri [3] to collect code complexity and textual similarity required by compared techniques DeepFL and FLUCCS.

Time costs. Table 3 presents the time costs for GRACE on the HEAD version of each project. In particular, Column “#Vertexes” and Column “#Edges” present the number of vertexes and edges in the constructed unified coverage graph; Column “Construct” presents the graph construction time; Column “Train” and Column
“Test” present the training and testing time for Grace. To restrain the scale of the graph, we consider only suspicious methods (i.e., covered by at least one failed test) and tests covering at least one suspicious method during graph construction. Based on the table, we can find that graph construction is highly efficient, i.e., only one minute for the largest project Closure. In addition, training time varies from seconds to one hour (i.e., 73 minutes for the largest project Closure), which is acceptable since training process is often performed offline. After the training model is ready, Grace then takes seconds to perform testing process. Overall, Grace is a lightweight learning-based technique in practice.

Hyperparameters. We globally use learning rate of 0.01 and embedding size of 32 for all projects. For the sake of efficiency, we maximize batch size based on the scale of graphs to make full use of GPU memory. In particular, we use batch size of 60 for all projects except Closure (i.e., batch size of 20), since its scale of graph is significant larger than other projects as shown in Table 3. Following prior work [41], we use a default training epoch (i.e., 10) when performing within-project prediction. The experimental results for configurations can be found at our GitHub website [8] due to space limit. Furthermore, all experiments are conducted with fixed random seed to avoid randomness and guarantee reproducibility.

Environment. All experiments are conducted on a Dell workstation with 300G RAM, Intel Xeon CPU E5-2680 v4 @ 2.40GHz, and eight 24G GPUs of GeForce RTX 3090, running Ubuntu 16.04.6 LTS. We build our experiments on PyTorch V1.7.1 [7].

<table>
<thead>
<tr>
<th>Subject</th>
<th>Graph representation</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Vertices</td>
<td># Edges</td>
</tr>
<tr>
<td>Chart</td>
<td>1.715</td>
<td>232,222</td>
</tr>
<tr>
<td>Time</td>
<td>3.785</td>
<td>276,174</td>
</tr>
<tr>
<td>Lang</td>
<td>54</td>
<td>474</td>
</tr>
<tr>
<td>Math</td>
<td>3.316</td>
<td>55,650</td>
</tr>
<tr>
<td>Mockito</td>
<td>1.946</td>
<td>631,560</td>
</tr>
<tr>
<td>Closure</td>
<td>6.246</td>
<td>4,161,080</td>
</tr>
</tbody>
</table>

5.6 Threats to Validity

Threats to internal validity lie in technique implementations and experimental scripts. To mitigate the threat, we manually check our code and build them on state-of-the-art frameworks, e.g., ASM [18] and PyTorch [7]. We also directly use the original implementations from prior work [41]. Threats to external validity lie in benchmarks used in our study. To reduce this threat, we perform our experiments on the widely-used benchmark with hundreds of real-world bugs. Furthermore, to our knowledge, we also make the first attempt to evaluate fault localization techniques on the latest version of the benchmark, i.e., Defects4J (V2.0.0), which contains additional over two hundreds real bugs. In the future, we plan to further evaluate our approach on extra bugs [32]. Threats to construct validity lie in measurements used in our study. To reduce this threat, we use multiple measurements which are all widely used in fault localization studies [40–42, 46]. In addition, we also perform our experiments under various settings (e.g., within/cross project prediction and two-fold/leave-one-out cross validation) to strengthen generality of the study.

