FDG: A Precise Measurement of Fault Diagnosability Gain of Test Cases

Gabin An
School of Computing, KAIST
Daejeon, Republic of Korea
agb94@kaist.ac.kr

Shin Yoo
School of Computing, KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

1 INTRODUCTION

Fault Localisation (FL) techniques aim to reduce the cost of software debugging by automatically identifying the root cause of the observed test failures [54]. While some FL techniques use only static data such as lexical proximity between bug reports and source code [48], the majority of FL techniques are based on dynamic analysis: Spectrum Based Fault Localisation (SBFL) relies on both the test coverage and the test results [25, 38, 53], whereas Mutation Based Fault Localisation exploits the relationship between mutants and test cases [22, 37, 40, 41]. In both cases, the quality of the test suite used with an FL technique can significantly affect its performance. It is widely known that FL techniques thrive only when accompanied by a rich and diverse test suite [44].

In reality, however, it is not always the case that the given test suite is sufficiently diverse to ensure successful fault localisation. In fact, it is not unusual to start the debugging process only with a single failing test input, calling for the need of test augmentation. Many techniques have been developed to guide the test augmentation, i.e., to add the test case that can maximise the diagnosability of the augmented test suite [3, 11]. Equally relevant are the techniques that prioritise the given test cases for better and earlier fault localisation [17, 58], as well as techniques that aim to measure the diagnostic capability of the given test suite. Most of these techniques are built around a metric that measures the diagnosability of either a single test, or a set of tests. However, our analysis of existing test diagnosability metrics reveals room for improvement. Specifically, while some of the techniques are result-aware, i.e., incorporate the result of test executions (pass/fail) into the diagnosability computation, none of them directly uses the suspiciousness scores of individual program elements during localisation, even though they can provide critical information by pinpointing where the prioritisation should focus on next.

Based on our analysis of existing techniques, we propose FDG, a metric that can precisely measure the Fault Diagnosability Gain that a test case can bring to a test suite. FDG is designed to be used during fault localisation, and uses the current suspiciousness scores to precisely capture which part of the target program directly depends on the quality of the used test suites. Consequently, it is extremely useful to be able to precisely measure how much diagnostic power each test case can introduce when added to a test suite used for FL. Such a measure can help us not only to prioritise and select test cases to be used for FL, but also to effectively augment test suites that are too weak to be used with FL techniques. We propose FDG, a new measure of Fault Diagnosability Gain for individual test cases. The design of FDG is based on our analysis of existing metrics that are designed to prioritise test cases for better FL. Unlike other metrics, FDG exploits the ongoing FL results to emphasise the parts of the program for which more information is needed. Our evaluation of FDG with DEFECTS4J shows that it can successfully help the augmentation of test suites for better FL. When given only a few failing test cases (2.3 test cases on average), FDG can effectively augment the given test suite by prioritising the test cases generated automatically by EvoSuite: the augmentation can improve the acc@1 and acc@10 of the FL results by 11.6x and 2.2x on average, after requiring only ten human judgements on the correctness of the assertions EvoSuite generates.

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:
In a more realistic setting, we also evaluate FDG using automatically generated test cases to augment the initial set of failing test inputs. We posit that the true cost of test suite augmentation is not the cost of test data generation, as it can be automated. Rather, the major cost of test augmentation is the human effort required to produce the test oracles for the generated test data. Our evaluation of FDG assumes an iterative FL scenario, in which an insufficient test suite is augmented, on the fly, with test data automatically generated by EvoSuite. Here, FDG is used to choose the next test case to be presented to the human engineer for the oracle labelling. We show that after only ten interactions with the human engineer, FDG can boost acc@1 and acc@10 by 11.6 (from 5 to 58) and 2.2 (from 63 to 147) times, respectively, compared to the initial test suites that contain only a few failing test cases. In addition, we also show that FDG is fairly resilient to labelling errors with an error rate of up to 30%.

The main contributions of this paper are as follows:

- We analyse, and empirically compare, existing metrics that measure the diagnosability of test cases for fault localisation. Our evaluation compares how quickly existing metrics can improve the accuracy of SBFL techniques by prioritising the cost of test data generation, as it can be automated. Rather, the cost required to augment an initial set of failing inputs for fault localisation.
- We propose a novel diagnosability metric for fault localisation, FDG, based on the analysis of existing metrics. FDG distinguishes itself from existing metrics in that it actively considers threats to validity. Finally, Section 8 concludes.

2 PRELIMINARIES

This section introduces the basic notations, the background of SBFL techniques, and the definition of ambiguity groups.

2.1 Basic Notation

Given a program $P$, let us define the following:

- Let $E = \{e_1, \ldots, e_n\}$ be the set of program elements that consist $P$, such as statements or methods, and $T = \{t_1, \ldots, t_m\}$ be the test suite for $P$.
- Let $C_T$ be the $m \times n$ coverage matrix of $T$:

$$C_T = \begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix} \in \mathbb{R}^{m \times n}$$

where $C_T[i, j] = a_{ij} = 1$ if $t_i$ covers $e_j$ and 0 otherwise.
- Given an arbitrary test case $t$, let $c_t \in \mathbb{B}^{1 \times n}$ be the coverage vector of $t$, where $c_t[j] = 1$ if $t$ executes $e_j$ and 0 otherwise.
- Let $R : T \rightarrow \mathbb{B}$ be the function that maps a test in $T$ to its result: $R(t) = 0$ if $t$ reveals a fault in $P$, and 1 otherwise. Subsequently, the set of failing tests, $T_f$, can be defined as $\{t \in T | R(t) = 0\}$. $P$ is faulty when $T_f$ is not empty.

2.2 Spectrum-Based Fault Localisation (SBFL)

SBFL is a statistical approach for the automated localisation of software faults [54]. It utilises program spectrum, a summary of the runtime information collected from the program executions, to find the faulty program elements. A representative program spectrum of a program element $e_j \in E$ consists of four values: $(e_p, n_p, e_f, n_f) = (N_{t_1}, N_{t_0}, N_{t_1}, N_{t_0})$ where

$$N_{ab} = |\{t_i \in T | C_T[i, j] = a \wedge R(t_i) = b\}|$$

SBFL statistically estimates the suspiciousness of each program element based on the following rationale: “The more the execution of an element is correlated with failing tests (high $e_f$, low $n_f$) and less with passing ones (high $n_p$, low $e_p$), the more suspicious the program element is, and vice versa”. A risk evaluation formula $S : I^4 \rightarrow \mathbb{R}$ is a formula that converts the four program spectrum values to a suspiciousness score. Many risk evaluation formulas [1, 25, 38, 53] have been designed to implement this core idea. For example, Ochiai [1], one of the most widely studied risk evaluation formulas, computes the suspiciousness as follows:

$$Ochiai(e_p, n_p, e_f, n_f) = e_f / \sqrt{(e_f + n_f) \times (e_f + e_p)} \tag{1}$$

Fig. 1 illustrates a concrete example of calculating Ochiai scores from the coverage matrix and test results. SBFL is one of the most widely studied FL techniques [54] as it is applicable as long as test coverage is available. Due to the applicability, SBFL scores are often used as features for more complicated learn-to-rank FL techniques [31, 33, 51], motivating us to improve its effectiveness.

