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

Yun Lin
National University of Singapore
Singapore
dcsliny@nus.edu.sg

Jun Sun
Singapore Management University
Singapore
junsun@smu.edu.sg

Gordon Fraser
University of Passau
Germany
gordon.fraser@uni-passau.de

Ziheng Xiu
National University of Singapore
Singapore
e0140856@u.nus.edu

Ting Liu
Xi'an Jiaotong University
China
tingliu@mail.xjtu.edu.cn

Jin Song Dong
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

ABSTRACT

In Search-based Software Testing (SBST), test generation is guided by fitness functions that estimate how close a test case is to reach an uncovered test goal (e.g., branch). A popular fitness function estimates how close conditional statements are to evaluating to true or false, i.e., the *branch distance*. However, when conditions read Boolean variables (e.g., `if(x && y)`), the branch distance provides no gradient for the search, since a Boolean can either be true or false. This *flag problem* can be addressed by transforming individual procedures such that Boolean flags are replaced with numeric comparisons that provide better guidance for the search. Unfortunately, defining a semantics-preserving transformation that is applicable in an *interprocedural* case, where Boolean flags are passed around as parameters and return values, is a daunting task. Thus, it is not yet supported by modern test generators.

This work is based on the insight that fitness gradients can be recovered by using runtime information: Given an uncovered interprocedural flag branch, our approach (1) calculates context-sensitive branch distance for all control flows potentially returning the required flag in the called method, and (2) recursively aggregates these distances into a continuous value. We implemented our approach on top of the EvoSuite framework for Java, and empirically compared it with state-of-the-art testability transformations on 807 non-trivial methods suffering from interprocedural flag problems, sampled from 150 open source Java projects. Our experiment demonstrates that our approach achieves higher coverage on the subject methods with statistical significance and acceptable runtime overheads.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

search-based, testing, testability, program analysis

ACM Reference Format:
Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397358

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397358

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].

```
1  public int example(int a, int b){
2      int x = a + b;
3      if(Math.abs(x)==123){...}
4      boolean y = a > b && a < b + 10;
5      if(y==true){...} //flag problem
6      boolean z = f(a, b)
7      if(z==true){...} //interprocedural flag problem
8  }
```

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

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 to (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 node at

line 3 is $123 - |-3| = 120$, i.e., the corresponding 'fitness' for t with regard to the `true` branch is 120. The fitness function guides search algorithms towards reaching coverage objectives; for example, an optimal test case for the example branch is `a=-100, b=-23` with a branch distance of 0, i.e., it covers the branch.

An important problem with this approach is the *flag problem* [13]: A flag problem is present when the branching condition reads boolean variables, e.g., `y==true` (line 5 in Listing 1). In such a case, the branch distance is either 0 (e.g., when `y=true`) or 1 (e.g., when `y=false`). 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 && a<b+10` 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 approach (51.1% v.s. 47.9% coverage within same time budget) with statistical significance and acceptable runtime overheads.

```
1  Integer get(int[] keys, int value){
2    if(!checkPrecondition(keys, value))
3      return null;
4    for(int i=0; i<keys.length; i++)
5      if(checkValue(keys, i, value))
6        return keys[i];
7    return null;
8  }
```

Listing 2: Target method under test, containing two interprocedural flags.

```
1  boolean checkPrecondition(int[] keys, int value) {
2    if(checkArrayRange(keys)
3         && checkValueRange(value))
4      return true;
5    return false;
6  }
7
8  boolean checkArrayRange(int[] keys) {
9    return keys.length < 10000;
10 }
11
12 boolean checkValueRange(int value) {
13   return Math.abs(value) < 10000;
14 }
```

Listing 3: Precondition checking code, which is called by the target method and produces a boolean flag.

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 `OpenIntToFieldHashMap.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

