Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing

Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong.

1 INTRODUCTION

Search-based software testing (SBST) transforms the problem of generating test cases into an optimization problem, where tests are iteratively evolved by effective meta-heuristic search algorithms. To achieve this objective, the search algorithms are guided by fitness functions that estimate how close tests are to achieving test objectives such as branch coverage. Although SBST has been shown to be successful and effective in practice in many different domains [11, 14, 20, 28, 30, 33, 35, 41, 53, 58], in particular unit testing [26, 45], there is evidence that SBST does not achieve optimal coverage [7, 25], which negatively impacts the fault detection potential of generated test suites [47].

A primary challenge in SBST is defining an effective fitness function. A widely used and effective fitness function employed by many well-known SBST tools is based on branch distance [40]. A branching node $n$ in a program can be regarded as an operator comparing two operands, $op_1$ and $op_2$. Given a test case $t$ exercising one branch of $n$, the branch distance from $t$ (covering) the other branch of $n$ is defined by the value difference of $op_1$ and $op_2$. For example, given the example program in Listing 1, the branch distance from a test case $t \rightarrow a=-3$, $b=0$ to the true branch of the branching

Listing 1: Example method containing for a regular flag problem and an interprocedural flag problem.

```java
public int example(int a, int b)
{
    int x = a + b;
    if(Math.abs(x)==123){...}
    boolean y = a > b && a < b + 10;
    if(y==true){...} //flag problem
    boolean z = f(a, b);
    if(z==true){...} //interprocedural flag problem
}
```

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.

© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8008-9/20/07... $15.00
https://doi.org/10.1145/3395363.3397358
A flag problem is present when the branching condition reads \( z \) on 807 non-trivial Java methods with interprocedural flag problem, sites or iterations of loops, and recursive as it handles cascading flag context-sensitive as it distinguishes method calls from different call required flag. Then, we aggregate these fitness values into a con- gram paths so that we can dynamically calculate a fitness indicating function which may return the flag. Then, we instrument these pro- branch suffering from the interprocedural flag problem, we stat- rather than transformation. Once we detect an uncovered program approach addresses the problem only through code instrumentation 

Unlike existing approaches based on testability transformation, our context-sensitive approach to address the interprocedural flag problem. Although attempts have been made at defining transformations to recursively rewrite all boolean types in an object-oriented program [31]. As a result, it is no longer a quantitative measure on how far a test case is from covering certain branch, and thus there is no gradient in the fitness landscape that the search algorithm could use to reach the objective of covering the branch. A commonly proposed approach to address the flag problem is testability transformation [10, 13, 29, 31, 34, 40]. The idea of testability transformation is to transform the intermediate representation of the source code so that (1) the transformed code preserves the semantics and (2) the boolean variables in branching conditions are replaced by predicates reading non-boolean variables. For instance, the condition \( y==true \) at line 5 of Listing 1 is replaced with \( a>b \) after the transformation. Flag removal approaches use different strategies to transform different types of boolean flags [31]. For example, Baresel et al. [10] and Binkley et al. [13] proposed different flag removal approaches to resolve flags assigned in loops.

Existing work on flag removal only focuses on flags contained inside individual procedures. However, if branching conditions are based on method calls with boolean return-types, this problem becomes interprocedural. For example, consider Line 6 in Listing 1: The flag \( z \) is assigned a boolean return value, and the branch distance for the if-condition in line 7 can only either be 0 or 1. The interprocedural flag problem (IPF) is prevalent: In an empirical study on all methods from 150 open source Java projects we found that 11.8 % of them suffer from interprocedural flag problems (see Section 4.2 for more details). There currently is no satisfactory solution to the interprocedural flag problem. Although attempts have been made at defining transformations to recursively rewrite all boolean types in an object-oriented program [34], correct transformation rules that preserve semantics turn out to be too intricate to be practical. Consequently, the interprocedural flags remain a problem in SBST.

In this paper, we propose a lightweight recursive and context-sensitive approach to address the interprocedural flag problem. Unlike existing approaches based on testability transformation, our approach addresses the problem only through code instrumentation rather than transformation. Once we detect an uncovered program branch suffering from the interprocedural flag problem, we statically analyze the control flow graph of the function producing the boolean value and identify all potential program paths in the function which may return the flag. Then, we instrument these program paths so that we can dynamically calculate a fitness indicating how far a test case is away from exercising each path towards the required flag. Then, we aggregate these fitness values into a continuous value as the cumulative branch distance. Our approach is context-sensitive as it distinguishes method calls from different call sites or iterations of loops, and recursive as it handles cascading flag method calls in branching conditions. We conduct our experiment on 807 non-trivial Java methods with interprocedural flag problem, sampled from 150 open source Java projects. The experiment results demonstrate that our approach outperforms state-of-the-art approaches (11.8% vs. 47.9% coverage within same time budget) with statistical significance and acceptable runtime overheads.