2.3 Ambiguity Groups

An ambiguity group [16, 52] is a set of program elements that are only executed by the same set of test cases. Given a program $P$ and its elements $E$, we define $AG(T)$ as a set of such ambiguity groups under the test suite $T$. The size of $AG(T)$ is equal to the number of unique columns in $C_T$. More formally, $AG(T)$ is a partition [18] of the set $E$ that satisfies the following properties:

(1) $\forall e \in E, \exists g \in AG(T), e \in g$
we need an additional test case that can break the ambiguity group, even though only original test suite. When Between them, to further distinguish which one is more suspicious between them, original test cases ambiguity groups, and their size is smaller. performance of an SBFL technique increases when there are more all the elements in an ambiguity group are tied in the final ranking example, in Fig. 1: A simple motivating example with one faulty element \(e_3\). The dots (*) show the coverage relation, and the ranks are computed using the max tie-breaker.

\[
\begin{align*}
 (2) \forall g \in AG(T). \forall e_i, e_j \in g. C_T[:i] = C_T[::j] \\
 (3) \forall g_1, g_2 \in AG(T). g_1 \neq g_2 \land e_i \in g_1 \land e_j \in g_2 \\
 \implies C_T[:i] \neq C_T[::j]
\end{align*}
\]

Elements in the same ambiguity groups are assigned the same suspiciousness score, since their program spectra are identical. For example, in Fig. 1, \(AG(\{t_1, t_2, t_3\}) = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5\}\}\), and all the elements in an ambiguity group are tied in the final ranking and hence cannot be uniquely diagnosed as faulty. Therefore, the performance of an SBFL technique increases when there are more ambiguity groups, and their size is smaller.

3 ANALYSIS OF EXISTING DIAGNOSABILITY METRICS

Fig. 1 contains a motivating example that shows the role of a diagnosability metric in FL. In the existing SBFL results based on original test cases \(\{t_1, t_2, t_3\}\), program elements \(e_3\) and \(e_4\) form an ambiguity group. They share the highest suspiciousness score, i.e., 1.0, even though only \(e_3\) is the actual faulty element. In this case, to further distinguish which one is more suspicious between them, we need an additional test case that can break the ambiguity group, i.e., covering either one of the program elements.

Let us now consider the four additional test cases, \(t'_4\) to \(t''_4\), to augment the existing test suite. Adding \(t'_4\) or \(t''_4\) does not provide additional diagnostic information, as the program elements in the same ambiguity group are either executed or not executed together. In comparison, \(t'_4\) and \(t''_4\) can provide more information because they can break an ambiguity group, \(\{e_1, e_2\}\) and \(\{e_3, e_4\}\), respectively. Between them, \(t'_4\) should be prioritised over \(t''_4\) since the ambiguity group \(\{e_3, e_4\}\) is currently more suspicious than \(\{e_1, e_2\}\). Fig. 1 shows how Ochiai scores change as each test case is added to the original test suite. When \(t'_4\) is added, the faulty element \(e_3\) is finally ranked at top as the suspiciousness score of the non-faulty element \(e_4\) decreases. Quantifying such diagnostic contribution made by individual test cases can inform our test suite augmentation, or even the order of test execution.

There are many existing diagnosability metrics that are designed to capture the information gain contributed by individual test cases. In many cases, these metrics are used as part of a test prioritisation technique that is designed to achieve more accurate fault localisation earlier, or with fewer test cases \([16, 19, 58]\). We can also derive such metrics from diagnosability measures of the entire test suites \([8, 44]\) by looking at the difference in diagnosability gain achieved by the addition a single test. To the best of our knowledge, these metrics have never been directly compared to each other using the same benchmark. The remainder of this section describes existing diagnosability metrics, and empirically compares them using DEFECT4J, a collection of real-world faults in Java programs.

3.1 Analysis of Existing Diagnosability Metrics

We analyse nine diagnosability metrics for SBFL from the relevant literature published after 2010. Two of these are simply coverage based Test Case Prioritisation \([37]\) techniques that serve as baselines. Out of the seven remaining metrics, four metrics have been introduced as part of test prioritisation techniques that aim to improve FL performance, while the remaining three are derived from the diagnosability metrics for entire test suites. Given a diagnosability metric \(f\), we use the notation \(f(T, t)\) to denote the estimated diagnosability gain brought by a newly chosen single test case \(t\) to the set of already executed test cases \(T\).

3.1.1 Test Case Prioritisation Metrics. Since SBFL techniques are based on an aggregation of test coverage, we include two coverage-based test case prioritisation techniques, total and additional \([12, 46]\), as baselines. Total prioritisation greedily selects the test case with the highest coverage over all program elements, while additional prioritisation favours the test case with the highest coverage over remaining uncovered program elements only:

\[
\text{Total}(T, t) = |\{e_j \in E|C_T[j] = 1\}|
\]

\[
\text{Add}(T, t) = |\{e_j \in E|C_T[j] = 1 \land \forall t_i \in T. C_T[i, j] = 0\}|
\]

In both cases, our rationale is that increasing coverage as early as possible may lead to an earlier increase in the amount of information provided to SBFL techniques. Reniers and Reiss proposed Nearest Neighbours fault localisation \([45]\), which selects the passing test cases whose execution trace is the closest to the given failing execution. They use a program-dependency-based metric to measure the proximity. Motivated by this work, Bandyopadhyay et al. put weights to test cases using the average of Jaccard similarity between the coverage of a given test case and the failing ones \([5]\). Although Bandyopadhyay et al. do not use this metric to prioritise test cases, we include the metric since it aims to measure the diagnosability of individual test cases. We call this metric as Prox hereafter:

\[
\text{Prox}(T, t) = \frac{\sum_{i \in T} \text{Jaccard}(C_T[i, :], e_T)}{|T_T|}
\]

Hao et al. \([19]\) select a representative subset of test cases from a given test suite to reduce the oracle cost (i.e., to reduce the number of outputs to inspect). Among the three coverage-based strategies
introduced, S1, S2, and S3, the most effective one for fault localisation was S3: it takes into account the relative importance of ambiguity groups, which in turn is defined as the ratio of failing tests that execute each group, \( pr_i(g) = e_j / (e_j + n_j) \).

\[
S3(T, t) = \sum_{g \in AG'(T)} pr_i(g) \cdot \text{div}(t, g)
\]

where \( AG' \) denotes a variation of ambiguity group, which includes program elements that share the same spectrum values, and \( \text{div}(t, g) \) denotes the size of the smaller subset when \( t \) is used to divide \( g \) to two subsets based on coverage, i.e., the minimum between \( |\{e_j \in g|C_j| = 1\}| \) and \( |\{e_j \in g|C_j| = 0\}| \). Therefore, S3 assigns higher priority to tests that can more evenly split the more important ambiguity groups.