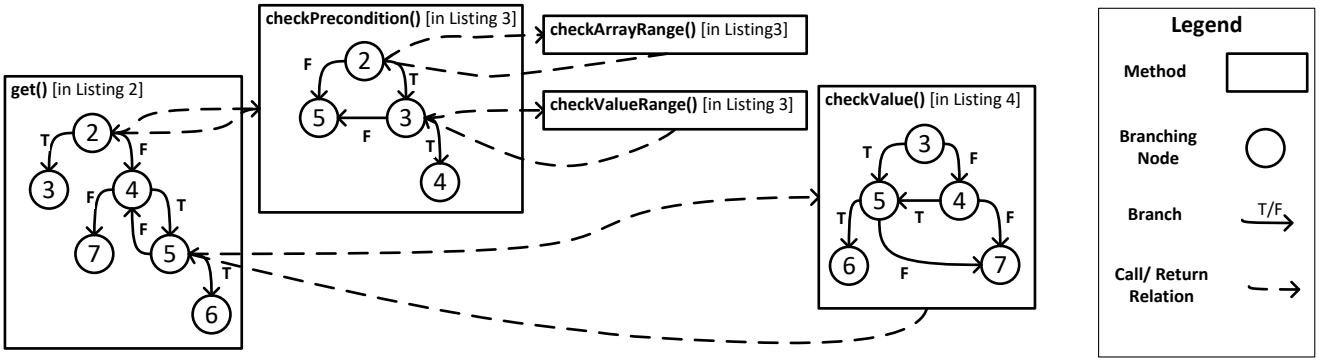

Figure 1: Program Dependency Graph for the example method in Listing 2.

```
1  boolean checkValue(int[] keys, int index, int val) {
2     int key = hashKey(val);
3     if((keys[index] == 0
4         || 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 >>> 20) ^ (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.

legitimacy of the inputs (line 2 in Listing 2) and iteratively goes through each element in `keys` to check whether the corresponding key of `value` is contained in `keys` (line 4–6). It returns null if either the inputs does not satisfy the precondition (line 3) or the key of `value` is not contained in `keys` (line 7). The program contains two method calls in line 2 and 5. The details of the invoked methods are shown in Listing 3 and 4.

For readability, we show the program dependency graph in Figure 1, where each method is represented as a rectangle. Within a rectangle, the control flow graph (CFG) of the method is shown where each node is represented as a circle labelled with its source code line number and the control flows between nodes are represented by lines labelled by branch value (i.e., true or false). Connecting the rectangles, the call relation is represented by dashed lines. For simplicity, we skip the CFG of `checkArrayRange()` and `checkValueRange()`.

Automatically generating test cases for method `get()` to achieve high branch coverage is non-trivial, in particular for the following two branches:

- The branch $\langle 2, 3\rangle$ in line 2 (i.e., the edge $\langle 2, 3\rangle$, Listing 2).
- The branch $\langle 5, 6\rangle$ in line 5 (i.e., the edge $\langle 5, 6\rangle$, Listing 2).

Using random testing, the branch $\langle 2, 3\rangle$ is hard to cover because randomly generated test cases are most likely to violate the precondition (see Listing 3)[1]. The second branch is hard to cover because the mapping rule from a value to its key is complicated (see line 2 in Listing 4) and thus it is hard to randomly generate a pair of "magical" key (in `keys`) and corresponding `value` under the guarding conditions (see the condition in line 5 in Listing 4).

Testing method `get()` using SBST suffers from the interprocedural flag problem. For instance, the state-of-the-art SBST tool *EvoSuite* is unable to generate tests with sufficient branch coverage. This is because *EvoSuite* uses branch distance to quantitatively measure how far a test suite is from the uncovered branches. In this example, a human reader can observe that the test case $t \leftarrow$`keys=[0], value=999` is a "closer" candidate towards covering branch $\langle 2, 3\rangle$ in Listing 2 than $t' \leftarrow$`keys=[0], value=99`. However, existing branch distance (e.g., [40]) is defined based on the operands of the branching nodes (e.g., in line 2, based on the returned boolean values from `checkPrecondition()`). If the operands are boolean variables, the branch distance will always be evaluated to 0 or 1 and thus do not provide a measurement on how close a test suite is to cover certain branch.