In summary, this paper makes the following contributions:

- We propose a lightweight context-sensitive and recursive approach to address the interprocedural flag problem.
- We present an implementation of our technique based on EvoSuite [21], and the binaries and source code are made available at [1].
- We empirically show the prevalence of interprocedural flag problem in 150 open source Java projects. To the best of our knowledge, we are the first to show how common the interprocedural flag problem is in large-scale open source projects.
- We conduct an experiment on 807 methods suffering interprocedural flag problems sampled from 150 Java projects, showing the effectiveness of our approach.

The rest of the paper is structured as follows. Section 2 presents a motivating example. Section 3 describes our approach in detail. Section 4 evaluates the effectiveness of our approach. Section 5 reviews related work. Section 6 discusses related issues and concludes the paper.

2 MOTIVATING EXAMPLE

In this section, we motivate our approach using the example shown in Listing 2, which is a simplified version of the OpenIntToField-HashMap.get() method in the Apache Commons Math project [2].

The example function takes two input parameters, keys and value, and returns the corresponding key from the array keys if keys contains the computed key of value. It first checks the
Theoretically, the sampling space should range from $2^{-31}$ to $2^{31} - 1$. Nevertheless, existing tools such as EvoSuite usually sample with bias towards a smaller range such as $[-1024, 1024]$.

1
2  boolean checkValue(int[] keys, int index, int val) {
3  int key = hashKey(val);
4  if((keys[index] == 0 || val >= Math.pow(2, index) + 100)
5     && keys[index] == key)
6     return true;
7  return false;
8 }
9
10  int hashKey(int key) {
11    final int h = key * ((key >>> 28) * (key >>> 12));
12    return h * (h >>> 7) * (h >>> 4) + 100;
13  }

Listing 4: Key comparison code, which is called by the target method and produces a boolean flag.

Figure 1: Program Dependency Graph for the example method in Listing 2.
The goal of our approach is to recover the missing quantitative value returned by $m$. Our approach takes as input a target method and a test case. Figure 2 shows an overview of the approach. For example, exercising branch (5, 6) requires that checkValue() returns true (i.e., exercising line 6 in Listing 4). However, a test case may exercise different paths during different iterations of the loop. For instance, the test case $t \leftarrow keys=[0, 1], value=0$ calls checkValue() twice. The first one exercises path (3, 5, 7) and the second one exercises path (3, 4, 7). How do we aggregate the branch distances of different iterations?

Note that the above problems may co-occur and happen in a cascading way. For example, there may be a chain of boolean-typed method calls; each called method may be invoked in a loop; and each method may have multiple paths to return a boolean value. In this work, we propose a recursive and context-sensitive approach to address the above problems systematically.

3 APPROACH

3.1 Definitions

We first define the terminology. We call the method under test the target method, denoted as $m_t$. We refer to a method $m$ as called method if $m$ is called directly in the target method $m_t$ or indirectly through a called method. A called method $m$ is called a flag method if there exists a condition $c$ of a branching node in the target method $m_t$ which directly or indirectly depends on the value returned by $m$. In addition, we call that boolean value used in the condition $c$ an interprocedural flag and the branch to be covered an interprocedural flag branch. For example, method checkPrecondition() in Listing 2 is a flag method of the target method $m_t$. Given a test case $t \leftarrow keys=[0, 1], value=0$, there are two uncovered interprocedural flag branches in get() method, i.e., branch (2, 3) and branch (5, 6). The corresponding interprocedural flags are the values returned by method checkPrecondition() and by method checkValue().

Furthermore, given an interprocedural flag branch $b$ which depends on a flag method $m$, there may be multiple program paths in $m$ which return a boolean value such that $b$ is covered. We call such program paths in $m$ as the required paths for covering $b$. We call the last branch in a required path as the required branch. For example, for branch (2, 3) in Listing 2, the path (2, 3, 4) in Listing 3 is a required path, and the branch (3, 4) is a required branch.

3.2 Overview

The goal of our approach is to recover the missing quantitative measurement on how far a test case is from covering a branch due to the interprocedural flag problem. Our rationale is that, given an interprocedural flag branch, we collect and aggregate the branch distances of the required paths in the called flag methods so that we can have one quantitative measurement. Figure 2 shows an overview of our approach, where each ellipse represents a process and each rounded rectangle represents an artifact. Grey rectangles represent input and output while grey processes represent our key contributions. Our approach takes as input a target method and a test case (which can be generated automatically) and generates quantitative branch distance for evaluating how far the test case is from every interprocedural flag. Given the target method, we first construct its program dependency graph (e.g., the one in Fig. 1) so that we identify all the called methods and their control flow graphs (Step 1). Next, we instrument the target method and its called methods in order to obtain the required branch distances at runtime. Note that the instrumentation is sensitive to call context and iteration context (Step 2). Then, we run the test case for the target method and collect branch distances based on the required paths for every interprocedural flag (Step 3). Finally, we recursively analyze the CFG of the relevant called methods to aggregate the branch distances based on the required paths to a branch distance for every interprocedural flag in the target method (Step 4).