RAPTER [16] prioritises test cases by the amount of ambiguity group reduction achieved by individual test cases. While the aim of reducing ambiguity group is similar to S3, RAPTER considers only the size of ambiguity groups, and not the test results. Here, \( p(g) = |g|/n \) is the probability that \( g \) contains a faulty element, which is assigned in proportion to the size of the group, and \( |g|^{-1} \) refers to the expected wasted effort if a developer considers program elements in \( g \) randomly.

\[
\text{RAPTER}(T, t) = \sum_{g \in AG(T \cup \{t\})} p(g) \cdot |g|^{-1}
\]

FLINT [58] is an information-theoretic approach that formulates test case prioritisation as an entropy reduction process. In FLINT, a test case is given higher priority when it is expected to reduce more entropy (H) [49] in the suspiciousness distribution across program elements. The expected entropy reduction of each test case is predicted based on the conditional probability of the test case failing; the probability of a new test case failing is approximated as the failure rate observed among the test cases executed so far.

\[
\text{FLINT}(T, t) = -\alpha \cdot H(P_f) - (1 - \alpha) \cdot H(P_p)
\]

Here, \( \alpha \) is the observed failure rate, \( |T_f|/|T| \), and \( P_p \) and \( P_f \) are the suspiciousness distributions of \( T \cup \{t\} \), computed under the assumption that \( t \) passes and fails, respectively. The suspiciousness distributions are computed by normalising the Tarantula [25] scores. Among the studied existing metrics, FLINT is the only one that does use suspiciousness scores to measure the diagnosability of a new test to execute. However, it only considers the overall distribution of suspiciousness via Shannon entropy, and does not use scores of individual program elements to focus the prioritisation. Note that we negate the original RAPTER and FLINT metrics so that a higher score means a higher diagnosability gain.

3.1.2 Test Suite Diagnosability Metrics. This section describes test suite diagnosability metrics that are designed to measure the diagnosability of entire test suites. Despite originally being designed for test suites, a test suite diagnosability metric \( F \) can also be used to quantify the diagnosability gain of an individual test, \( t \), by computing the difference between the diagnosability of an original test suite, \( F(T) \), and that of the enhanced test suite, \( F(T \cup \{t\}) \):

\[
f(T, t) = F(T \cup \{t\}) - F(T)
\]

\(^1\)S3, on the other hand, considers test results via \( pri(g) \).

Baudry et al. [8] analysed the features of a test suite that are related to the fault diagnosis accuracy and introduced the Test-for-Diagnosis (TfD) metric that measures the number of ambiguity groups (referred to as Dynamic Basic Blocks (DBB) in their paper). They proposed composing a test suite that maximises the number of DBBs for high diagnosability.

\[
\text{TfD}(T) = |AG(T)|
\]

EntBug [11] evaluates a test suite based on its coverage matrix density, which is defined as the ratio of ones in the coverage matrix. EntBug augments an existing test suite with additionally generated test cases with the goal of balancing the density of the coverage matrix to 0.5.

\[
\text{EntBug}(T) = 1 - |1 - 2 \cdot \rho(T)|
\]

where \( \rho(T) = \sum_{C_T[i, j]}/|E| \cdot |T| \). Note that this definition is the normalised version [44] of EntBug.

More recently, Perez et al. propose DDU [44], a test suite diagnosability metric for SBFL, that combines three key properties, density, diversity, and uniqueness, all being properties that a test suite should exhibit to achieve high localisation accuracy. The density component is identical to EntBug, while the uniqueness component is the ratio of the number of ambiguity groups over the number of all program elements, i.e., \( |AG(T)|/|E| \). Lastly, the diversity component is designed to ensure the diversity of test executions, i.e., the contents of the rows in the coverage matrix. Formally, it is defined as the Gini-Simpson index [26] among the rows.

\[
\text{DDU}(T) = \text{density}(T) \times \text{diversity}(T) \times \text{uniqueness}(T)
\]

3.1.3 Classification of Diagnostic Capability Metrics. Among the aforementioned metrics, some can be calculated with only the test coverage, while others require the test results plus the coverage. Thereby, we broadly classify them into two categories based on the utilised information:

- **Result-Agnostic**: metrics utilising only coverage information; total coverage, additional coverage, RAPTER, TfD, EntBug, and DDU belong to this category.
- **Result-Aware**: metrics utilising coverage information and previous test results; Prox, S3, and FLINT belong here.

When we order tests according to diagnosability metrics, the result-agnostic metrics are not affected by the results of previously chosen, whereas the result-aware metrics are.

3.2 Evaluation of Existing Metrics

We empirically compare and analyse the performance of existing metrics by studying how much each metric can accelerate fault localisation of real world bugs. For this, we apply the studied existing metrics to the bugs and human-written test cases in the version 2.0 of DEFECTS4J, a widely studied real world fault benchmark (for details about subject programs and bugs, please refer to Section 5.2). We adopt a ranking-based evaluation protocol stated below: while there are questions about whether the ranking form is the most effective way for humans to consume results of FL techniques [42], we posit that test suites capable of producing more accurate rankings are also capable of providing higher diagnosability due to the diversity in its coverage.
while S3 achieves higher localisation performance after the first ten iterations, note that the similarity to failing tests, shows faster initial convergence, the same number of iterations. Prox, which sorts test cases by the ten test cases are selected, whereas TfD, the best performing mAP value, more efficiently prioritise for coverage and switch to prioritisation for fault localisation once a test fails, whereas our evaluation scenario starts with a failing test case. Consequently, the observed failure rate starts at 1.0 for FLINT, which significantly skews its following analysis.

Among the result-agnostic metrics, Total and EntBug perform identically on our subjects. Total gives higher priority to tests that cover more program elements. Similarly, EntBug also takes into account the number of covered program elements, while aiming to reach the optimal density of 0.5. However, since the level of coverage achieved by individual human-written test cases in our subject is fairly low, EntBug ends up trying to increase the coverage for all ten iterations, which in turn increases the density. Another interesting observation is that TfD outperforms DDU, even though DDU gives high priority to test cases that are either not similar to existing failing test cases or cover more elements, thanks to the diversity and the density metrics. However, the results of Prox show that choosing tests similar to failing tests can be effective, while Total and EntBug show that relying on coverage density alone may not be effective. Based on this, we posit that, when a failing test already exists and is known, it is better to prioritise tests that are similar to the failing test than to focus on test diversity.

In summary, our evaluation using DEFECTS4J shows that two result-aware metrics, S3 and Prox, outperform all result-agnostic metrics. When diagnosing faults that have already been observed by failing executions, result-agnostic metrics cannot efficiently prioritise test cases because they cannot distinguish the more suspicious elements from those that are less so.

### 4 Fault Diagnosability Gain

This section proposes a novel diagnosability metric, FDG (Fault Diagnosability Gain), that better measures the fault diagnosability gain by leveraging the ongoing FL results during the prioritisation.

#### 4.1 Design of FDG (Fault Diagnosability Gain)

Our new diagnosability metric, FDG, consists of two subcomponents, Split and Cover: Split is designed to break more suspicious ambiguity groups, and Cover is designed to focus on covering more suspicious elements.

##### 4.1.1 Breaking the Suspicious Ambiguity Groups

Split measures the expected wasted localisation effort when a new test is added to T. It extends RAPTER by weighting each ambiguity group using the previous test results as in S3, instead of the size of the ambiguity group. However, while S3 only considers the number of failing test cases, Split uses suspiciousness scores, which take into account both the number of failing test cases and passing ones.

Let us first define the probability of an ambiguity group containing the fault, \( p(g) \), as the sum of the probabilities of its elements being faulty:

\[
p(g) = \sum_{e_j \in g} p(e_j)
\]
where $p(e_j)$ refers to the probability of each program element $e_j$ being faulty. We obtain $p(e_j)$ by applying the `softmax` function on the suspiciousness scores calculated using $T$, i.e., $p(e_j) = w_j / \sum_i w_i$, where $w_i$ is the suspiciousness score of $e_j$. Using $p(g)$, Split is defined as follows:

$$\text{Split}(T, t) = 1 - \frac{1}{n-1} \sum_{g \in AG(T \cup \{t\})} p(g) \cdot \left(\frac{|g| - 1}{2}\right) \left(\frac{n-1}{2}\right)$$

Here, $(n-1)/2$ is the normalisation constant, which is the maximum localisation effort when all program elements belong to a single ambiguity group. In general, if $T \cup \{t\}$ produces larger and more suspicious ambiguity groups, Split will penalise $t$ more heavily.

