Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning

Yiling Lou
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
yiling.lou@pku.edu.cn

Qihao Zhu
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
zhuqh@pku.edu.cn

Jinhao Dong
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
dongjinhao@stu.pku.edu.cn

Xia Li
Department of Software Engineering and Game Design and Development, Kennesaw State University
Kennesaw, US
xli37@kennesaw.edu

Zeyu Sun
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
szy_@pku.edu.cn

Dan Hao*
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
haodan@pku.edu.cn

Lu Zhang
HCST, Department of Computer Science and Technology, Peking University
Beijing, China
zhanglucs@pku.edu.cn

Lingming Zhang
Department of Computer Science, University of Illinois at Urbana-Champaign
Illinois, USA
lingming@illinois.edu

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.

*Dan Hao is the corresponding author. HCST is short for Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education, China.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Fault Localization, Graph Neural Network, Representation Learning

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468580

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 automatically and has been extensively studied to facilitate software debugging process. More specifically, fault localization techniques often leverage various static and/or dynamic program analysis information to compute suspiciousness scores (i.e., probability of being faulty) for each program entity. Program entities are then ranked in the descending order of their suspiciousness scores, based on which manual bug fixing or automated program repair [21, 26, 31, 66, 73] can further be applied. To date, researchers have proposed to leverage various information to facilitate fault localization, such as coverage [10, 33, 68], mutation [42, 50, 52, 78], predicate switching [80], program repair [15, 46], bug report [37, 64], and code history [62]. Among them, coverage-based fault localization has been intensively 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 [16, 62, 69, 81, 82] has 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 [41, 42, 62, 85] learns 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 suffers 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 **Gra**ph-based representation learning to fully utilize **c**ov**e**rage 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-the-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 (e_f) or not covering e (n_f), and the number of passed tests covering e (e_p) 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 $Ochiai(e) = e_f(e_f + e_p)^{-\frac{1}{2}}(e_f + n_f)^{-\frac{1}{2}}$.

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, 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 $\boldsymbol{v}$, the element v_i shows whether the test covers the ith 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, FLUCCS [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

Table 1: Motivating example: Lang-47

ID	Method signature	Coverage								SBFL	Rank
		ft_1	ft_2	pt_1	pt_2	pt_3	pt_4	pt_5	pt_6		
m_1	`public String getNullText()`	✓	✓	✓	✓	✓				0.63	1
m_2	`public StrBuilder appendFixedWidthPadLeft(Object, int, char)`	✓		✓			✓			0.41	2
m_3	`public StrBuilder appendFixedWidthPadRight(Object, int, char)`		✓		✓			✓	✓	0.35	3
m_4	`public StrBuilder()`	✓	✓	✓	✓	✓	✓	✓		0.12	5
m_5	`public StrBuilder(int)`	✓	✓	✓	✓	✓	✓	✓	✓	0.12	5
m_6	`public StrBuilder ensureCapacity(int)`	✓	✓	✓	✓		✓	✓	✓	0.10	6

the 47th buggy version of Apache Commons Lang [1] in Defects4J (V1.2.0). Column "ID" and Column "Method signiture" 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., ft_1 and ft_2. 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 m_1 and m_2

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 test and less passed tests as more suspicious. For example, the correct method m_1 and the buggy method m_3 are both covered by three passed tests, but m_1 is covered by more failed tests than m_3. 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 m_3 before the correct method m_1. 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 ft_1 and the passed test pt_1 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 pt_1 and pt_2 cover almost all the methods (i.e., 5 out of 6), which can provide very limited hints to identify buggy methods compared the other passed tests covering less methods (e.g., pt_3 covering 3 methods). (2) *Code structures can be helpful information for coverage-based fault localization.* Although both the correct method m_1 and the buggy method m_2 are covered by the failed test ft_1, (i.e., all statements listed in Figure 1 are covered by ft_1), coverage on statements of different types may be not equally important for fault localization. For example, the correct method m_1 is simply structured with only one *return* statement, which can be covered by any test executing m_1; whereas, the buggy method m_2 has nested structures with *if* statements, which are more challenging to be covered for a test executing m_2 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 novelly 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 $\mathcal{G}^m_{ast}$ 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). $\mathcal{G}^m_{ast} = (\mathcal{V}^m_{ast}, \mathcal{E}^m_{ast})$, where $\mathcal{V}^m_{ast}$ and $\mathcal{E}^m_{ast}$ denote code nodes and code edges. In particular, for each code node $v_c \in \mathcal{V}^m_{ast}$, $attr(v_c)$ denotes a set of its attributes. v^m_{root} denotes the root code node in $\mathcal{G}^m_{ast}$.*

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 $\mathcal{T}$, test nodes $\mathcal{V}_{\mathcal{T}}$ refer to a set of all tests in $\mathcal{T}$. In particular, for each test node $v_t \in \mathcal{V}_{\mathcal{T}}$, its node attribute $attr(v_t)$ is its test outcome, i.e., $attr(v_t) \in \{$✗,✓$\}$.*

We represent each test as an individual test node and distinguish failed and passed tests by using their outcomes as node attributes, i.e., ✗ or ✓.