As mentioned in Section 2, our approach must address three challenges, i.e., the multiple path problem, the cascading interprocedural flag problem, and the call and iteration context problem. Next, we introduce (1) how we distinguish different call/iteration context, (2) how we aggregate branch distances in called flag methods, and (3) the overall fitness calculation algorithm.

3.3 Context Sensitive Branch Distance

In the following, we first introduce how branch distance is computed at runtime in classical approaches with context-insensitive branch distance, and how our approach addresses their limitations.

Branch Distance. Suppose that the target method $m_t$ has a branching node $n$. Assume that a test case $t$ of $m_t$ exercises $n$ precisely once. The branch distance of a branch $b$ of $n$ with regard to $t$ is written as $distance(b, t)$ where the condition in $b$ is to be satisfied in order to exercise branch $b$. Let us denote $b.c$ as the condition in $b$, function $distance(b, t)$ is defined as follows [40]:

Figure 2: Approach Overview
Table 1: Sensitive Branch Distances for Iterations

<table>
<thead>
<tr>
<th>Iter</th>
<th>Passed Conditions</th>
<th>App Level</th>
<th>Branch Distance</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;3, 5&gt;: key[index]==0 &amp;&amp; &lt;5, 7&gt;: !(keys[index]==key)</td>
<td>0</td>
<td>100/101=0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>&lt;3, 4&gt;: !(key[index]==0) &amp;&amp; &lt;4, 7&gt;: !(val==Math.pow(2, index))</td>
<td>1</td>
<td>1/2=0.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

the branch distance of a branch b as follows:

\[
distance(b, t) = \min\{distance(b, t)_i\}.
\] (2)

Note that, \(i \in [1, m]\) in Equation 2. With Equation 2, classical search algorithm records the minimum branch distance of a branch in the execution of t. While efficient in general, the information loss (by keeping only the minimum branch distance) incurs two limitations, which impairs the analysis of interprocedural flag problem.

**Limitation 1: Aggregation Confusion.** When an interprocedural flag problem happens (i.e., the classical algorithm is trapped to exercise an interprocedural flag branch), our rationale is to aggregate the branch distances in the called flag method to recover the gradient on the flag branch. However, if the flag method is called in different call sites of the target method, there could be more than two branch distances collected for a single branch, thus taking the minimum branch distance as in Equation 2 will convey misleading branch distance for at least one flag branch. Even worse, it may cause contradictory analysis result. For example, in Listing 5, if we are to cover the branches in line 2 (i.e., isFlag(x)) and line 3 (i.e., isFlag(Math.pow(2, x))), we should analyze the branch distance for the branch in line 8 (i.e., x<1). Intuitively, the gradient of branch distance in this case can be recovered by simply using the branch distance in line 8. However, a test case \(t \leftarrow x=8\) causes a problem of confusing distance. The condition of first interprocedural flag branch (line 2) is evaluated to be true while that of the second is evaluated to be false. When we are to aggregate the branch distance in the \(t\)true branch in line 8 (i.e., \(x < 1\)) back to the second interprocedural flag, Equation 2 shows that the branch distance is 0. The reason is that the branch is line 8 has been exercised, thus the minimum distance is 0. The result is contradictory to the fact that the second interprocedural flag has not yet been evaluated to true on \(t \leftarrow x=0\).

**Limitation 2: Local Optima Effect.** Even if the call site of a flag method is unique in target method, taking the minimum branch distance may cause the search process to be stuck in a local optimum. When the flag method is called within a loop, each loop iteration derives a branch distance for the required branch in the called flag method. Evolving the test case with regard to only minimum branch distance pays a price of losing other optimizing opportunities. Taking example of the interprocedural flag branch \(b = (5, 6)\) in Listing 4, the test case \(t \leftarrow keys=[0, 1], value=0\) for get() method call the checkValue() method twice (i.e., 2 iterations). For readability, we show the call graph in Figure 3, we show the target branch in get() method and the required branch in checkValue() in red. The first iteration exercises the branches (3, 5) and (5, 7) in Listing 4 (yellow
path in Figure 3), and the branch distance for $b$ is 0.99. The second iteration exercises the branches (3, 4) and (4, 7) in Listing 4 (blue path in Figure 3). Table 1 shows the branching condition evaluated in each iteration and the value of branch distances. Note that, there are two branch distances for the uncovered branch (5, 6), as the path (3, 4, 7) can be diverted from either node 3 or node 4. Their branch distances are 1.5 and 1.99 respectively.