6 RESULT ANALYSIS

6.1 RQ1: Effectiveness of Grace

Table 4 presents fault localization results of Grace and state-of-the-art coverage-based FL techniques on Defects4J (V1.2.0). The first two columns present corresponding subjects and techniques, and the remaining columns present results in terms of Top-1, Top-3, Top-5, MFR and MAR. From the table, we can observe that Grace substantially outperforms all the compared techniques in all studied metrics. Overall, Grace successfully localizes 195 bugs within Top-1, 29 more than DeepFLcoe, 35 more than FLUCCS, 140 more than CNNFL, and 115 more than Ochiai. In addition, MFR and MFR are also remarkably improved, i.e., 41.50% improvement in MFR and 37.78% improvement in MAR compared to the best compared technique DeepFLcoe, indicating that Grace is effective for all buggy elements. Moreover, Grace consistently outperforms other techniques on each project. For example, the improvement of Grace is prominent even on the largest project Closure, i.e., with 55.19% improvement in MFR and 55.29% improvement in MAR compared to the best compared technique DeepFLcoe on Closure. On the contrary, we notice that CNNFL performs extremely poorly on the Closure project, i.e., no bug is localized within Top-1. Such a poor performance actually results from its coverage representation that uses a boolean vector to represent the coverage of each test. The boolean vectors can be extremely sparse (i.e., most elements are zero), especially in large projects where a test can cover only a small ratio of program entities. Therefore, based on such a coverage representation, almost all the suspiciousness scores predicted by CNNFL are values close to zero. However, Grace would not suffer from such an issue in large projects, since we leverage graph neutral network on a graph structured representation, which focuses on only adjacent nodes rather than all nodes during learning process. This observation further demonstrates the advantage of our coverage representation and learning model on projects of different scales.

To further confirm the observations above, we perform Wilcoxon signed-rank test [67] with Bonferroni corrections [24] to investigate statistical significance between Grace and other state-of-the-art techniques. In particular, we compare the rankings of buggy elements generated by Grace and each compared technique in pair at the significance level of 0.05. The results suggest that the improvements in terms of MAR/MFR achieved by Grace are all statistically significant (i.e., $p$-value < 0.05).

6.2 RQ2: Impact Analysis

In this RQ, we further analyze the impact of each component in Grace. Figure 5 compares MFR and MAR metrics between variants and the default Grace (i.e., with listwise loss, fine-grained code and test representations). In particular, Figure 5(a) presents results of default Grace and variants of different ranking loss functions, i.e., pairwise and pointwise; Figure 5(b) presents results of default Grace and the variant of a coarse-grained code representation; Figure 5(c) presents results of default Grace and the variant of a coarse-grained test representation. Note that the results for the
Techniques and model architectures between Grace and DeepFL: 

- **Loss function**
  - Pairwise: The results demonstrate that the proposed code representation (i.e., representing code structures as code nodes and code edges in Definition 4.1) positively contributes to the effectiveness of Grace. In particular, Grace localizes 54 less bugs within Top-1 and downgrades MFR/MAR by 27.61%/23.94% when adopting the coarse-grained code representation. It confirms our motivation that integrating with fine-grained code structures can provide helpful hints from coverage-based fault localization.

- **Test representation**
  - Pointwise: The results demonstrate that the proposed test representation (i.e., representing tests as individual test nodes in Definition 4.1) also positively contributes to Grace. In particular, Grace localizes 25 less bugs within Top-1 and downgrades MFR/MAR by 9.18%/7.14% when removing the fine-grained test representation. This finding further confirms our motivation that abstracting tests into numbers can impair the integrity of coverage information and downgrade the effectiveness of fault localization.

- **Code representation**
  - Graph-based: The results demonstrate that the proposed code representation (i.e., representing code structures as code nodes) positively contributes to Grace. In particular, Grace localizes 40 less bugs within Top-1 and downgrades MFR/MAR by 26.33%/23.94% when adopting the coarse-grained code representation. It confirms our motivation that integrating with fine-grained code structures can provide helpful hints from coverage-based fault localization.