### 4.1.3 A Combined Metric, FDG

As both Split and Cover are normalised, we define FDG as the weighted sum of both:

$$\text{FDG}(T, t) = \alpha \cdot \text{Split}(T, t) + (1 - \alpha) \cdot \text{Cover}(T, t)$$

The coefficient $\alpha$ can be tuned using the information of known faults: we study the impact of $\alpha$ with RQ4 (see Section 5.1).

Figure 3: Overview of Iterative Fault Localisation with Test Augmentation

Figure 4: An EvoSuite-generated test case for the class FastMath of Math-59 in Defects4J

4.2 Iterative Fault Localisation with Test Augmentation

Let us propose an Iterative Fault Localisation (IFL) scenario, in which FDG is used to guide the augmentation of the test suite: an overview of the proposed scenario is presented in Fig. 3.

First, an automated test generation tool, such as EvoSuite [13] or Randoop [59], produces regression tests for the given faulty program (Test Generation): to reduce the number of generated test cases, we limit the scope of regression test generation only to the initial failing test cases. Since regression tests simply capture and record the current behaviour of the program, some assertions in the generated tests may capture the buggy behaviour of the program [43]. Note that the regression test cases generated by EvoSuite always capture the behaviour of the current system as identity assertions [14]. Fig. 4 shows an example of a test case automatically generated for the Defects4J Math-59 buggy version by EvoSuite: when calling `FastMath.max(0.0F, (-1653.0F))` (Line 3), the assertion generated from the buggy program expects the return value to be ~1653.0F (Line 4), and not 0.0F, which is the buggy behaviour of Math-59.

Consequently, unless the failure is detectable by implicit oracles [7] such as program crashes or uncaught exceptions, a human engineer has to determine whether the program behaviour captured in the test case is correct or incorrect (Human Labelling, or Oracle Elicitation). For example, an engineer will say that the test case in Fig. 4 is capturing the incorrect behaviour. However, since manual oracle elicitation can be costly, tests should be presented to the engineer in the order of their relative diagnostic capabilities so that the more relevant tests to the fault localisation are labelled earlier. Therefore, at each iteration, all remaining test cases in the generated test suite are prioritised based on their diagnosability gain, and the best one is selected to be labelled by the engineer. (Test Prioritisation) Note that the only cost not shown in Fig. 3 is that of measuring the coverage of generated tests, which we expect to be automatable for EvoSuite generated tests.

Algorithm 1 formally describes the workflow. The test generation tool, TestGenerator, generates tests $T'$ for the suspicious part $P_{\text{sus}}$ of a given faulty program $P$ within the time budget $b_t$ (Line 1-2). Then, until either the oracle querying budget, $b_q$, is exhausted or $T'$ becomes empty, a test case $t_i$ with the maximum diagnosability value is iteratively selected from $T'$ (Line 4-5). If the test $t_i$ reveals buggy behaviour of a program, the user labels it as Incorrect (i.e., $R(t_i) = 0$), or otherwise as Correct (i.e., $R(t_i) = 1$) (Line 6). Once labelled, the test is moved from $T'$ to $T$.
We ask the following research questions to evaluate both our newly
proposed diagnosability gain metric f, Test generation time budget θ,
when used in the IFL scenario described in Section
5.1 Research Questions

5.1.1 RQ1. Effectiveness. How effective is FDG when compared
to the existing metrics? To answer RQ1, we use the same eval-
uation protocol used for existing metrics, and compare the results
produced by FDG to those produced by existing metrics. RQ1 is
designed to investigate the performance of FDG in an ideal situation
with the perfect knowledge of coverage of all test cases as well as
human-written test oracles. While some of the assumptions may
be infeasible in practice, we aim to establish the upper bound of
FDG in an ideal setting, as a baseline to our subsequent evaluation
of IFL scenario. We set α to 0.5 for FDG.

In addition to mAP, we also report the Top-n Accuracy (acc@n),
a widely-adopted measure of fault localisation performance. For
all faulty subjects, acc@n counts the number of subjects where at
least one of the faulty program elements are ranked within the top
n locations. As in mAP, we report acc@n after each iteration using
the test cases chosen up to that point. Where there are multiple test
orderings for a single DEFECTS4J bug due to multiple failing test
cases (see Section 3.2.1), we average all relevant acc@n values.

5.1.2 RQ2. IFL Performance. How effectively does FDG facilit-
ate fault localisation by prioritising automatically generated
test cases? RQ2 aims to investigate the performance of FDG
when used in the IFL scenario described in Section 4.2. We conduct
IFL as shown in Algorithm 1, starting with an initial test suite, T,
which contains only the DEFECTS4J-provided failing test cases. Our
IFL scenario for RQ2 consists of the following elements:

Test Generation: We employ EvoSuite [13] version 1.0.71 as a test
data generation tool. The target of regression test generation is
restricted to only the methods covered by initial failing test cases, as
specified in the property relevant.classes in DEFECTS4J. We use
two time budgets (θ1), 3 and 10 minutes, and allocate the physical
time budget to a class proportionally to the number of suspicious
methods in the class. For example, consider suspicious classes A
and B that contain five and ten suspicious methods, respectively.
Under the 3-minute budget, we allocate 1 and 2 minutes to class A
and B, respectively. We generate 10 test suites for each fault and
time budget to cater for the randomness of EvoSuite.

Coverage & FL: We measure the coverage achieved by generated
test cases against all suspicious classes using Cobertura. We perform
method level SBFL using Ochiai as described in Section 3.2.1.

Test Prioritisation: FDG (with α = 0.5) is used as the test prioritisation
metric f in Line 5 of Algorithm 1.

Human Labelling: We simulate a perfect human oracle querying by
running the generated test cases on the fixed version. If a regression
test case fails on the fixed version (due to oracle violation or compile
error), we consider the test as a failing test case for the buggy
version. We report results from {1, 3, 5, 10} iterations by setting
the oracle querying budget, θb, accordingly.

5.1.3 RQ3. Robustness. How robust is FDG against human er-
rors, when used to guide oracle querying? Our third research
question concerns the assumption of the perfect human oracle
because, in reality, the human engineer may make incorrect judge-
ments about the generated oracles that capture the current
behaviour of a buggy program. To study how robust our IFL scenario
is against such mistakes, we simulate labelling errors by apply-
ing the probability p ∈ {0.1, 0.3, 0.5, 0.7, 0.9} of flipping the perfect
judgement, and observe how much the localisation performance
deteriorates.

5.1.4 RQ4. Parameter Tuning. What is the ideal parameter α
for FDG? FDG has a parameter α that adjusts the relative weights
of Split and Cover. For previous research questions, α has been set
to 0.5 to assume equal contribution from Split and Cover to the mea-
sure of diagnosability gain and, consequently, to the prioritisating.
Our final research question studies the impact of the parameter α
on the performance of FDG. To answer RQ4, we perform a grid
search for α ∈ {0.1, 0.3, 0.5, 0.7, 0.9} to find its value that leads to
the best prioritisation performance for EvoSuite-generated tests.