According to Equation 2, state-of-the-art approach uses the branch distance in the first iteration (i.e., 0.99) as the branch distance for branch (5, 6). Intuitively, it favors diverting from yellow path over blue path to the target (red) branch. However, in this case, diverting from blue path is more feasible than from yellow path. The condition $\text{keys}[\text{index}]==\text{key}$ (on branch (5, 6)) is extremely hard to satisfy when $\text{keys}[\text{index}]==\emptyset$ (on branch (3, 5)) as the hash result key is usually a non-zero value (line 2 in Listing 4). In contrast, the path condition $\text{value}>=\text{Math}.\text{pow}(2, \text{index}) \land \text{keys}[\text{index}]==\text{key}$ (on branches (4, 5), (5, 6)) is much easier to satisfy. For example, we can repetitively mutate the value of $\text{value}$ and $\text{keys}[\text{index}]$. Nevertheless, given that we keep $\text{distance}((5, 6), t_1)$ as the branch distance, the branch distance is insensitive to changes of $\text{value}$ and $\text{keys}[\text{index}]$. In this regard, the search algorithm will soon be stuck in local optima, leaving the fitness guidance ineffective.

**Solution.** Our remedy is to maintain the branch distance for a branch each time the branch is executed under different contexts and aggregate the branch distances (see details in Section 3.5). Figure 4 shows the meta-model for our instrumentation for distinguishing branch distances under various contexts. Each interprocedural flag branch has multiple required branches (see definition in Section 3.1). Each required branch has multiple iteration contexts, each of which is associated with a branch distance. An iteration context consists of a call context and the runtime execution trace towards the required branch. A call context is a call stack, each of its element describes what method is called and the position (or specific instruction) to trigger the call. Moreover, we keep the runtime execution trace towards the required branch in the top call in the call stack so that we can further distinguish branch distance for various iterations by the execution trace in the call frame.

For example, given the test case $t = \langle \text{key}=[0, 1], \text{value}=? \rangle$, we obtain a branch distance for the branch (4, 5) for each iteration. Both distances share the same call context, i.e., $\text{checkValue()} \sim \text{get()}$. In addition, we distinguish their iteration contexts as the first iteration has branch trace in $\text{get()}$ as (2, 4, 5) while the second iteration has branch trace as (2, 4, 5, 4, 5).

### 3.4 Graph Walking Algorithm

Given a test case and an uncovered interprocedural flag branch, we calculate its interprocedural branch distance by statically walking through the program dependency graph, starting from the interprocedural flag branch and ending by every of its required branch. For each required branch $b$, we compare the static walking trajectory (on program dependency graph) reaching $b$ and all its iteration contexts to select valid branch distances. Then, we recursively apply two-stage aggregation to derive an interprocedural branch distance. In general, we first aggregate the branch distance for each required branch. Then we aggregate the interprocedural branch distance from aggregated distance of multiple required branches. The process is recursive because the required branch may still be an interprocedural flag branch. By this means, our approach first identifies the relevant branches (and their branch distance) by walking towards each required branch, and aggregate the branch distances by walking back towards the target interprocedural branch. Algorithms 1 and 2 show the details.

**Overall Computation.** Algorithm 1 takes as input an interprocedural flag branch $b_f$, its static walking call context $wcc$, and a table of context-sensitive branch distances collected by running a test case, $rcr$ (see Figure 4). It outputs an aggregated branch distance. The walking call context $wcc$ is initialized as the target method. It grows each time we proceed to analyze a new called interprocedural flag method. The runtime context table $rcr$ maintains the context-sensitive branch distance for all the branches during the execution of test case.

When analyzing an uncovered interprocedural flag branch, we first parse the CFG of the flag method and identify all required branches (line 1). Then, we retrieve all the runtime execution traces towards those required branches under such a call context (line 2). As showed in Figure 4, a runtime execution trace consists of a sequence of branches in the CFG of the flag method. Technically, we maintain a map, mapping from a trace towards a required branch to a branch distance. Then, for each required flag branch, we compute its aggregated distance from all its execution traces (line 6–9). We will explain its details (i.e., $\text{branch}_f \text{fitness}()$ method) in Section 3.5. Afterwards, we can further compute an overall branch distance.
from aggregated branch distance of all the required branches (Line 4–11). We will elaborate the aggregation details (e.g., Equation 3) in Section 3.5. Finally, we return the normalized fitness value.