### Table 4: Comparison with state-of-the-art

<table>
<thead>
<tr>
<th>Subject</th>
<th>Techniques</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>MFR</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>Ochiai</td>
<td>6</td>
<td>14</td>
<td>15</td>
<td>9.00</td>
<td>9.51</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>7</td>
<td>13</td>
<td>14</td>
<td>38.76</td>
<td>39.52</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>15</td>
<td>19</td>
<td>16</td>
<td>8.08</td>
<td>8.85</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>11</td>
<td>19</td>
<td>5.52</td>
<td>5.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>16</td>
<td>20</td>
<td>22</td>
<td>3.84</td>
<td>4.49</td>
</tr>
<tr>
<td>Lang</td>
<td>Ochiai</td>
<td>24</td>
<td>44</td>
<td>50</td>
<td>4.63</td>
<td>5.01</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>22</td>
<td>46</td>
<td>53</td>
<td>4.03</td>
<td>4.25</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>40</td>
<td>53</td>
<td>55</td>
<td>3.40</td>
<td>3.63</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>43</td>
<td>54</td>
<td>56</td>
<td>3.08</td>
<td>3.28</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>46</td>
<td>54</td>
<td>55</td>
<td>2.34</td>
<td>2.70</td>
</tr>
<tr>
<td>Math</td>
<td>Ochiai</td>
<td>23</td>
<td>52</td>
<td>62</td>
<td>9.73</td>
<td>11.72</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>19</td>
<td>47</td>
<td>60</td>
<td>10.19</td>
<td>11.48</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>48</td>
<td>77</td>
<td>83</td>
<td>4.64</td>
<td>5.66</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>52</td>
<td>81</td>
<td>91</td>
<td>3.97</td>
<td>5.00</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>60</td>
<td>81</td>
<td>91</td>
<td>3.46</td>
<td>4.14</td>
</tr>
<tr>
<td>Time</td>
<td>Ochiai</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>15.96</td>
<td>18.87</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>38.41</td>
<td>38.60</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>8</td>
<td>15</td>
<td>18</td>
<td>9.00</td>
<td>11.90</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>12.68</td>
<td>13.74</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>11</td>
<td>16</td>
<td>20</td>
<td>8.80</td>
<td>9.42</td>
</tr>
<tr>
<td>Mockito</td>
<td>Ochiai</td>
<td>7</td>
<td>14</td>
<td>18</td>
<td>20.22</td>
<td>24.77</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>117.09</td>
<td>118.53</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>7</td>
<td>19</td>
<td>22</td>
<td>14.78</td>
<td>18.63</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>9</td>
<td>17</td>
<td>22</td>
<td>13.42</td>
<td>16.49</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>15</td>
<td>22</td>
<td>26</td>
<td>8.06</td>
<td>12.40</td>
</tr>
<tr>
<td>Closure</td>
<td>Ochiai</td>
<td>14</td>
<td>30</td>
<td>38</td>
<td>90.28</td>
<td>102.28</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>550.50</td>
<td>558.55</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>42</td>
<td>66</td>
<td>77</td>
<td>36.61</td>
<td>48.61</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>39</td>
<td>62</td>
<td>74</td>
<td>25.24</td>
<td>28.54</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>47</td>
<td>70</td>
<td>84</td>
<td>11.31</td>
<td>12.76</td>
</tr>
<tr>
<td>Overall</td>
<td>Ochiai</td>
<td>80</td>
<td>165</td>
<td>196</td>
<td>37.74</td>
<td>43.09</td>
</tr>
<tr>
<td></td>
<td>CNNFL</td>
<td>55</td>
<td>117</td>
<td>143</td>
<td>126.50</td>
<td>128.46</td>
</tr>
<tr>
<td></td>
<td>FLUCCS</td>
<td>160</td>
<td>249</td>
<td>275</td>
<td>16.53</td>
<td>21.53</td>
</tr>
<tr>
<td></td>
<td>DeepFLcoe</td>
<td>166</td>
<td>248</td>
<td>279</td>
<td>10.65</td>
<td>12.15</td>
</tr>
<tr>
<td></td>
<td>GRACE</td>
<td>195</td>
<td>263</td>
<td>298</td>
<td>6.23</td>
<td>7.36</td>
</tr>
</tbody>
</table>

Top-N metrics are similar and are omitted due to space limit. Based on the figures, we have the following observations.

**RQ2a: Impact of ranking loss functions.** Listwise is the most effective ranking loss function which achieves the best top and average rankings, outperforming pairwise and pointwise by 9.97% and 72.90% in MFR while by 10.64% and 71.63% in MAR. On the contrary, pointwise is the least effective one in terms of all metrics, e.g., it localizes 36 and 13 less bugs within Top-1 than listwise and pointwise respectively. This observation further confirms that the default listwise loss is the most suitable loss function for Grace. The reason may be that the proposed graph-based representation and graph neural network can reserve relationships between entities during learning process, which inherently supports the globally ranking mechanism (i.e., consider all elements during ranking) in listwise loss. In addition, it is interesting that in our study pairwise outperforms pointwise substantially, but the prior work presents the opposite conclusion that pairwise loss is less effective than pointwise when integrated in learning-based technique DeepFL [41]. Such inconsistencies may also result from the difference in data representations and model architectures between Grace and DeepFL.