The key to address such an interprocedural flag problem is to aggregate the branch distances in the called methods into one combined value to recover the "gradient" towards the uncovered branch. The following three challenges make this a non-trivial problem:

C1: The multiple paths problem. The called method may have multiple program paths, each of which returns a boolean value. For example, for covering the branch $\langle 2, 3\rangle$ in method `get()`, we require a test case to exercise line 5 in Listing 3. However, there are two paths leading to node 5, i.e., path $\langle 2, 5\rangle$ and path $\langle 2, 3, 5\rangle$ in method `checkPrecondition()`. Note, that exercising path $\langle 2, 5\rangle$ requires creating an array of length of 10,000, while exercising path $\langle 2, 3, 5\rangle$ only requires generating a large `value`. Moreover, during test suite optimization, the likelihood of exercising either path sometimes varies from generation to generation (if we use a genetic algorithm). Consequently, how can we take multiple paths into consideration and aggregate their branch distances?

C2: The cascading interprocedural flag problem. The called method may suffer from further interprocedural flag problems. We can observe that methods with boolean return types are invoked at line 2 and line 3 in Listing 3. In general, such interprocedural flag problems may cascade for multiple layers (depending on the call

[1]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].

graph depth) or infinitely many layers (e.g., recursive method calls). How do we aggregate the branch distance then?

C3: The call and iteration context problem. The branch distance of a branching condition of a boolean method call is sensitive to the call site and loop iteration context. For example, exercising branch $\langle 5, 6\rangle$ 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 ←`keys=[0, 1], value=0` calls `checkValue()` twice. The first one exercises path $\langle 3, 5, 7\rangle$ and the second one exercises path $\langle 3, 4, 7\rangle$. 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 a 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 `get()`. Given a test case t ←`keys=[0, 1], value=-20`, there are two uncovered interprocedural flag branches in `get()` method, i.e., branch $\langle 2, 3\rangle$ and branch $\langle 5, 6\rangle$. 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 $\langle 2, 3\rangle$ in Listing 2, the path $\langle 2, 3, 4\rangle$ in Listing 3 is a required path, and the branch $\langle 3, 4\rangle$ 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

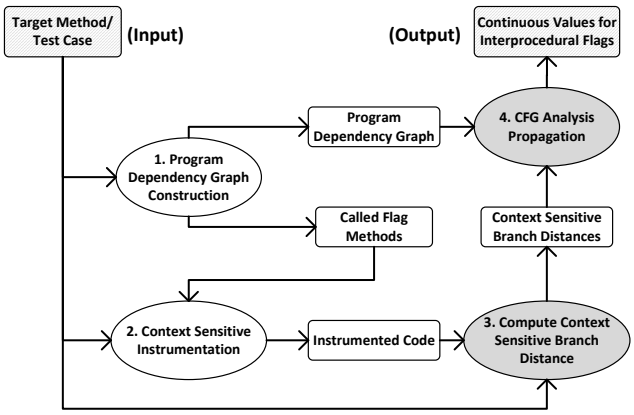

Figure 2: Approach Overview

(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 interprecedural 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 finalize the section with (3) the overall fitness calculation algorithm.

3.3 Context Sensitive Branch Distance

In the following, we first introduce how branch distance are 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]:

```
1  boolean example(int x) {
2     if(isFlag(x)) x++;
3     if(isFlag(Math.pow(2, x)) return true;
4     return false;
5  }
6
7  boolean isFlag(int x){
8     if (x < 1)
9        return true;
10    return false;
11 }
```

Listing 5: Example for Context Sensitive Branch Distance

$$distance(b,t) = \begin{cases} 0 & \text{if } b.c \text{ evaluates to } \texttt{true} \\ K & \text{else if } b.c \text{ is } \texttt{false} \\ |\ a-b\ | + K & \text{else if } b.c \text{ is } \texttt{a==b} \\ K & \text{else if } b.c \text{ is } \texttt{a!=b} \\ a-b+K & \text{else if } b.c \text{ is } \texttt{a<b} \\ a-b+K & \text{else if } b.c \text{ is } \texttt{a<=b} \\ b-a+K & \text{else if } b.c \text{ is } \texttt{a>=b} \\ b-a+K & \text{else if } b.c \text{ is } \texttt{a>b} \end{cases}$$

where K is a positive constant value which is the minimum value of all possible branch distances. In *EvoSuite*, K is set to be 1. Note that the branch distance is always a non-negative value. For instance, given the test case $t' \rightarrow$`a=-20, b=-3`, the branch distance for the branch at line 3 in Listing 1 (i.e., `Math.abs(a+b)==123`) is $||-20-3|-123|+K$=100 + K for the then-branch and 0 for the else-branch.

Normalized Branch Distance. If the test t does not cover the branching node n of the branch b, let n_p be the nearest branching node which is covered by t such that n control-depends on n_p, a normalized branch distance is used to quantify how far t is from covering b, as follows.

$$distance_n(b,t) = approach\_level + norm(distance_{n_p}(b',t)). \quad (1)$$

where *approach_level* measures the number of branching node from node n_p to node n along the program path; *norm* is a function which normalizes the distance to a value between 0 and 1; and $distance_{n_p}(b',t)$ is the branch distance defined above for the branch from n_p to n (i.e., b' in Equation 1). The normalization function may be implemented differently in different SBST approaches. One candidate implementation is $norm(x) = \frac{x}{1+x}$ [8]. For example, the test case $t\rightarrow$`keys=[], b=Integer.MAX_VALUE` exercises branch $\langle 2,3\rangle$ in `get()` method in Fig. 1. Thus, the approach level for branching node 5 in `get()` is 2 and $distance_2(b,t)$ is 1, thus the overall $distance_5(\langle 2,3\rangle, t) = 2 + \frac{1}{1+1} = 2.5$.

Context Insensitive Branch Distance. A test, t, can exercise a branch, b, multiple times under different contexts. This can happen when the branching node of b is either (1) a loop node exercised for multiple iterations or (2) the method defining b is called in multiple sites. Listing 5 shows the example. The branching node at line 8 is executed twice given any test case for method `example()`. Assume that the branching node of a branch, b, is exercised m times during the execution of t. Let us use $distance(b,t)_i$ for denoting the branch distance calculated for the ith time b's branching node is exercised. Existing state-of-the-art SBST tools like *EvoSuite* compute the branch distance of a branch b as follows:

$$distance(b,t) = \min\{distance(b,t)_i\}. \quad (2)$$

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*.

Table 1: Sensitive Branch Distances for Iterations

Iter	Passed Conditions	App Level	Branch Distance	Res
1	<3, 5>: key[index]==0 && <5, 7>: ! (keys[index]==key)	0	100/101=0.99	0.99
2	<3, 4>: !(key[index]==0)	1	1/2=0.5	1.5
	<3, 4>: !(key[index]==0) && <4, 7>: !(val>=Math.pow(2, index)+100)	1	101/102=0.99	1.99

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=0` 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 `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 $b = \langle 5,6\rangle$ 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()`

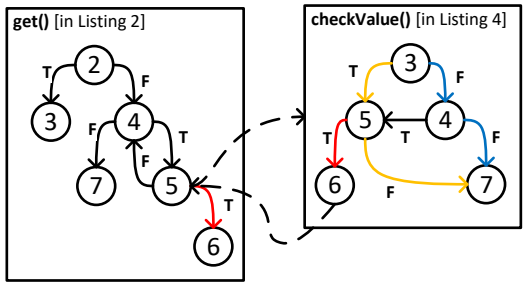

Figure 3: Local Optima Example

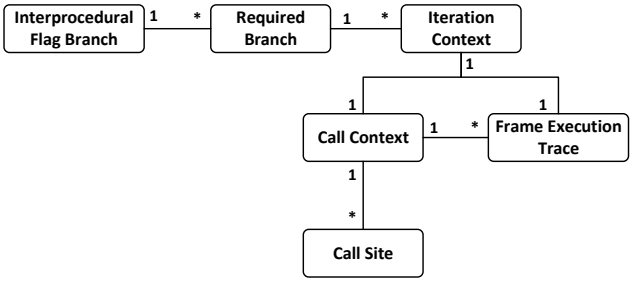

Figure 4: Meta Model for Branch Distance

in red. The first iteration exercises the branches $\langle 3, 5\rangle$ and $\langle 5, 7\rangle$ in Listing 4 (yellow path in Figure 3), and the branch distance for b is 0.99. The second iteration exercises the branches $\langle 3, 4\rangle$ and $\langle 4, 7\rangle$ 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 $\langle 5, 6\rangle$, as the path $\langle 3, 4, 7\rangle$ 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 $\langle 5, 6\rangle$. 