For example, if we are to compute the interprocedural flag branch distance for branch (5, 6) in get() method in Listing 2, with the test case \( t = \text{keys} = [0, 1] \), \( \text{value} = 0 \), Algorithm 1 first identifies that the required branch is branch (5, 6) in checkValue() method. In addition, there are two iterations to call the checkValue() method. The corresponding branch traces are \( (2, 4, 5) \) and \( (2, 4, 5, 4, 5) \) in get() method respectively. Under the call context of \( \text{get}() \) : \( 5^2 \), for each required branch we maintain a map from branch trace to branch distance. For example, for the required branch \( (3, 6) \) in checkValue() method, we maintain a map such as \( \{(2, 4, 5) = 0.99, (2, 4, 5, 4, 5) = 1.99\} \). Then, we aggregate those branch distance accordingly. The details are as follows.

Computation for Required Branch. Algorithm 2 shows how we calculate branch distance for individual required flag branch. It handles various scenarios for calculating the branch distance, including when to aggregate the branch distance, and when to recursively call Algorithm 1 (i.e., \( \text{fitness}_{r}() \) in line 3 and 6 in Algorithm 2).

In general, Algorithm 2 handles three scenarios: (1) when the required branch is exercised (e.g., \( \text{isFlag}(\text{x}) \rightarrow \text{return isFlag2}(\text{x}) \)); in this case, we return the branch distance from reversing result of \( \text{isFlag2}() \); (2) when the required branch is, again, an interprocedural flag branch \( b_r \) but not exercised (line 6), in this case, we return the branch distance from generating the same flag with \( b_r \); and (3) when the required branch has some non-flag branch distance (line 9). For the first and second scenarios, we update the call context and recursively call Algorithm 1. For the third scenario, we compute the branch distance from runtime context table as classical approach and recover the gradient.

### 3.5 Aggregating Branch Distance

We design the aggregation of multiple branch distance based on the rationale that we still favor the smallest distance, but the aggregated branch distance should also be sensitive to changes of other branch distance. With that property, we can avoid the second limitation of Equation 2, i.e., stuck in local optima.

Let \( t \) be a test case and \( b \) be a branch with \( k \) branch distance. Let the \( i \)-th branch distance be \( \text{distance}(b, t) \) \( (i > 0) \), we aggregate the branch distance as follows:

\[
\text{distance}(b, t) = \sum_{i=1}^{k} \frac{1}{\text{distance}(b, t)_{i}}
\]

Equation 3 has the following properties:

- \( \text{distance}(b, t) \) is equal to 0 if \( \exists j \in [1, k] \) so that \( \text{distance}(b, t)_{j} \) is 0.
- \( \text{distance}(b, t) \sim 0 \) if \( \exists j \in [1, k] \) so that \( \text{distance}(b, t)_{j} \sim 0 \).

The notation \( a \sim b \) stands for \( a \) is approaching the value of \( b \). For the second property, we have that:

\[
\frac{\partial \text{distance}(b, t)}{\partial \text{distance}(b, t)_{i}} = k \cdot \left( \sum_{i=1}^{k} \frac{1}{\text{distance}(b, t)_{i}} \right)^{-2}.
\]

It means that, once \( \exists i \) so that \( \text{distance}(b, t)_{i} \) is the smallest distance, the derivative of \( \text{distance}(b, t)_{i} \) over \( \text{distance}(b, t)_{j} \) decreases fastest so that we can favor it to decrease to 0. By this means, we achieve the goal of favoring the smallest branch distance.

On the other hand, all the other branch distances are considered so that we can avoid being stuck in a local optima. For example, for branch (5, 6) in Listing 4 (or Rectangle checkValue() in Fig. 1), the test case \( t = \text{keys} = [0, 1] \), \( \text{value} = 0 \) has \( \text{distance}(5, 6)_{1} = 0.99 \) while \( \text{distance}(5, 6)_{2} = 1.5 \). Therefore, when \( \text{distance}(5, 6)_{1} \) stuck on local optima, the fitness is still sensitive to \( \text{distance}(5, 6)_{2} \) and gradually evolves to a test case covering the target branch.

Last but not least, the numerator \( k \) is used to avoid test cases incurring a large number of iterations. Note that, if we let numerator be 1, the more loop iterations a test case incurs, the larger the
denominator. As a result, the search algorithm will favor those test cases incurring large number iterations. This phenomenon causes two drawbacks: (1) the number of iterations is usually irrelevant to exercise an uncovered branch; and (2) the test cases incurring many iterations have larger runtime overhead for the search process. In this regard, we add $k$ as a penalizing factor to mitigate the problem.

4 EVALUATION

We build our proof-of-concept tool Evoif (EvoSuite on Interprocedural Flag) based on EvoSuite [21]. In the implementation of Evoif, we enhance EvoSuite’s instrumentation to support our meta model in Figure 4 and extend a new test case fitness function $f$branch based on EvoSuite framework. The tool and its source code, as well as all the experimental data, are accessible at [1]. In this section, we aim to answer the following research questions:

- **RQ1:** How prevalent is the interprocedural flag problem in practice?
- **RQ2:** Does Evoif improve the testing performance on code suffering from interprocedural flag problems compared to the state-of-the-art approach?
- **RQ3:** Does Evoif incur acceptable overhead when code under test does not suffer from interprocedural flag problems?