5.2 Subject

Our empirical evaluation considers 357 faults in five different projects
provided by DEFECTS4J version 2.0.0 [27], a real-world fault bench-
mark of Java programs. Each fault in DEFECTS4J is in the program
source code, not in the configuration nor test files, and the cor-
responding patch is provided as a fixing commit. For each faulty
program, human-written test cases, at least one of which is bound
to fail due to the fault, are provided. The failing tests all pass once

\[ \text{Algorithm 1: Iterative FL with Test Augmentation} \]

\begin{verbatim}
Input: Faulty program P, Initial test suite T, Test results R,
Test generation tool TestGenerator, SBFL formula FaultLocaliser, Diagnosability gain metric f, Test generation time budget θ,
Querying budget θq
Precondition: ∃ t ∈ T: R(t) = 0
Output: Fault localisation result
1 P_susp ← GetSuspiciousParts(T, R)
2 T′ ← TestGenerator(P_susp, θ)
3 i ← 1
4 while i ≤ θq ∧ T′ ≠ ∅ do
5    t_s ← argmax_{t′ ∈ T′} f(T, t)
6    R(t_s) ← GetHumanLabel(t_s)
7    T ← T ∪ {t_s}
8    T′ ← T′ \ {t_s}
9    i ← i + 1
10 return FaultLocaliser(T, R)
\end{verbatim}

(Line 7-8). Finally, after the loop terminates, the final FL result is
returned (Line 11).

5 EXPERIMENTAL SETUP

This section presents the research questions and describes the setup
and the environment of our empirical evaluation.

We ask the following research questions to evaluate both our newly
proposed diagnosability gain metric, FDG, and our IFL approach.

5.1.1 RQ1. Effectiveness. How effective is FDG when compared
to the existing metrics? To answer RQ1, we use the same eval-
uation protocol used for existing metrics, and compare the results
produced by FDG to those produced by existing metrics. RQ1 is
designed to investigate the performance of FDG in an ideal situation
with the perfect knowledge of coverage of all test cases as well as
human-written test oracles. While some of the assumptions may
be infeasible in practice, we aim to establish the upper bound of
FDG in an ideal setting, as a baseline to our subsequent evaluation
of IFL scenario. We set α to 0.5 for FDG.

In addition to mAP, we also report the Top-n Accuracy (acc@n),
a widely-adopted measure of fault localisation performance. For
all faulty subjects, acc@n counts the number of subjects where at
least one of the faulty program elements are ranked within the top
n locations. As in mAP, we report acc@n after each iteration using
the test cases chosen up to that point. Where there are multiple test
orderings for a single DEFECTS4J bug due to multiple failing test
cases (see Section 3.2.1), we average all relevant acc@n values.

5.1.2 RQ2. IFL Performance. How effectively does FDG facilit-
ate fault localisation by prioritising automatically generated
test cases? RQ2 aims to investigate the performance of FDG
when used in the IFL scenario described in Section 4.2. We conduct
IFL as shown in Algorithm 1, starting with an initial test suite, T,
which contains only the DEFECTS4J-provided failing test cases. Our
IFL scenario for RQ2 consists of the following elements:

Test Generation: We employ EvoSuite [13] version 1.0.71 as a test
data generation tool. The target of regression test generation is
restricted to only the methods covered by initial failing test cases, as
specified in the property relevant.classes in DEFECTS4J. We use
two time budgets (θ1), 3 and 10 minutes, and allocate the physical
time budget to a class proportionally to the number of suspicious
methods in the class. For example, consider suspicious classes A
and B that contain five and ten suspicious methods, respectively.
Under the 3-minute budget, we allocate 1 and 2 minutes to class A
and B, respectively. We generate 10 test suites for each fault and
time budget to cater for the randomness of EvoSuite.

Coverage & FL: We measure the coverage achieved by generated
test cases against all suspicious classes using Cobertura. We perform
method level SBFL using Ochiai as described in Section 3.2.1.

Test Prioritisation: FDG (with α = 0.5) is used as the test prioritisation
metric f in Line 5 of Algorithm 1.

Human Labelling: We simulate a perfect human oracle querying by
running the generated test cases on the fixed version. If a regression
test case fails on the fixed version (due to oracle violation or compile
error), we consider the test as a failing test case for the buggy
version. We report results from {1, 3, 5, 10} iterations by setting
the oracle querying budget, θb, accordingly.

5.1.3 RQ3. Robustness. How robust is FDG against human er-
rors, when used to guide oracle querying? Our third research
question concerns the assumption of the perfect human oracle
because, in reality, the human engineer may make incorrect judge-
ments about the generated oracles that capture the current
behaviour of a buggy program. To study how robust our IFL scenario
is against such mistakes, we simulate labelling errors by apply-
ing the probability p ∈ {0.1, 0.3, 0.5, 0.7, 0.9} of flipping the perfect
judgement, and observe how much the localisation performance
deteriorates.

5.1.4 RQ4. Parameter Tuning. What is the ideal parameter α
for FDG? FDG has a parameter α that adjusts the relative weights
of Split and Cover. For previous research questions, α has been set
to 0.5 to assume equal contribution from Split and Cover to the mea-
sure of diagnosability gain and, consequently, to the prioritisating.
Our final research question studies the impact of the parameter α
on the performance of FDG. To answer RQ4, we perform a grid
search for α ∈ {0.1, 0.3, 0.5, 0.7, 0.9} to find its value that leads to
the best prioritisation performance for EvoSuite-generated tests.

5.2 Subject

Our empirical evaluation considers 357 faults in five different projects
provided by DEFECTS4J version 2.0.0 [27], a real-world fault bench-
mark of Java programs. Each fault in DEFECTS4J is in the program
source code, not in the configuration nor test files, and the cor-
responding patch is provided as a fixing commit. For each faulty
program, human-written test cases, at least one of which is bound
to fail due to the fault, are provided. The failing tests all pass once

\[ \text{This is not an official release but is the most recent version on GitHub (commit 080e12). For each fault, the maximum number of tests per class is set to 200, and the} \]

\[ \text{length of a test case is limited to 20 to avoid too complex test cases being generated, which is difficult for engineers to investigate.} \]
the fixing commit is applied to the faulty version. Table 1 shows the statistics of our subjects. The Suspicious column contains the average number of methods covered by at least one failing test.

For RQ1, we exclude a total of six faults from the 357 in Table 1: two omission faults, Lang-23 and Lang-56, as they cannot be localised within the faulty version by SBFL techniques, as well as four deprecated faults (Lang-2, Time-21, Closure-63, and Closure-93).

For RQ2 to RQ4, we exclude two additional faults, Time-5 and Closure-105, because the DEFECTS4J-provided lists of relevant (suspicious) classes of those faults do not contain the actual faulty class due to an unknown reason. Overall, 349 faults are used, with 2.3 failing test cases on average per fault in the initial test suites.

### 5.3 Implementation & Environment

All our experiment have been performed on machines equipped with Intel Core i7-7700 CPU and 32GB memory, running Ubuntu 16.04. We use Java 8 for EvoSuite and Cobertura. Our replication package, available online at https://github.com/agb94/FDG-artifact [2], includes all implementation as well as the data required to reproduce our work.

### 6 RESULTS

This section presents the answers to our research questions based on the results of our empirical evaluations.