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 `keys[index]==key` (on branch $\langle 5, 6\rangle$) is extremely hard to satisfy when `keys[index]==0` (on branch $\langle 3, 5\rangle$) as the hash result `key` is usually a non-zero value (line 2 in Listing 4). In contrast, the path condition `value>=Math.pow(2,index)` $\wedge$ `keys[index]==key` (on branches $\langle 4, 5\rangle$, $\langle 5, 6\rangle$) is much easier to satisfy. For example, we can repetitively mutate the value of `value` and `keys[index]`. Nevertheless, given that we keep $distance(\langle 5, 6\rangle, t)_1$ as the branch distance, the branch distance is insensitive to changes of `value` and `keys[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 distance 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 $t \leftarrow$`keys=[0, 1], value=0`, we obtain a branch distance for the branch $\langle 4, 5\rangle$ for each iteration. Both distances share the same call context, i.e., `checkValue()~get()`. In addition, we distinguish their iteration contexts as the first iteration has branch trace in `get()` as $\langle 2, 4, 5\rangle$ while the second iteration has branch trace as $\langle 2, 4, 5, 4, 5\rangle$.

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. Algorithm 1 and 2 show the details.

Overall Computation. Algorithm 1 takes as input an interprocedural flag branch b_{if}, its static walking call context *wcc*, and a table of context-sensitive branch distances collected by running a test case, *rct* (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 *rct* 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., $branch\_fitness_r()$ method) in Section 3.5.

Algorithm 1: Calculate Fitness for Interprocedural Flag Branch ($branch\_fitness_{if}$)

Input : an interprocedural flag branch b_{if}, walking call context wcc, runtime context table rct
Output: an aggregated branch distance $fitness$

1 $branch\_set_r \leftarrow identify\_required\_branches(b_{if}.called\_cfg)$;
2 $id\_map \leftarrow rct.find(b_{if}, wcc)$; // return a map from iteration to a branch distance.
3 $fitness\_set \leftarrow \emptyset$;
4 **for** $b_r \in branch\_set_r$ **do**
5 $\quad fitness\_set_{iter} \leftarrow \emptyset$;
6 $\quad$ **for** $trace \in id\_map.keys$ **do**
7 $\quad\quad fitness_{iter} \leftarrow branch\_fitness_r(b_r, wcc, trace, rct)$;
8 $\quad\quad fitness\_set_{iter} \leftarrow fitness\_set_{iter} \cup \{fitness_{iter}\}$;
9 $\quad fitness \leftarrow$ aggregate $fitness\_set_{iter}$ using Equation 3;
10 $\quad fitness\_set \leftarrow fitness$;
11 $fitness_{agg} \leftarrow$ aggregate $fitness\_set$ using Equation 3;
12 **return** $normalize(fitness_{agg})$;

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 $\langle 5, 6\rangle$ in `get()` method in Listing 2, with the test case $t \leftarrow$`keys=[0, 1], value=0`, Algorithm 1 first identifies that the required branch is branch $\langle 5, 6\rangle$ in `checkValue()` method. In addition, there are two iterations to call the `checkValue()` method. The corresponding branch traces are $\langle 2, 4, 5\rangle$ and $\langle 2, 4, 5, 4, 5\rangle$ in `get()` method respectively. Under the call context of `get():5` [2], for each required branch we maintain a map from branch trace to branch distance. For example, for the required branch $\langle 3, 6\rangle$ in `checkValue()` method, we maintain a map such as $\langle\langle 2, 4, 5\rangle$=0.99, $\langle 2, 4, 5, 4, 5\rangle$=1.99$\rangle$. 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., $fitness_{if}()$ in line 3 and 6 in Algorithm 2).

In general, Algorithm 2 handles three scenarios: (1) when the required branch is exercised (e.g., `isFlag(int x){return isFlag2(x);}`, in this case, we return the branch distance from reversing result of `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.

[2]Here, we use line number 5 as the call site. Nevertheless, we use the bytecode instruction index in our implementation.

Algorithm 2: Calculate Fitness for Required Branch ($branch\_fitness_r$)

Input : a required flag branch b_r, walking call context wcc, iteration $trace$, runtime context table rct
Output: fitness f

// the branch requires reversing the output of an exercised interprocedural flag method.
1 **if** b_r *is interprocedural flag branch* && b_r *is exercised* **then**
2 $\quad wcc \leftarrow wcc.append(b_r.callsite)$;
3 $\quad$ **return** $branch\_fitness_{if}(\neg b_r, wcc, rct)$;
// the branch requires exercising an interprocedural flag method.
4 **if** b_r *is interprocedural flag* && b_r *is not exercised* **then**
5 $\quad wcc \leftarrow wcc.append(b_r.callsite)$;
6 $\quad fit \leftarrow branch\_fitness_{if}(b_r, wcc, rct)$;
7 **else**
8 $\quad id\_map_r \leftarrow rct.find(b_r, wcc)$;
9 $\quad fit \leftarrow id\_map_r.get(trace)$;
10 **return** fit;

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 ith branch distance be $distance(b, t)_i$ ($i > 0$), we aggregate the branch distance as follows:

$$distance(b, t) = \frac{k}{\sum_{i=1}^{k} \frac{1}{distance(b,t)_i}} \tag{3}$$

Equation 3 has the following properties:

- $distance(b, t)$ is equal to 0 if $\exists i \in [1, k]$ so that $distance(b, t)_i$ is 0.