4.1 Experiment Setup

4.1.1 Benchmark Tool. We choose EvoSuite [21] as our benchmark tool. EvoSuite supports various testability transformations, including transforming common methods in the Java String and Collection classes. For example, EvoSuite can transform the String equals() method into one returning an int (based on the edit distance) to address the flag problem. It also supports a number of useful heuristics to cover challenging branches [21, 23].

4.1.2 Subject Methods. For this experiment, we use 807 methods suffering from interprocedural flag methods from 150 projects. The 150 projects consist of standard SF100 data set [22] and 50 extra complemented projects (we will discuss why we complement 50 projects later). Our selection (or method filtering) heuristics are fixed and the selection of those methods is replicable in our experiment. The heuristics are designed for EvoSuite’s limitation of class instrumentality and mutation capability. As for class instrumentality, current EvoSuite implementation cannot allow us to instrument JDK library classes. As a result, we miss the branch distance in some called flag method such as the equals() method in java.lang.Object. As for mutation capability, EvoSuite has limitation to instantiate legitimate complex parameter object (including abstract class and interface), e.g., java.sql.ResultSet. As a result, the EvoSuite framework keeps generating test cases triggering program crashes before executing the target methods during the evolution. In both cases, we cannot evaluate the performance difference between EvoSuite and Evoif from the perspective of code implementation.

Filtering Heuristics. Addressing the mutation capability is beyond the scope of this work. Therefore, to evaluate the effectiveness of our gradient recovering approach, we conservatively define the filtering rules (in Table 2) to select methods which are less likely to be affected by the limitation of mutation capability. In Table 2, we also list the quantity of methods filtered by every filtering heuristic. We will thoroughly discuss the filtering heuristics in details in Section 4.5 and Section 4.6.

**Project Complement.** The heuristics in Table 2 cause the number of usable IPF methods from the traditional SF100 data set to be small. On one hand, some project in SF100, such as greencow (the 28th project in SF100), contains only one class with two branchless methods. In addition, 56% of the Java classes in SF100 benchmark contain only branchless methods [44]. On the other hand, our heuristics to get grid of complicated data structure remove a lot of methods (we will discuss it in Section 4.5 and 4.6). Therefore, following the convention in software testing community [44], we complemented 50 large popular Java open source projects to scale up our experiment. These projects are chosen considering their scale and popularity (e.g., Weka (v3.8.0) [5], JFreeChart (v1.5.0) [4], etc.). Some have been used in the annual unit test generation contest [46]. We deem them to be representative and realistic.

4.1.3 Performance Evaluation. In this experiment, both Evoif and EvoSuite stop when the 100s time budget is used up or they have achieved 100% branch coverage. Given that the used time and coverage of EvoSuite and Evoif on each target method is evaluated for 10 times, we compare the mean and median coverage and time respectively. We classify the performance comparison on a target method into 5 categories.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Description</th>
<th>#IPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Called IPF is not instrumentable.</td>
<td>1673</td>
</tr>
<tr>
<td>H2</td>
<td>Target method with no primitive parameters.</td>
<td>4573</td>
</tr>
<tr>
<td>H3</td>
<td>Target method with parameter of interface or abstract class type.</td>
<td>1468</td>
</tr>
<tr>
<td>H4</td>
<td>Called IPF method with no primitive parameters.</td>
<td>131</td>
</tr>
<tr>
<td>H5</td>
<td>Called IPF method with parameters of interface or abstract class type.</td>
<td>654</td>
</tr>
<tr>
<td>Used IPF Methods</td>
<td>The methods used for the experiment.</td>
<td>807</td>
</tr>
<tr>
<td>Total</td>
<td>/</td>
<td>9306</td>
</tr>
</tbody>
</table>

Also listed in the table is the quantity of methods filtered by every filtering heuristic. We will thoroughly discuss the filtering heuristics in details in Section 4.5 and Section 4.6.
We observe that there are 78973 individual methods with program
variants provided by EvoSuite. More specifically, 255 (i.e., 31.6%) v.s. 50 (i.e., 6.2%), (2) the used
categories. Overall, 255 methods fall to “better coverage”, 57 fall to
the worse coverage. We can see that (1) the number of methods suffering from interprocedural flag problems.
and 4s v.s. 5s on median), and (3) there are a noticeable number of
traits of both approaches equal performance (more specifically, 51.3%). In terms of average coverage, our approach achieves 51.1% coverage comparing to 47.9% of EvoSuite. The Mann-Whitney u test on coverage shows that the two-tailed significance
value $p < 0.0001$, indicating that our improvement on the coverage is statistically significant.