Definition 4.3. ***Coverage representation.*** *Given a method m in the buggy program and the test suite $\mathcal{T}$, statement-level coverage $\mathbb{C}[m,t]$ denotes a set of statements in m that are covered by the test t. We represent coverage as a set of edges $\mathcal{E}^m_{cov}$ between code nodes $\mathcal{V}^m_{ast}$ and test nodes $\mathcal{V}_{\mathcal{T}}$, i.e., $\mathcal{E}^m_{cov} = \{< v_c, v_t > | c \in \mathbb{C}[m,t], t \in \mathcal{T}\}$, where v_c denotes the corresponding code node of the statement c in $\mathcal{G}^m_{ast}$ and v_t denotes the corresponding test node of the test t in $\mathcal{V}_{\mathcal{T}}$.*

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.

[1]In this work, we treat AST as an unweighted and undirected graph.

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

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.

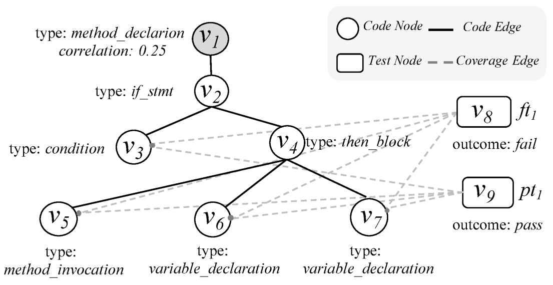


Figure 2: Representations for the method m_2

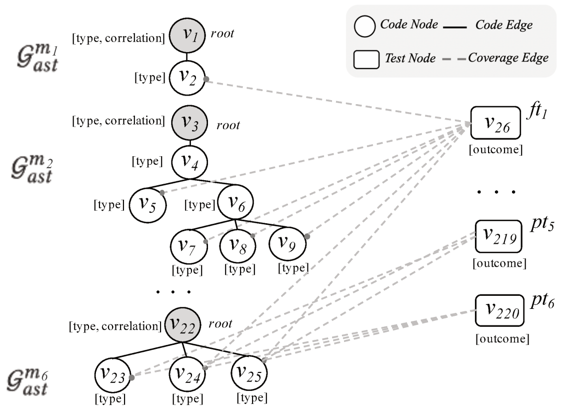


Figure 3: Unified coverage graph for Lang-47

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 $\mathcal{G}$ 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 its statement-level AST $\mathcal{G}^m_{\text{ast}}$ by removing token-level nodes and relevant edges. (2) We further connect $\mathcal{G}^m_{\text{ast}}$ 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.

$$\text{cor}(w_m, w_t) = \text{len}(w_m \cap w_t)/\text{len}(w_t) \tag{1}$$

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 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the element A_{v_i,v_j} in the adjacency matrix $A \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, 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 $\hat{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ [65], where D denotes a diagonal matrix, i.e., $D_{v_i,v_i} = \text{de}(i)^{-\frac{1}{2}}$, and $\text{de}(v_i)$ 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 {✗,✓, 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 $\mathcal{G}$ has been represented by the systematic adjacency matrix $\hat{A}$ 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