#### 6.1 RQ1: Effectiveness

Table 2 shows the localisation performance of each metric on DEFECTS4J human-written tests. Each row presents acc@n values (n ∈ [1, 3, 5, 10]) of SBFL results at each iteration; rows Init. and Full represent the results when using a single initial failing test case and full test suite (about 4.2K tests on average), respectively. The numbers in bold represent the highest values in the iteration for the corresponding n value. Additionally, the mAP values after ten iterations are shown next to the name of each metric.

The results show that FDG outperforms all nine studied metrics by achieving the highest acc@3, 5, 10 values in all iterations except for the first. Especially at the tenth iteration, acc@10 is 23% and 14% higher than those obtained by the state-of-the-art metrics, S3 and Prox, respectively. Furthermore, we report the performance of Split and Cover as stand-alone metrics. Although they do not perform as well as FDG, the two subcomponents still achieve higher

#### 6.2 RQ2: IFL Performance

Table 3 shows the average number of total and failing test cases generated by EvoSuite for the studied faults, using 3 and 10 minutes as time budgets, b1, respectively (hereafter denoted by T3 and T10). These tests correspond to T′ in Algorithm 1. Using these generated test cases, we evaluate fault localisation performance after a different number of oracle queries.

Fig. 5 shows how the acc@n values change as the query budget increases for T3 (top) and T10 (bottom) scenarios. Overall, T10 has slightly better localisation results and less standard deviation in performance than T3, although the difference is not significant. This is as expected, for T10 is larger, thus more likely to contain diverse test cases. We note that the effectiveness of test suite augmentation

<table>
<thead>
<tr>
<th>Subject</th>
<th># Faults</th>
<th>kLoC</th>
<th>Avg. # Tests</th>
<th>Avg. # Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Full</td>
</tr>
<tr>
<td>Commons-lang</td>
<td>65</td>
<td>22</td>
<td>1,527</td>
<td>1.9</td>
</tr>
<tr>
<td>Commons-math</td>
<td>106</td>
<td>85</td>
<td>2,713</td>
<td>1.7</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>26</td>
<td>96</td>
<td>4,903</td>
<td>3.5</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>27</td>
<td>28</td>
<td>1,946</td>
<td>2.8</td>
</tr>
<tr>
<td>Closure</td>
<td>133</td>
<td>90</td>
<td>5,038</td>
<td>2.6</td>
</tr>
<tr>
<td>Total</td>
<td>357</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th># Faults</th>
<th>kLoC</th>
<th>Avg. # Tests</th>
<th>Avg. # Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Full</td>
</tr>
<tr>
<td>Commons-lang</td>
<td>65</td>
<td>22</td>
<td>1,527</td>
<td>1.9</td>
</tr>
<tr>
<td>Commons-math</td>
<td>106</td>
<td>85</td>
<td>2,713</td>
<td>1.7</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>26</td>
<td>96</td>
<td>4,903</td>
<td>3.5</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>27</td>
<td>28</td>
<td>1,946</td>
<td>2.8</td>
</tr>
<tr>
<td>Closure</td>
<td>133</td>
<td>90</td>
<td>5,038</td>
<td>2.6</td>
</tr>
<tr>
<td>Total</td>
<td>357</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th># Faults</th>
<th>kLoC</th>
<th>Avg. # Tests</th>
<th>Avg. # Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Full</td>
</tr>
<tr>
<td>Commons-lang</td>
<td>65</td>
<td>22</td>
<td>1,527</td>
<td>1.9</td>
</tr>
<tr>
<td>Commons-math</td>
<td>106</td>
<td>85</td>
<td>2,713</td>
<td>1.7</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>26</td>
<td>96</td>
<td>4,903</td>
<td>3.5</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>27</td>
<td>28</td>
<td>1,946</td>
<td>2.8</td>
</tr>
<tr>
<td>Closure</td>
<td>133</td>
<td>90</td>
<td>5,038</td>
<td>2.6</td>
</tr>
<tr>
<td>Total</td>
<td>357</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The error bars show the standard error from different seeds. (Total subjects = 349)

Answer to RQ1: FDG significantly outperforms all nine studied metrics achieving the highest acc@n values at almost every iteration. When ten additional test cases are selected, FDG shows at least 14% higher acc@10 compared to other metrics.

mAP values than all existing metrics after ten iterations, showing each of them can outperform the existing metrics as a stand-alone diagnosability metric.

Additionally, we also compare the FL accuracy at the tenth iteration to the accuracy obtained using the initial test set only, as well as the full test suites. After adding ten test cases based on FDG, acc@1 and acc@10 values are increased by 17.0 and 3.4 times, respectively, compared to the initial test set. Notably, FDG achieves 62% of acc@1 and 80% of acc@10 compared to the full test suite, after ten iterations, despite the fact that the full test suites have approximately 380 times (≈4.2K/11) more tests than the 11 test cases obtained using FDG (one initial + ten additional).

Answer to RQ1: FDG significantly outperforms all nine studied metrics achieving the highest acc@n values at almost every iteration. When ten additional test cases are selected, FDG shows at least 14% higher acc@10 compared to other metrics.

Figure 5: The acc@n values for each time and query budgets. The error bars show the standard error from different seeds. (# total subjects = 349)
with automatically generated test cases is less than that of augmentation with human-written test cases. However, augmenting the test suite with generated test cases also increases the SBFL accuracy significantly. The test suite with ten newly labelled test cases achieves 11.6% acc@1 (± 58/5) and 2.2% acc@10 (± 147/68) compared to those of the initial test suite (# queries = 0).

Interestingly, we note that accurate fault localisation does not always require many additional failing test cases to be generated.

Table 2: The acc@n values with the selected human-written test cases of DEFECT4J at each iteration (# total subjects = 351)