Qualitative Analysis. The results show that Evoif outperforms EvoSuite in general improves the performance on 312 methods, a.k.a, 38.7% of the total methods) as the algorithm can search towards a valid test case with recovered gradients. Nevertheless, there are still a number of methods where Evoif does not improve or even “underperforms” EvoSuite. We empirically investigated the details and qualitatively analyzed the worse and equal coverage.

Equal Coverage and Time. In this experiment that, despite that 414 (i.e., 51.3%) of the target methods fall into equal category, we observe that, in equal category, both Evoif and EvoSuite achieve 100% coverage with less than or equal to 5s on 63 methods, and more importantly, both approaches achieve 0% coverage (and use up the time budget) on 193 methods. The former shows that the interprocedural flag problem is sometimes trivial and can be covered with random guess (see example in Listing 6, as long as the eps variable is set to a large value, the condition of interprocedural flag on line 3 is easy to satisfy). The latter case (i.e., 0% coverage) lies in that EvoSuite framework has trouble on initializing the constructor of the class declaring the target method. Such an implementation limitation largely impairs the effectiveness of Evoif.

Worse Coverage. We investigate the target methods where Evoif underperforms EvoSuite. It happens usually when the interprocedural flag branch $b_i$ depends on non-interprocedural branch $b_n$. Note that, Evoif may have a larger runtime overhead than EvoSuite as our approach requires more instrumentation to maintain the context information to make sure the branch distance is context-sensitive. Therefore, in general, Evoif needs to spend more time to cover “normal” branch before Evoif gives a chance to “break through” the tough interprocedural flag branch. Listing 7 shows an example from IDA SDK project [3], we need to break through the branch on line 3 before the approach of Evoif takes effect on the branch on line 4. Note that, EvoSuite applies rule-based testability transformation on this case so that String.equals() method return continuous branch distance and EvoSuite can cover it within a number of trials. Nevertheless, such a heuristic testability transformation rule cannot work on the branch on line 4. Noteworthy, during our investigation for this example, Evoif will have the same coverage performance with EvoSuite with 200s time budget and outperforms EvoSuite with a time budget of 250s.
Worse Time. Trivial interprocedural flag branch is the major reason for Evoif to achieve its coverage with worse time. Of the 37 target methods where Evoif underperforms EvoSuite in time, both approaches achieve 100% coverage within 10s on 30 of them. It means that those flags are trivial and can be covered with a few seconds.

4.4 RQ3: Runtime Overhead

During this experiment, within the budget of 100s, Evoif evolves on average 498.9 iterations. In contrast, EvoSuite evolves on average 680.9 iterations. A more detailed runtime overhead distribution can be referred in [1]. As discussed above, the performance overhead of Evoif is caused by building the context-sensitive branch distance. In general, the trade-off between runtime overhead and gradient recovering is paid off as Evoif achieves higher coverage even with less number of evolving iterations.

In summary, we conclude that (1) Evoif outperforms EvoSuite in general (EvoSuite achieves higher average coverage and the improvement is statistically significant), (2) Evoif may not improve or even underperforms EvoSuite when the interprocedural is trivial or some non-interprocedural flag branch guards the interprocedural flag branch, and (3) the gradient recovering technique incurs acceptable overhead (less evolving iterations but higher coverage) for interprocedural flag problem.

4.5 Discussion

4.5.1 Applicability. Our experiment empirically shows that testing can be an optimization problem in theory, but code synthesizing problem in practice. The success of applying a search-based software testing in practice involves the careful design of multiple components, for example a well-designed fitness to gauge a “fitter” test case, a comprehensive mutation capability on generated test cases, an efficient optimization algorithm (like DynaMOSA), etc. In this work, we target a finer design of fitness, which presents its effectiveness within the reach of EvoSuite’s mutation capability. More specifically, the fitness can only work if EvoSuite can provide enough search space to explore. Our approach works well on more target methods with primitive parameters (including String) while is nullified when the target method requires deep analysis on complex data structure and polymorphism. We deem that addressing an orthogonal problem such as improving mutation capability has beyond the scope of this work. Nevertheless, we are targeting to improve mutation capability in our future work.

4.5.2 No Free Lunch. From an information perspective, it is not free for us to recover the gradient for branch distance for interprocedural flag problem. Our context-sensitive approach essentially trades some additional runtime overhead for the gradient information. The experiment shows that such trade-off is paid-off in general. Nevertheless, the price may sometimes be paid in vain, which leads to worse coverage. It is the common problem for all online learning technique used in software testing or fuzzing approach. A more practical and economical way can be a hybrid testing approach combining Evoif and EvoSuite. Intuitively, we can first apply lightweight EvoSuite with a short time budget to cover those trivial branches. Then, for the remaining uncovered “tough” branches, we can then apply Evoif. We will explore the optimal way to switch between different testing strategies in our future work.