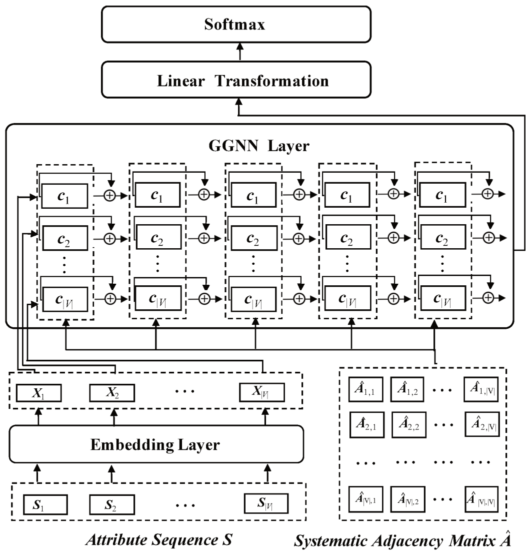

Softmax
Linear Transformation
GGNN Layer
c_1
c_2
$c_{|V|}$
X_1 X_2 ··· $X_{|V|}$
Embedding Layer
S_1 S_2 ··· $S_{|V|}$
$\hat{A}_{1,1}$ $\hat{A}_{1,2}$ ··· $\hat{A}_{1,|V|}$
$\hat{A}_{2,1}$ $\hat{A}_{2,2}$ ··· $\hat{A}_{2,|V|}$
$\hat{A}_{|V|,1}$ $\hat{A}_{|V|,2}$ ··· $\hat{A}_{|V|,|V|}$
Attribute Sequence S
Systematic Adjacency Matrix Â

Figure 4: Architecture of Grace

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 $X \in \mathbb{R}^{|\mathcal{V}|\times d}$, 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 tth 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, $\boldsymbol{c}_v^{(t)}$ denotes the cell state for node v in the tth iteration, and initially $\boldsymbol{c}_v^{(1)} = X_{v:}$. $\boldsymbol{a}_v^{(t)}$ propagates cell states of all its adjacent nodes in the $t-1$th iteration as shown in Equation 2.

$$\boldsymbol{a}_v^{(t)} = \hat{A}_{v:}[\boldsymbol{c}_1^{(t-1)\top}; \ldots; \boldsymbol{c}_{|\mathcal{V}|}^{(t-1)\top}] \tag{2}$$

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., $\boldsymbol{a}_v^{(t)}$) 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.

$$\boldsymbol{f}_v^{(t)} = \text{sigmoid}(W_f \boldsymbol{a}_v^{(t)} + \boldsymbol{b}_f) \tag{3}$$

$$\begin{aligned} \boldsymbol{i}_v^{(t)} &= \text{sigmoid}(W_i \boldsymbol{a}_v^{(t)} + \boldsymbol{b}_i) \\ \widetilde{\boldsymbol{c}_v^{(t)}} &= \tanh(W_g \boldsymbol{a}_v^{(t)} + \boldsymbol{b}_g) \end{aligned} \tag{4}$$

$$\boldsymbol{c}_v^{(t)} = \boldsymbol{f}_v^{(t)} \odot \boldsymbol{c}_v^{(t-1)} + \boldsymbol{i}_v^{(t)} \odot \widetilde{\boldsymbol{c}_v^{(t)}} \tag{5}$$

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 $\boldsymbol{b} \in \mathbb{R}$. 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.

$$y_i' = Wz_i + b \tag{6}$$

$$p(v_i) = \frac{\exp\{y_i'\}}{\sum_{j=1}^{n}\exp\{y_j'\}} \tag{7}$$

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.

$$\mathcal{L}_{list} = -\sum_{i=1}^{n} g(v_i)\log(p(v_i)) \tag{8}$$

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.

$$\begin{aligned}\mathcal{L}_{pair} &= \sum_{i\in v^-}\sum_{j\in v^+}\max\{\alpha - (p(v_i) - p(v_j)), 0\} \\ \mathcal{L}_{point} &= -(g(v_i)\log(p(v_i)) + (1 - g(v_i))\log(1 - p(v_i)))\end{aligned} \tag{9}$$

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

ID	Name	#Bug	#Test	LoC
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
Chart	JFreeChart	26	2,205	96K
Closure	Google Closure compiler	133	7,927	90K
Mockito	Mockito framework	38	1,366	23K
Defects4J (V1.2.0)		**395**	**21,475**	**344K**
Cli	Commons-cli	39	361	4K
Codec	Commons-codec	18	850	10K
Collections	Commons-collections	4	1,286	65K
Compress	Commons-compress	47	73	12K
Csv	Commons-csv	16	257	2K
Gson	Gson	18	NA	NA
JacksonDatabind	Jackson-databind	112	NA	NA
JacksonCore	Jackson-core	26	692	31K
JacksonXml	Jackson-dataformat-xml	6	160	6K
Jsoup	Jsoup	48	530	14K
JxPath	Commons-jxpath	22	401	21K
Defects4J (V2.0.0)		**226**	**4,610**	**165K**

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$_{cov}$ 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 denoted 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 denoted 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 $k + 1$th.