- $distance(b, t) \rightsquigarrow 0$ if $\exists i \in [1, k]$ so that $distance(b, t)_i \rightsquigarrow 0$.

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

$$\frac{\partial distance(b,t)}{\partial distance(b,t)_i} = k \cdot (\sum_{i=1}^{k} \frac{1}{distance(b,t)_i})^{-2} \cdot distance(b,t)_i^{-2} \tag{4}$$

It means that, once $\exists i$ so that $distance(b, t)_i$ is the smallest distance, the derivative of $distance(b, t)$ over $distance(b, t)_i$ 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 $\langle 5, 6\rangle$ in Listing 4 (or Rectangle `checkValue()` in Fig. 1), the test case $t \leftarrow$`keys=[0, 1], value=0` has $distance(\langle 5, 6\rangle)_1 = 0.99$ while $distance(\langle 5, 6\rangle)_2 = 1.5$. Therefore, when $distance(\langle 5, 6\rangle)_1$ stuck on local optima, the fitness is still sensitive to $distance(\langle 5, 6\rangle)_2$ 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 `fbranch` 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.

Table 2: Heuristics for Filtering Experimental Methods

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

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.

- **Better Coverage:** Within the time budget, *Evoif* has a better coverage than *EvoSuite*.
- **Worse Coverage:** Within the time budget, *Evoif* has a worse coverage than *EvoSuite*.
- **Better Time:** Both approaches achieve same coverage, *Evoif* uses less time than *EvoSuite*.
- **Worse Time:** Both approaches achieve same coverage, *Evoif* uses more time than *EvoSuite*.
- **Equal:** This case is none of the case. That is, both approaches achieve the same coverage with the same time.

4.1.4 Runtime Configuration. *EvoSuite* provides a rich set of search algorithms, including monotonic genetic algorithm [24, 26], memetic algorithm [26], MOSA algorithm [43], DynaMOSA algorithm [44], etc. In this experiment, we select DynaMOSA algorithm [44] for the following reason.

Justification. A successful test evolution towards a target branch requires that the search algorithm addresses multiple challenges

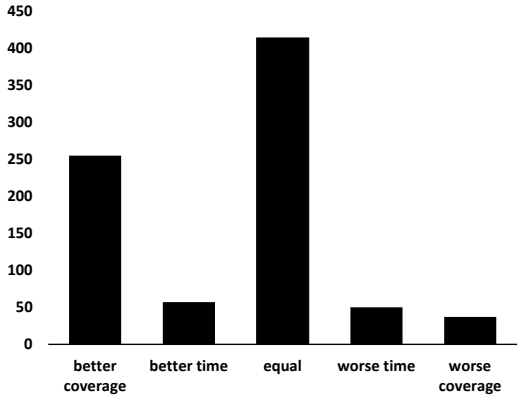

Figure 5: Overall comparison of *Evoif* vs *EvoSuite* on methods with interprocedural flags.

at the same time, for example applying appropriate mutations, designing effective optimization algorithm, etc. Therefore, choosing a lesser search algorithm causes that both *Evoif* and *EvoSuite* stuck on the *orthogonal* challenges, which prevents us from comparing *Evoif* and *EvoSuite* effectively. For example, MOSA algorithm will generate test cases for an unnecessary large number of optimization goals, making the Pareto frontier too large to select discriminative test cases to evolve. Thus, we only select the DynaMOSA algorithm which has been evaluated to outperform all other algorithms provided by *EvoSuite* [44]. DynaMOSA is a state-of-the-art multiple objective evolutionary algorithm. It outperforms single objective evolutionary algorithms and traditional multiple objective evolutionary algorithms on various coverage criteria. Its key advantage lies in the capability of dynamically prioritizing some uncovered branches on runtime to improve the testing efficiency.

We run our experiment on 14 nodes on NCL cloud in Singapore (https://ncl.sg/), each node is with Intel Xeon E5-2620 CPU of 2.1GHz and 64G DDR4 Memory. The detailed *EvoSuite* runtime configurations can be found in our tool website [1].

4.2 RQ1: Prevalence of IPF Problem

We observe that there are 78973 individual methods with program branches in all 150 projects, and 9306 (i.e., 11.8%) of them suffer from interprocedural flag problems. Moreover, we check the ratio of methods suffering from interprocedural flag problem for each project. The average ratio of methods that have at least one interprocedural flag is 14.4% and the median ratio is 11.9%. A more detailed project-wise IPF distribution can be referred in [1]. *In this regard, we conclude that the interprocedural flag problem is prevalent.*

4.3 RQ2: Performance

Fig. 5 shows how the comparisons are distributed among the 5 categories. Overall, 255 methods fall to "better coverage", 57 fall to "better time", 414 fall to "equal coverage", 31 fall to "worse time", and 50 fall to "worse coverage". We can see that (1) the number of methods in *Better Coverage* is much larger than that in *Worse Coverage* (more specifically, 255 (i.e., 31.6%) v.s. 50 (i.e., 6.2%), (2) the used time is similar for those methods where both *Evoif* and *EvoSuite* achieve same coverage (more specifically, 12.1s v.s. 14.5s on average and 4s v.s. 5s on median), and (3) there are a noticeable number of methods where both approaches have 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.