<table>
<thead>
<tr>
<th>Metric</th>
<th>EntBug (mAP=0.119)</th>
<th>Total (mAP=0.119)</th>
<th>DDU (mAP=0.149)</th>
<th>Add. (mAP=0.158)</th>
<th>RAPTER (mAP=0.158)</th>
<th>DDU (mAP=0.158)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init.</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
</tr>
<tr>
<td>Iter.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9 40 61 80</td>
<td>9 40 61 80</td>
<td>10 43 64 85</td>
<td>9 39 60 80</td>
<td>9 39 60 80</td>
<td>9 39 60 80</td>
</tr>
<tr>
<td>2</td>
<td>9 41 63 85</td>
<td>9 41 63 85</td>
<td>12 49 74 99</td>
<td>10 44 67 94</td>
<td>10 44 67 94</td>
<td>28 67 81 100</td>
</tr>
<tr>
<td>3</td>
<td>9 40 64 90</td>
<td>9 40 64 90</td>
<td>14 53 76 107</td>
<td>11 47 71 99</td>
<td>12 48 72 100</td>
<td>32 71 88 112</td>
</tr>
<tr>
<td>4</td>
<td>9 40 64 94</td>
<td>9 40 64 94</td>
<td>15 54 77 112</td>
<td>12 52 76 104</td>
<td>13 53 76 104</td>
<td>33 81 95 127</td>
</tr>
<tr>
<td>5</td>
<td>9 41 67 95</td>
<td>9 41 67 95</td>
<td>15 54 79 115</td>
<td>12 54 80 112</td>
<td>13 51 81 112</td>
<td>35 81 95 130</td>
</tr>
<tr>
<td>6</td>
<td>9 41 67 96</td>
<td>9 41 67 96</td>
<td>15 55 79 118</td>
<td>14 56 82 115</td>
<td>16 56 83 117</td>
<td>35 87 100 134</td>
</tr>
<tr>
<td>7</td>
<td>9 41 68 96</td>
<td>9 41 68 96</td>
<td>15 56 81 123</td>
<td>15 57 85 119</td>
<td>16 57 84 117</td>
<td>36 89 105 142</td>
</tr>
<tr>
<td>8</td>
<td>9 41 68 99</td>
<td>9 41 68 99</td>
<td>15 56 81 125</td>
<td>17 60 86 121</td>
<td>18 59 85 123</td>
<td>37 89 109 145</td>
</tr>
<tr>
<td>9</td>
<td>11 43 68 99</td>
<td>11 43 68 99</td>
<td>16 58 82 127</td>
<td>19 62 87 125</td>
<td>20 61 87 126</td>
<td>37 91 111 149</td>
</tr>
<tr>
<td>10</td>
<td>12 44 68 101</td>
<td>12 44 68 101</td>
<td>17 59 84 127</td>
<td>19 65 88 127</td>
<td>20 63 87 127</td>
<td>39 93 111 149</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>FLINT (mAP=0.113)</th>
<th>Prox (mAP=0.250)</th>
<th>S3 (mAP=0.271)</th>
<th>Split (mAP=0.277)</th>
<th>Cover (mAP=0.278)</th>
<th>FDG (mAP=0.298)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init.</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
<td>4 30 45 65</td>
</tr>
<tr>
<td>Iter.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4 31 47 69</td>
<td>27 71 97 130</td>
<td>22 51 69 92</td>
<td>22 51 69 92</td>
<td>25 75 94 122</td>
<td>24 64 87 109</td>
</tr>
<tr>
<td>2</td>
<td>8 45 66 79</td>
<td>34 87 118 145</td>
<td>32 70 93 124</td>
<td>31 70 92 121</td>
<td>35 93 113 145</td>
<td>41 100 124 155</td>
</tr>
<tr>
<td>3</td>
<td>8 47 68 79</td>
<td>40 97 122 158</td>
<td>40 88 108 137</td>
<td>40 91 112 146</td>
<td>46 105 130 164</td>
<td>48 107 135 175</td>
</tr>
<tr>
<td>4</td>
<td>8 48 69 81</td>
<td>42 107 130 168</td>
<td>47 97 124 151</td>
<td>46 106 126 153</td>
<td>53 114 145 181</td>
<td>52 115 145 190</td>
</tr>
<tr>
<td>5</td>
<td>8 48 70 82</td>
<td>45 109 137 174</td>
<td>54 105 128 158</td>
<td>52 114 135 160</td>
<td>54 121 154 188</td>
<td>57 121 158 204</td>
</tr>
<tr>
<td>6</td>
<td>9 49 72 83</td>
<td>47 112 140 182</td>
<td>56 113 135 160</td>
<td>59 122 141 164</td>
<td>55 128 155 192</td>
<td>60 130 164 212</td>
</tr>
<tr>
<td>7</td>
<td>9 49 72 86</td>
<td>47 115 144 185</td>
<td>63 118 141 164</td>
<td>60 125 144 171</td>
<td>57 131 162 198</td>
<td>64 135 170 221</td>
</tr>
<tr>
<td>8</td>
<td>9 50 73 86</td>
<td>48 120 153 188</td>
<td>66 120 146 171</td>
<td>62 129 151 176</td>
<td>61 133 167 204</td>
<td>65 139 174 219</td>
</tr>
<tr>
<td>9</td>
<td>11 52 75 86</td>
<td>49 122 155 190</td>
<td>66 124 149 179</td>
<td>64 130 153 178</td>
<td>62 133 175 209</td>
<td>66 145 178 220</td>
</tr>
<tr>
<td>10</td>
<td>11 52 75 86</td>
<td>50 125 160 195</td>
<td>70 127 153 180</td>
<td>68 134 157 189</td>
<td>63 139 170 212</td>
<td>68 148 179 222</td>
</tr>
</tbody>
</table>

Table 3: Average number of tests generated by EvoSuite (number of failing tests in parenthesis)

<table>
<thead>
<tr>
<th>Time</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>Lang</td>
</tr>
<tr>
<td>3 mins</td>
<td>32 (1.0)</td>
</tr>
<tr>
<td>10 mins</td>
<td>34 (1.1)</td>
</tr>
</tbody>
</table>

The localisation results tend to be positive despite the small number of failing test cases that have been generated (see Table 3). This is because generated passing test cases can still effectively contribute to decreasing the suspiciousness of non-faulty program elements.

Answer to RQ2: Given a small set of failing test cases only, the accuracy of SBFL can be greatly improved by querying human engineers about the oracles for a small number of automatically generated tests prioritised by FDG.

6.3 RQ3: Robustness

Fig. 6 shows how localisation performance varies against different labelling error rates. The results show that localisation performance, acc@10, gradually deteriorates as the error rate (p) increases. However, we observe that the localisation results are fairly robust up to the error rate of $p = 0.3$: the acc@10 values still increase as the query budget increases. Based on the report from Pastore et al. [43] that shows a qualified crowd can achieve accuracy above 0.69 when
classifying incorrect assertions for EvoSuite-generated test cases, the error rate tolerance of up to 0.3 shows the robustness of FDG.

Overall, the results show that, despite some errors in labelling, adding new test cases significantly increases the SBFL accuracy compared to the initial test suite. This is because the correctly labelled subsequent test cases can mitigate the impact of earlier labelling errors. Further, we observe that T10 is slightly more robust than T3 against the same error rate. This suggests that the richer labelling errors than smaller test suites.

Answer to RQ3: Although SBFL performance deteriorates as the labelling error rate increases, the results show that FDG can be resilient to labelling errors, especially up to the error rate of 30%. A richer and larger test suites are also more resilient to labelling errors than smaller test suites.

6.4 RQ4: Parameter Tuning
Table 4 shows the performance of FDG obtained while varying $\alpha$ from 0.1 to 0.9 with the interval of 0.2: a value higher than 0.5 means that FDG puts more emphasis on Split, while lower than 0.5 puts more emphasis on Cover. When only one query is made, FL performance measured using acc@10 is best at $\alpha = 0.5$. However, as more queries are made, the higher $\alpha$ values, e.g. 0.7, 0.9, or 1.0, tend to produce better performance. These results show that it is better to balance the weights of two metrics ($\alpha = 0.5$) when the query budget is low, whereas it is better to focus on splitting the ambiguity groups ($\alpha > 0.5$) when there are multiple chances to query test oracles from developers. Since both program structure and test suite composition can significantly affect the ambiguity groups, we suggest that $\alpha$ needs to be tuned based on actual fault data when used in practice. However, the results also show that FDG can outperform state-of-the-art diagnosability metrics even with the default value of $\alpha = 0.5$.

Answer to RQ4: When oracle querying is performed only once, FL performance increases the most when using FDG with $\alpha = 0.5$. However, as more queries are made, the higher $\alpha$ values lead to higher FL performance improvement.

7 DISCUSSION
We discuss some assumptions and implications related to FDG.

7.1 Rankings and FL
Whether reporting in the form of linear rankings is the best vessel for fault localisation remains under debate. Parnin and Orso state that human developers do not really follow a given ranking linearly while debugging [42], while Xia et al. state that they do help, especially if they are of high quality [55]. In this work, we simply adopt the ranking based evaluation as one possible way of quantitatively measuring the diagnosability gain.