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 3: Efficiency of Grace

Subject	Graph representation			Model	
	# Vertexes	# Edges	Construct (s)	Train (s)	Test (s)
Chart	1,715	232,222	13.13	10.64	6.48
Time	3,785	276,174	8.65	28.76	21.82
Lang	54	474	6.51	2.22	0.34
Math	3,316	55,650	11.51	140.91	11.46
Mockito	1,946	631,560	8.03	64.44	48.50
Closure	6,246	4,161,080	62.81	4,384.53	79.88

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 DeepFL$_{cov}$, 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 DeepFL$_{cov}$, 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 DeepFL$_{cov}$ 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

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

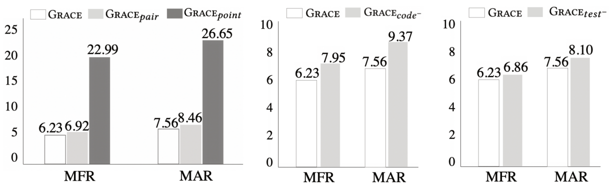
Subject	Techniques	Top-1	Top-3	Top-5	MFR	MAR
Chart	Ochiai	6	14	15	9.00	9.51
	CNNFL	7	13	14	38.76	39.32
	FLUCCS	15	19	16	8.08	8.85
	DeepFL$_{cov}$	11	19	20	5.52	5.83
	Grace	16	20	22	3.84	4.49
Lang	Ochiai	24	44	50	4.63	5.01
	CNNFL	22	46	53	4.03	4.25
	FLUCCS	40	53	55	3.40	3.63
	DeepFL$_{cov}$	43	54	56	3.08	3.28
	Grace	46	54	55	2.34	2.70
Math	Ochiai	23	52	62	9.73	11.72
	CNNFL	19	47	60	10.19	11.48
	FLUCCS	48	77	83	4.64	5.66
	DeepFL$_{cov}$	52	81	91	3.97	5.00
	Grace	60	81	91	3.46	4.14
Time	Ochiai	6	11	13	15.96	18.87
	CNNFL	4	7	8	38.41	38.60
	FLUCCS	8	15	18	9.00	11.90
	DeepFL$_{cov}$	12	15	16	12.68	13.74
	Grace	11	16	20	8.80	9.42
Mockito	Ochiai	7	14	18	20.22	24.77
	CNNFL	3	4	7	117.09	118.53
	FLUCCS	7	19	22	14.78	18.63
	DeepFL$_{cov}$	9	17	22	13.42	16.49
	Grace	15	22	26	8.06	12.40
Closure	Ochiai	14	30	38	90.28	102.28
	CNNFL	0	0	1	550.50	558.55
	FLUCCS	42	66	77	36.61	48.61
	DeepFL$_{cov}$	39	62	74	25.24	28.54
	Grace	47	70	84	11.31	12.76
Overall	Ochiai	80	165	196	37.74	43.09
	CNNFL	55	117	143	126.50	128.46
	FLUCCS	160	249	275	16.53	21.53
	DeepFL$_{cov}$	166	248	279	10.65	12.15
	Grace	195	263	298	6.23	7.56

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: different from Grace which integrally considers all methods as one training data item, DeepFL regards each method as an individual training data item, and may ignore the relationships between methods. Thus, *pairwise* cannot outperform *pointwise* for DeepFL.

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.

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.



(a) Loss function (b) Code represent. (c) Test represent.

Figure 5: Impact of Grace components

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.

Table 5: Integrating Grace with DeepFL

Techniques	Top-1	Top-3	Top-5	MFR	MAR
DeepFL	207	277	304	6.99	8.43
DeepFL$_{Grace}$	225	290	316	4.81	6.21

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$_{cov}$. Moreover, MAR and MFR are consistently improved at least by 50.57% and 38.38% compared to all the other coverage-based techniques. In addition, we can observe that compared to within-project prediction (i.e., RQ1) on Defects4J (V1.2.0), all techniques perform worse on Defects4J (V2.0.0) in the cross-project prediction scenario. For example, DeepFL$_{cov}$ can localize 42.03% bugs within Top-1 on Defects4J (V1.2.0) while only 19.03% bugs within Top-1 on Defects4J (V2.0.0); as for Grace, it can localize 49.36% bugs within Top-1 on Defects4J (V1.2.0) while 37.64% bugs within Top-1 on Defects4J (V2.0.0). The observation is as expected, since in the within-project prediction scenario, testing data and training data are from the same project, which tend to share similar features; whereas the cross-project prediction can be more challenging since characteristics between projects can be very different. Even though, we can observe that compared to other techniques, Grace exhibits the smallest effectiveness drop between within-project and cross-project prediction. In summary, our results demonstrate that even when trained in the cross-project prediction scenario, Grace still consistently outperforms state-of-the-art coverage-based techniques on hundreds of extra bugs.