```
1  public void solvePhase1(final SimplexTableau tableau) throws
       OptimizationException {
2   ...
3    if (!MathUtils.equals(tableau.getRhs(), 0, this.epsilon)) {
4      throw new NoFeasibleSolutionException();
5    }
6  }
7
8  public static boolean equals(double x, double y, double eps) {
9    return equals(x, y) || FastMath.abs(y - x) <= eps;
10 }
```

Listing 6: Trivial Interprocedural Flag

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* gets 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.

```
boolean getMethod(String p1, String p2){
  ...
  if (p1.equals(p2)
    && signatureCorrect(p1)) {
    return false;
  }
}
```

Listing 7: Normal Parent Branch

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

degenerate to a simple random testing approach. Our work aims at recovering the disappeared gradients, which can facilitate and improve many different search-based software testing approaches.

5.2 Testability Transformation

Testability transformation [10, 13, 29, 31, 34, 40] 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. Baresel *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 instrumentating 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 *Evoif* 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

[1] [n. d.]. Anonymous Webiste. https://sites.google.com/view/evoipf/home. Accessed: 2019-05-13.
[2] [n. d.]. Apache Math. https://commons.apache.org/proper/commons-math/download_math.cgi. Accessed: 2020-01-27.
[3] [n. d.]. IDA SDK. https://www.hex-rays.com/products/ida/support/download.shtml. Accessed: 2020-01-27.
[4] [n. d.]. JFreechat. http://www.jfree.org/jfreechart/download.html. Accessed: 2020-01-27.
[5] [n. d.]. Weka. https://sourceforge.net/projects/weka/files/weka-3-8/3.8.0/. Accessed: 2020-01-27.
[6] Aldeida Aleti and Lars Grunske. 2015. Test Data Generation with a Kalman Filter-based Adaptive Genetic Algorithm. *J. Syst. Softw.* 103, C (May 2015), 343–352. https://doi.org/10.1016/j.jss.2014.11.035
[7] Aldeida Aleti, I. Moser, and Lars Grunske. 2017. Analysing the Fitness Landscape of Search-based Software Testing Problems. *Automated Software Engg.* 24, 3 (Sept. 2017), 603–621. https://doi.org/10.1007/s10515-016-0197-7
[8] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147.
[9] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2018. Towards Model Checking Android Applications. *IEEE Transactions on Software Engineering* 44, 6 (2018), 595–612.
[10] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. 2004. Evolutionary Testing in the Presence of Loop-assigned Flags: A Testability Transformation Approach. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 108–118. https://doi.org/10.1145/1007512.1007527
[11] André Baresel, Harmen Sthamer, and Michael Schmidt. 2002. Fitness Function Design to Improve Evolutionary Structural Testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO'02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1329–1336. http://dl.acm.org/citation.cfm?id=2955491.2955736
[12] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. 2009. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. 8, 2 (June 2009), 239–287. https://doi.org/10.1007/s11047-008-9098-4
[13] David W. Binkley, Mark Harman, and Kiran Lakhotia. 2011. FlagRemover: A Testability Transformation for Transforming Loop-assigned Flags. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 12 (Aug. 2011), 33 pages. https://doi.org/10.1145/2000791.2000796
[14] Oliver Bühler and Joachim Wegener. 2008. Evolutionary Functional Testing. *Comput. Oper. Res.* 35, 10 (Oct. 2008), 3144–3160. https://doi.org/10.1016/j.cor.2007.01.015
[15] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based Test Data Generation for SQL Queries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1220–1230. https://doi.org/10.1145/3180155.3180202
[16] Naipeng Dong, Hugo Jonker, and Jun Pang. 2013. Enforcing Privacy in the Presence of Others: Notions, Formalisations and Relations. In *Computer Security – ESORICS 2013*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.). 499–516.
[17] Naipeng Dong and Tim Muller. 2018. The Foul Adversary: Formal Models. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 11232. 37–53.
[18] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. 2016. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 185–195.
[19] Z. Dong, A. Andrzejak, and K. Shao. 2015. Practical and accurate pinpointing of configuration errors using static analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 171–180.
[20] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. 1–12.
[21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179
[22] Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 178–188. http://dl.acm.org/citation.cfm?id=2337223.2337245
[23] Gordon Fraser and Andrea Arcuri. 2013. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 362–369.