4.6 Threats to Validity

Threats to external validity arise from our selection of benchmarks. We conducted our experiment on the 150 open source Java projects. Although the number of projects is large, the selected projects may not be representative, and further experiments with more methods are needed to generalize our results. Threats to internal validity arise from how the experiments were carried out. We ran Evoif and EvoSuite for only 10 times on each method, while the effect of randomness for test-generation may require a larger number of runs to offset. However, to address this threat, we applied both tools in large number of Java methods. In addition, the time budget for each comparison is only 100s, which means that some “equal” comparisons may not be equal if we run both approaches for a longer time. Nevertheless, the same problem may arise under any time budget. In the future, we will run experiments with longer time budget and under more search algorithms for more generalized result. Threats to construct validity come from what measure we chose to evaluate the success of our techniques. We compared the performance of Evoif with EvoSuite in terms of coverage on individual methods, but did not quantify the effects on entire classes. In the future, we will extend our experiments regarding the above threats to generalize our results.

5 RELATED WORK

5.1 Search Based Software Testing

Search based Software Testing (SBST) regards software testing as an optimization problem [28]. Miller et al. [41] pioneered the work for generating test cases with parameters of float types. Following their work, SBST is used for a range of software testing problems, including functional testing [14], regression testing [35], mutation testing [33], as well as test case prioritization [49]. The majority of existing works leverage meta-heuristic search algorithms [12] for generating test cases to cover challenging test goals. Given such a framework, researchers aim to cover various test goals (e.g., branch coverage, path coverage, use-def coverage, etc.) with different search strategies (genetic algorithm, hill-climbing algorithm, etc) [6, 15, 24, 42, 48], fitting test representations [11, 15, 30, 53], and the fitness metrics [11, 30, 45, 53, 58]. Readers are referred to survey papers [32, 40] for more details. Following their work, Aleti et al. [7] investigated the fitness landscape characterisation and showed that the most problematic landscape feature is the presence of many plateaus. The interprocedural flag problem is one of the reasons causing such many-plateaux landscape, where the absence of gradients in the search landscape makes any search-based approach

```java
1. boolean getMethod(String p1, String p2){
2.     ...
3.     if (p1.equals(p2)) {
4.         return false;
5.     }
6. }
7. 
```

Listing 7: Normal Parent Branch
degenerate to a simple random testing approach. Our work aims at recovering the discovered gradients, which can facilitate and improve many different search-based software testing approaches.

5.2 Testability Transformation

Testability transformation [10, 13, 29, 31, 34, 46] is one of the major techniques for addressing flag problems. Researchers use testability transformation to remove flags by replacing boolean condition with non-boolean condition. Harman et al. [31] classified the flag problem into 5 levels based on the complication to make testability transformation. Barasel et al. [10] and Binkley et al. [13] proposed different flag removal approaches to address the level-5 flag problem, i.e., transforming the flag assigned in the loop. However, these papers only focus on the flag problem within procedures, leaving the interprocedural flag problem unaddressed.

One most relevant work comparing to our approach is Li et al.’s method transforming approach [34], which recursively transforms all boolean method calls into ones with non-boolean return types. However, their rule-based approach is limited in practice as the number of necessary transforming rules is huge and keeping the rules practical and consistent is complicated, and it is hard to prove that the rules are semantic-preserving for recursive method transformations. In contrast, our approach does not require complicated transformation rules, as we only require instrumenting the called methods and aggregate the branch distance through program analysis. Our experiment shows that, with acceptable runtime overheads, we can achieve good coverage improvement.

6 CONCLUSION AND FUTURE WORK

In this work, we proposed a context-sensitive and recursive approach to address the interprocedural flag problem in search-based software testing. Unlike the traditional testability transformation based approaches, our approach only requires instrumentation to get the branch distances inside the called flag methods, and aggregates those branch distances with regard to specific call contexts and iteration contexts.

In our future work, we will explore hybrid testing strategy combining Evo† and EvoSuite or even other testing technique such as symbolic execution [52], loop summarization techniques [55–57], and formal analysis [9, 16, 17]. Moreover, we will investigate our guided testing techniques on more platforms [38, 39]. Finally, we will explore the testing criteria beyond the coverage for finding more bugs, and integrate testing with various root cause analysis [18, 19, 27, 36, 37, 50, 51, 54].

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and suggestions. This research has been partially supported by the following grants. The National Research Foundation, Prime Ministers Office, Singapore, under its Corporate Laboratory@University Scheme, National University of Singapore and under its National Cybersecurity R&D Program, Singapore Telecommunications Ltd., the National Research Foundation Singapore under its NSOE Programme (NSOE-TSS2019-03 and NSOE-TSS2019-05), the EPSRC project EP/N023978/2, and the National Science Foundation of China (No. 61632015, U1766215, 61833015).

REFERENCES