Table 6: Cross-project effectiveness on Defects4J (V2.0.0)

Subject	Techniques	Top-1	Top-3	Top-5	MFR	MAR
Overall	Ochiai	32	74	93	14.26	20.19
	CNNFL	27	60	77	21.76	27.12
	FLUCCS	57	97	119	14.85	20.95
	DeepFL$_{cov}$	43	89	112	14.03	21.06
	Grace	85	119	140	6.92	12.91

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

[1] 2020. Apache Commons Lang. http://commons.apache.org/proper/commons-lang/.
[2] 2020. DeepFL Website. https://github.com/DeepFL/DeepFaultLocalization.git.
[3] 2020. Indri. https://www.lemurproject.org/indri.php.
[4] 2020. JavaAgent. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html.
[5] 2020. Javalang. https://github.com/c2nes/javalang/.
[6] 2020. Jhawk. http://www.virtualmachinery.com/jhawkprod.html.
[7] 2020. PyTorch. https://pytorch.org.
[8] 2021. Replication package. https://github.com/yilinglou/Grace.
[9] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035
[10] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 39–46. https://doi.org/10.1109/PRDC.2006.18
[11] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 88–99. https://doi.org/10.1109/ASE.2009.25
[12] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995*. IEEE Computer Society, 143–151. https://doi.org/10.1109/ISSRE.1995.497652
[13] 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)*. Ieee, 1–6.
[14] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. arXiv:1607.06450 [stat.ML]
[15] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 907–918. https://doi.org/10.1145/3324884.3416566
[16] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. 2007. Using Machine Learning to Support Debugging with Tarantula. In *ISSRE 2007, The 18th IEEE International Symposium on Software Reliability, Trollhättan, Sweden, 5-9 November 2007*. IEEE Computer Society, 137–146. https://doi.org/10.1109/ISSRE.2007.31
[17] Sergey Brin and Lawrence Page. 2012. The anatomy of a large-scale hypertextual web search engine. *Comput. Networks* 56, 18 (2012), 3825–3833. https://doi.org/10.1016/j.comnet.2012.10.007
[18] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[19] Bruno Castro, Alexandre Perez, and Rui Abreu. 2019. Pangolin: An SFL-Based Toolset for Feature Localization. In *34th IEEE/ACM International Conference on*

Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, 1130–1133. https://doi.org/10.1109/ASE.2019.00119

[20] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 656–667. https://doi.org/10.1145/3236024.3236053

[21] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1123–1134.

[22] Wei Chen, Tie-Yan Liu, Yanyan Lan, Zhiming Ma, and Hang Li. 2009. Ranking Measures and Loss Functions in Learning to Rank. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta (Eds.). Curran Associates, Inc., 315–323.

[23] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[24] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64.

[25] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. 1998. A general framework for adaptive processing of data structures. *IEEE Trans. Neural Networks* 9, 5 (1998), 768–786. https://doi.org/10.1109/72.712151

[26] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 19–30. https://doi.org/10.1145/3293882.3330559

[27] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 729–734.

[28] Greg4cr. 2021. Defects4J – version 2.0. https://github.com/rjust/defects4j.

[29] 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 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[30] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[31] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 255–266. https://doi.org/10.1109/ASE.2019.00033

[32] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[33] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 273–282. https://doi.org/10.1145/1101908.1101949

[34] Frolin S. Ocariza Jr., Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. 2016. Automatic fault localization for client-side JavaScript. *Softw. Test. Verification Reliab.* 26, 1 (2016), 69–88. https://doi.org/10.1002/stvr.1576

[35] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[36] 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, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 165–176. https://doi.org/10.1145/2931037.2931051

[37] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 314–325. https://doi.org/10.1145/3338906.3338935

[38] Tuan Manh Lai, Trung Bui, and Sheng Li. 2018. A Review on Deep Learning Techniques Applied to Answer Selection. In *Proceedings of the 27th International Conference on Computational Linguistics, COLING 2018, Santa Fe, New Mexico, USA, August 20-26, 2018*, Emily M. Bender, Leon Derczynski, and Pierre Isabelle (Eds.). Association for Computational Linguistics, 2132–2144.