Note that higher SBFL rankings mean more diverse patterns and fewer ambiguity groups in the achieved coverage, both of which contribute to the general diagnosability of a test suite. In fact, it is entirely possible that the human labelling phase (see Section 4.2) can be the debugging activity itself, as the labeller will have to consider program behaviour and test results for the given task. However, even in such a scenario, we expect that FDG will still prioritise test results efficiently for the labeller. Such a human study would be a very interesting future work.

7.2 Retaining Generated Test Cases
The test suite augmentation scenario proposed in Section 4.2 assumes that 1) there is no other test case apart from the failing test cases, and 2) the new test cases are generated for the purpose of FL only. What if there already exists a test suite, but the newly generated test cases can further improve its diagnosability? The generated test cases do not have to be single-use only: once labelled, selected test cases can be retained to be part of the existing test suites, as long as the testing budget can afford them.

7.3 FDG as Test Generation Objective
Instead of prioritising already generated test cases using FDG, it may also be possible to use to directly adopt FDG as a fitness (or an objective) function that guides test case generation. Note that FDG itself can be considered as a weighted-sum of Cover and Split, which will try to achieve coverage and break ambiguity groups.
respectively. As such, we expect that search-based test data generation guided by FDG may yield test suites with high coverage and high diagnosability.

8 RELATED WORK
We have discussed existing work on diagnosability metrics for test cases in Section 3.1. This section further discusses fault localisation techniques that involve human-in-the-loop, as well as a test suite augmentation for fault localisation and algorithmic debugging, which are related to our IFL scenario in Section 4.2.

8.1 Human-in-the-Loop Debugging
Many debugging techniques depend on user feedback. Types of required feedback include the correctness of a specific program element [6, 15], the correctness of a specific variable value [35], the correctness of a specific execution [32], and the relative location of the fault with respect to breakpoints [20]; some techniques provide visual feedback [21, 28, 29]. FDG requires the user to check the automatically generated test assertions, which is similar to checking the correctness of executions [32]. However, the main contribution of FDG lies in the way it chooses which test to get the feedback about. ENLIGHTEN [34] asks the developer to investigate input-output pairs of the most suspicious method invocations: the feedback is then encoded as a virtual test and used to update the localisation results. Recently, Böhme et al. proposed Learn2Fix [9], a human-in-the-loop program repair technique. Learn2Fix generates test cases by mutating the failing test and allowing the user to label them in the order of their likelihood of failing. Learn2Fix trains an automatic bug oracle using user feedback, which is in turn used to amplify the test suite for better patch generation. While our iterative localisation scenario also requires human interaction, the scenario differs from existing techniques in that it only requires judgements on the correctness of automatically generated assertions.

8.2 Test Augmentation for Fault Localisation
Automated test generation has been widely used to support fault localisation by augmenting insufficient test suites. Artzi et al. [3] used dynamic symbolic execution to generate test cases that are similar to failing executions [45]. This type of methodology also can be plugged into the part of our debugging scenario as a test generation tool. BUGEX [47] generates additional test cases similar to a given failing one using EvoSUITE and uses an automated oracle to differentiate between passing and failing executions. Finally, BUGEX identifies the runtime properties that are relevant to the failure by comparing passing and failing executions. Compared to BUGEX, our debugging scenario does not assume the existence of an automated bug oracle and instead uses test prioritisation to reduce the cost of querying oracles from a developer. $F^3$ [24] is a fault localisation technique for field failures, which extends a bug reproduction technique, BugRedux [23]; it synthesises failing and passing executions similar to the field failure and uses them for its customised fault localisation technique. While $F^3$ can only debug program crashes that can be detected implicitly, our scenario aims to localise faults where no such automatic oracle is available.

Xuan et al. [56] split test cases into smaller test cases to increase the fault diagnosability of the given test suite. In comparison, we consider cases in which only a few failing test cases are available and use an automated test data generation technique to support fault localisation. Recently, Kuma et al. [30] presented several strategies for selecting the minimum number of test cases out of many automatically generated test cases. However, they only focus on differences in a program spectrum and do not quantify the diagnosability of test cases, which is why we exclude this work from our analysis. Moreover, Kuma et al. use the behaviour of the past version as the test oracle, which may limit its applicability due to limited access to the past versions or version compatibility issues.

8.3 Algorithmic Debugging
Algorithmic debugging [10] was first proposed by Shapiro [50] for logic programming languages such as PROLOG. Shapiro introduces an approach that can systemically narrow down the cause of incorrect computation using what is called a debugging tree, whose root node corresponds to the final outcome of the computation, and each subtree represents an intermediate computation whose result is used to compute its parent node. The debugger can systematically guide the user through the debugging tree, narrowing down the possible root cause of the incorrect computation. FDG shares a few similarities with algorithmic debugging: the user is expected to judge the correctness of the final computation (i.e., the test oracle), and the technique tries to systematically narrow down the root cause by breaking ambiguity groups. However, FDG does not require the user to make any judgement about any target program elements, allowing anyone who understands the input-output specification to use it.

9 THREATS TO VALIDITY
Threats to internal validity regard factors that may influence the observed effects, such as the integrity of the coverage and test results data, as well as the test data generation and fault localisation. To mitigate such threats, we have used the widely studied Cobertura and EvoSUITE, as well as the publicly available scripts in DEFECTS4J, to collect or generate data.

Threats to external validity concern any factors that may limit the generalisation of our results. Our results are based on DEFECTS4J, a benchmark against which many fault localisation techniques are evaluated. However, only further experimentations using more diverse subject programs and faults can strengthen the generalisability of our claims. We also consider an iterative scenario, in which tests are chosen and added one by one. It is possible that non-constructive heuristics such as Genetic Algorithm may produce different and potentially better orderings. Our approach is based on the assumption that an iterative process will make it easier for the human engineer to make the oracle judgement.

Finally, threats to construct validity concern situations where used metrics may not reflect the actual properties they claim to measure. All evaluation metrics used in our study are widely used in fault localisation literature, leaving little room for misunderstanding. It is possible that Coincidental Correctness (CC) [36] has interfered with our measurements, as it is known to exist in DEFECTS4J [4]. However, it is theoretically impossible to entirely filter out CC from test results. We also note that all coverage-based fault localisation techniques are equally affected by CC.
10 CONCLUSION

We propose FDG, a novel measure of the diagnosability of test cases for SBFL. When evaluated with the human-written test cases in DEFECT4J, FDG can successfully augment single failing test cases, so that adding only ten more test cases improves acc@10 by 3.4 times. The augmentation based on FDG also achieves 23% and 14% higher acc@10 when compared to those based on state-of-the-art metrics, S3 and Prox, respectively. We also introduce an iterative fault localization scenario, in which the localization starts with insufficient test suites that contain only failing test cases. By automatically generating test cases using EvoSuite, and prioritising them for human oracle judgements using FDG, we show that acc@1 can be improved by 11.6 times after only ten human interactions. Future work will consider closer integration of FDG and test data generation to improve the overall efficiency of our approach. In particular, we expect that FDG can provide guidance for search based test data generation to improve the diagnosability of the generated test cases.

ACKNOWLEDGMENT

Gabin An and Shin Yoo are supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001), and Samsung Electronics (Grant No. I0201210-07969-01).

REFERENCES