**RQ2c: Impact of test representation.** The results demonstrate that the proposed test representation (i.e., representing tests as individual test nodes in Definition 4.1) also positively contributes to Grace. In particular, Grace localizes 25 less bugs within Top-1 and downgrades MFR/MAR by 9.18%/7.14% when removing the fine-grained test representation. This finding further confirms our motivation that abstracting tests into numbers can impair the integrity of coverage information and downgrade the effectiveness of fault localization.

**RQ2b: Impact of code representation.** The results demonstrate that the proposed code representation (i.e., representing code structures as code nodes and code edges in Definition 4.1) positively contributes to the effectiveness of Grace. In particular, Grace localizes 54 less bugs within Top-1 and downgrades MFR/MAR by 27.61%/23.94% when adopting the coarse-grained code representation. It confirms our motivation that integrating with fine-grained code structures can provide helpful hints from coverage-based fault localization.

6.3 RQ3: Integrating with Other Information

In this RQ, we integrate Grace with DeepFL to investigate complementarity between features learned by Grace and other information used in DeepFL. Table 5 presents fault localization results of the original DeepFL and the enhanced DeepFL (i.e., DeepFL-Grace). The results show that Grace can further boost DeepFL by localizing 225 bugs (i.e., 18 more bugs) within Top-1, which has also achieves the best fault localization results on Defects4J (V1.2.0) to our knowledge. Moreover, MAR and MFR are consistently improved by 31.18% and 26.33%, respectively. We further perform Wilcoxon signed-rank test with Bonferroni corrections at the significance level of 0.05, which confirms that the improvement is significant (p-value < 0.05). Our results indicate that Grace has indeed learned helpful information for fault localization by fully exploiting detailed coverage, and the learned features are complementary to various information used in state-of-the-art learning-to-combine technique DeepFL, including suspiciousness scores computed by spectrum-based fault localization and mutation-based fault localization, textual similarity, and code complexity. In addition, this finding also indicates that integrating Grace with other information can further enable more powerful fault localization.
6.4 RQ4: Cross-Project Prediction on Defects4J (V2.0.0)

We further evaluate Grace on the newer version of Defects4J benchmark, i.e., Defects4J (V2.0.0), to our knowledge, which has been used in fault localization studies for the first time. Table 6 presents fault localization results of Grace and state-of-the-art coverage-based fault localization techniques in the cross-project prediction scenario. From the table, we can observe that Grace still substantially outperforms all compared techniques by localizing 85 bugs within Top-1, i.e., 53 more than Ochiai, 58 more than CNNFL, 28 more than FLUCCS, and 42 more than DeepFL

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>MFR</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepFL</td>
<td>207</td>
<td>277</td>
<td>304</td>
<td>6.99</td>
<td>8.43</td>
</tr>
<tr>
<td>DeepFLGrace</td>
<td>225</td>
<td>290</td>
<td>316</td>
<td>4.81</td>
<td>6.21</td>
</tr>
</tbody>
</table>

7 CONCLUSION

In this work, we present a novel coverage-based fault localization technique, Grace, which fully utilizes coverage information with graph-based representation learning. We first propose a novel graph-based representation to reserve all detailed coverage information and fine-grained code structures into one graph: with tests and program entities as nodes, while with coverage and code structures as edges. Then we leverage Gated Graph Neural Network to learn valuable features from the graph-based coverage representation and to rank program entities in a listwise way. Our evaluation on the widely used benchmark Defects4J (V1.2.0) shows that Grace significantly outperforms state-of-the-art coverage-based fault localization. In particular, Grace localizes 195 bugs within Top-1 whereas the best comparison technique can at most localize 166 bugs within Top-1. We further investigate the impact of each component and find that they all positively contribute to Grace. In addition, our results also demonstrate that Grace has learned essential features from coverage, which are complementary to various information used in existing learning-based fault localization. Finally, we evaluate Grace in the cross-project prediction scenario on extra 226 bugs from Defects4J (V2.0.0), and find that Grace consistently outperforms state-of-the-art coverage-based techniques.

ACKNOWLEDGEMENTS

This work was partially supported by the National Key Research and Development Program of China No. 2017YFB1001803 and the National Natural Science Foundation of China under Grant Nos. 61872008. This work was also partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430.

REFERENCES