[39] David Landsberg, Hana Chockler, and Daniel Kroening. 2016. Probabilistic Fault Localisation. In *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10028)*, Roderick Bloem and Eli Arbel (Eds.). 65–81. https://doi.org/10.1007/978-3-319-49052-6_5

[40] 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, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 177–188. https://doi.org/10.1145/2931037.2931049

[41] 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180. https://doi.org/10.1145/3293882.3330574

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

[43] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[44] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007), 225–237.

[45] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. Chapter One - A Survey on Regression Test-Case Prioritization. *Adv. Comput.* 113 (2019), 1–46. https://doi.org/10.1016/bs.adcom.2018.10.001

[46] 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 *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 75–87. https://doi.org/10.1145/3395363.3397351

[47] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 46–57. https://doi.org/10.1109/ISSRE.2015.7381798

[48] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 535–546. https://doi.org/10.1145/2884781.2884874

[49] Tomás Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura (Eds.). ISCA, 1045–1048.

[50] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 153–162. https://doi.org/10.1109/ICST.2014.28

[51] S Nachiyappan, A Vimaladevi, and CB SelvaLakshmi. 2010. An evolutionary algorithm for regression test suite reduction. In *2010 International Conference on Communication and Computational Intelligence (INCOCCI)*. IEEE, 503–508.

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

[53] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 199–209. https://doi.org/10.1145/2001420.2001445

[54] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 609–620. https://doi.org/10.1109/ICSE.2017.62

[55] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. 2009. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems* 8, 7 (2009), 579–588.

[56] Moeketsi Raselimo and Bernd Fischer. 2019. Spectrum-based fault localization for context-free grammars. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 15–28. https://doi.org/10.1145/3357766.3359538
[57] Sofia Reis, Rui Abreu, and Marcelo d'Amorim. 2019. Demystifying the Combination of Dynamic Slicing and Spectrum-based Fault Localization. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 4760–4766. https://doi.org/10.24963/ijcai.2019/661
[58] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*. IEEE Computer Society, 30–39. https://doi.org/10.1109/ASE.2003.1240292
[59] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.
[60] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.
[61] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. 2018. Learning a SAT solver from single-bit supervision. *arXiv preprint arXiv:1802.03685* (2018).
[62] 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, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 273–283. https://doi.org/10.1145/3092703.3092717
[63] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 1–11. https://doi.org/10.1145/2771783.2771797
[64] Shaohua Wang, Foutse Khomh, and Ying Zou. 2013. Improving bug localization using correlations in crash reports. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE Computer Society, 247–256. https://doi.org/10.1109/MSR.2013.6624036
[65] Eric W Weisstein. 1999. Laplacian matrix. *https://mathworld. wolfram. com/* (1999).
[66] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1–11. https://doi.org/10.1145/3180155.3180233
[67] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.
[68] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319
[69] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani M. Thuraisingham. 2012. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Trans. Reliab.* 61, 1 (2012), 149–169. https://doi.org/10.1109/TR.2011.2172031
[70] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368
[71] W. Eric Wong and Yu Qi. 2009. Bp Neural Network-Based Effective Fault Localization. *Int. J. Softw. Eng. Knowl. Eng.* 19, 4 (2009), 573–597. https://doi.org/10.1142/S021819400900426X
[72] 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 808–819. https://doi.org/10.1145/2884781.2884834
[73] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648
[74] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Inf. Softw. Technol.* 124 (2020), 106312. https://doi.org/10.1016/j.infsof.2020.106312
[75] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICSM.2011.6080769
[76] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, Mauro Pezzè and Mark Harman (Eds.). ACM, 235–245. https://doi.org/10.1145/2483760.2483782
[77] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, Tadashi Dohi and Bojan Cukic (Eds.). IEEE Computer Society, 170–179. https://doi.org/10.1109/ISSRE.2011.26
[78] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.
[79] 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, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 261–272. https://doi.org/10.1145/3092703.3092731
[80] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 272–281. https://doi.org/10.1145/1134285.1134324
[81] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 445–455. https://doi.org/10.1109/SANER.2019.8668002
[82] Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering* 2016 (2016).
[83] 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 *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 14–24. https://doi.org/10.1109/ICSE.2012.6227210
[84] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An Overlapping-Aware Code Retriever. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 883–894. https://doi.org/10.1145/3324884.3416530
[85] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Trans. Software Eng.* 47, 2 (2021), 332–347. https://doi.org/10.1109/TSE.2019.2892102