[24] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291.

[25] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[26] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A Memetic Algorithm for Whole Test Suite Generation. *J. Syst. Softw.* 103, C (May 2015), 311–327. https://doi.org/10.1016/j.jss.2014.05.032

[27] Wang Haijun, Xie Xiaofei, Li Yi, Wen Cheng, Li Yuekang, Liu Yang, Qin Shengchao, Chen Hongxu, and Sui Yulei. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM.

[28] M. Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Future of Software Engineering (FOSE '07)*. 342–357. https://doi.org/10.1109/FOSE.2007.29

[29] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. 2008. Formal Methods and Testing. Springer-Verlag, Berlin, Heidelberg, Chapter Testability Transformation: Program Transformation to Improve Testability, 320–344. http://dl.acm.org/citation.cfm?id=1806209.1806220

[30] M. Harman and J. Clark. 2004. Metrics are fitness functions too. In *10th International Symposium on Software Metrics, 2004. Proceedings.* 58–69. https://doi.org/10.1109/METRIC.2004.1357891

[31] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Trans. Softw. Eng.* 30, 1 (Jan. 2004), 3–16.

[32] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2012. Empirical Software Engineering and Verification. Springer-Verlag, Berlin, Heidelberg, Chapter Search Based Software Engineering: Techniques, Taxonomy, Tutorial, 1–59. http://dl.acm.org/citation.cfm?id=2184075.2184076

[33] Y. Jia and M. Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 249–258. https://doi.org/10.1109/SCAM.2008.36

[34] Yanchuan Li and Gordon Fraser. 2011. Bytecode Testability Transformation.

[35] Z. Li, M. Harman, and R. M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237. https://doi.org/10.1109/TSE.2007.38

[36] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 509–519.

[37] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-Based Debugging. In *Proceedings of the 39th International Conference on Software Engineering*. 393–403.

[38] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. DaPanda: Detecting Aggressive Push Notifications in Android Apps. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 66–78.

[39] Kulani Mahadewa, Kailong Wang, Guangdong Bai, Ling Shi, Jin Song Dong, and Zhenkai Liang. 2019. Scrutinizing Implementations of Smart Home Integrations. *IEEE Transactions on Software Engineering* (2019).

[40] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.

[41] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 223–226. https://doi.org/10.1109/TSE.1976.233818

[42] Duy Tai Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Minh Quang Tran. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. 1–12.

[43] A. Panichella, F. M. Kifetew, and P. Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[44] A. Panichella, F. M. Kifetew, and P. Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.

[45] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE '15)*. Springer, 93–108.

[46] U. Rueda, T. E. J. Vos, and I. S. W. B. Prasetya. 2015. Unit Testing Tool Competition – Round Three. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. 19–24. https://doi.org/10.1109/SBST.2015.12

[47] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[48] Anupama Surendran and Philip Samuel. 2017. Evolution or Revolution: The Critical Need in Genetic Algorithm Based Testing. *Artif. Intell. Rev.* 48, 3 (Oct. 2017), 349–395. https://doi.org/10.1007/s10462-016-9504-8

[49] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. TimeAware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1146238.1146240

[50] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining Regressions via Alignment Slicing and Mending. *IEEE Transactions on Software Engineering* (2019), 1–1.

[51] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating Vulnerabilities in Binaries via Memory Layout Recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 718–728. https://doi.org/10.1145/3338906.3338966

[52] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering*. 291–302.

[53] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information & Software Technology* 43, 14 (2001), 841–854. http://dblp.uni-trier.de/db/journals/infsof/infsof43.html#WegenerBS01

[54] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.

[55] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 61–72.

[56] Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. 2017. Loopster: static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 84–94.

[57] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 188–198.

[58] Xiong Xu, Ziming Zhu, and Li Jiao. 2017. An Adaptive Fitness Function Based on Branch Hardness for Search Based Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, New York, NY, USA, 1335–1342. https://doi.org/10.1145/3071178.3071184