It is not Only About Control Dependent Nodes: Basic Block Coverage for Search-Based Crash Reproduction

Pouria Derakhshanfar[1 [0000−0003−3549−9019]], Xavier Devroey[1 [0000−0002−0831−7606]], and Andy Zaidman[1 [0000−0003−2413−3935]]

Delft University of Technology, Delft, The Netherlands
p.derakhshanfar@tudelft.nl, x.d.m.devroey@tudelft.nl, a.e.zaidman@tudelft.nl

Abstract. Search-based techniques have been widely used for white-box test generation. Many of these approaches rely on the *approach level* and *branch distance* heuristics to guide the search process and generate test cases with high line and branch coverage. Despite the positive results achieved by these two heuristics, they only use the information related to the coverage of explicit branches (*e.g.,* indicated by conditional and loop statements), but ignore potential implicit branchings within basic blocks of code. If such implicit branching happens at runtime (*e.g.,* if an exception is thrown in a branchless-method), the existing fitness functions cannot guide the search process. To address this issue, we introduce a new secondary objective, called Basic Block Coverage (*BBC*), which takes into account the coverage level of relevant basic blocks in the control flow graph. We evaluated the impact of *BBC* on *search-based crash reproduction* because the implicit branches commonly occur when trying to reproduce a crash, and the search process needs to cover only a few basic blocks (*i.e.,* blocks that are executed before crash happening). We combined *BBC* with existing fitness functions (namely *STDistance* and *WeightedSum*) and ran our evaluation on 124 hard-to-reproduce crashes. Our results show that *BBC*, in combination with *STDistance* and *WeightedSum*, reproduces 6 and 1 new crashes, respectively. *BBC* significantly decreases the time required to reproduce 26.6% and 13.7% of the crashes using *STDistance* and *WeightedSum*, respectively. For these crashes, *BBC* reduces the consumed time by 44.3% (for *STDistance*) and 40.6% (for *WeightedSum*) on average.

Keywords: automated crash reproduction · search-based software testing · evolutionary algorithm · secondary objective.

1 Introduction

Various search-based techniques have been introduced to automate different white-box test generation activities (*e.g.,* unit testing [9, 10], integration testing [7], system-level testing [2], *etc.*). Depending on the testing level, each of

these approaches utilizes dedicated fitness functions to guide the search process and produce a test suite satisfying given criteria (*e.g.,* line coverage, branch coverage, *etc.*).

Fitness functions typically rely on *control flow graphs (CFGs)* to represent the source code of the software under test [12]. Each node in a CFG is a *basic block* of code (*i.e.,* maximal linear sequence of statements with a single entry and exit point without any internal branch), and each edge represents a possible *execution flow* between two blocks. Two well-known heuristics are usually combined to achieve high line and branch coverages: the *approach level* and the *branch distance* [12]. The former measures the distance between the execution path of the generated test and a target basic block (*i.e.,* a basic block containing a statement to cover) in the CFG. The latter measures, using a set of rules, the distance between an execution and the coverage of a *true* or *false* branch of a particular predicate in a branching basic block of the CFG.

Both *approach level* and *branch distance* assume that only a limited number of basic blocks (*i.e., control dependent* basic blocks [1]) can change the execution path away from a target statement (*e.g.,* if a target basic block is the true branch of an conditional statement). However, basic blocks are not atomic due to the presence of **implicit branches** [4] (*i.e.,* branches occuring due to the exceptional behavior of instructions). As a consequence, any basic block between the entry point of the CFG and the target basic block can impact the execution of the target basic block. For instance, a generated test case may stop its execution in the middle of a basic block with a runtime exception thrown by one of the statements of that basic block. In these cases, the search process does not benefit from any further guidance from the approach level and branch distance.

Fraser and Arcuri [11] introduced testability transformation, which instruments the code to guide the unit test generation search to cover implicit exceptions happening in the class under test. However, this approach does not guide the search process in scenarios where an implicit branch happens in the other classes called by the class under test. This is because of the extra cost added to the process stemming from the calculation and monitoring of the implicit branches in all of the classes, coupled with the class under test. For instance, the class under test may be heavily coupled with other classes in the project, thereby finding implicit branches in all of these classes can be expensive.

However, for some test case generation scenarios, like **crash reproduction**, we aim to cover a limited number of paths, and thereby we only need to analyse a limited number of basic blocks [5, 13, 16, 19, 21]. Current crash reproduction approaches rely on information about a reported crash (*e.g.,* stack trace, core dump *etc.*) to generate a **crash reproducing test case (CRT)**

Among these approaches, search-based crash reproduction [16, 19] takes as input a **stack trace** to guide the generation process. More specifically, the statements pointed by the stack trace act as target statements for the approach level and branch distance. Hence, current search-based crash reproduction techniques suffer from the lack of guidance in cases where the involved basic blocks contain implicit branches (which is common when trying to reproduce a crash).

Listing 1.1. Method `fromMap` from XWIKI version 8.1 [17]

```
402     public BaseCollection fromMap(Map<[...]> map, BaseCollection object){
403         for (PropertyClass property : (Collection<[...]>) getFieldList()) {
404             String name = property.getName();
405             Object formvalues = map.get(name);
406             if (formvalues != null) {
407                 BaseProperty objprop;
408                 if (formvalues instanceof String[]) {
409                     [...]
410                 } else if (formvalues instanceof String) {
411                     objprop = property.fromString(formvalues.toString());
412                 } else {
413                     objprop = property.fromValue(formvalues);
414                 }
415                 [...]
416             }}
417         return object;}
```

This paper introduces a novel secondary objective called **Basic Block Coverage (*BBC*)** to address this guidance problem in crash reproduction. *BBC* helps the search process to compare two generated test cases with the same distance (according to approach level and branch distance) to determine which one is closer to the target statement. In this comparison, *BBC* analyzes the coverage level, achieved by each of these test cases, of the basic blocks in between the closest covered control dependent basic block and the target statement.

To assess the impact of *BBC* on search-based crash reproduction, we re-implemented the existing *STDistance* [16] and *WeightedSum* [19] fitness functions and empirically compared their performance with and without using *BBC* (4 configurations in total). We applied these four crash reproduction configurations to 124 hard-to-reproduce crashes introduced as JCrashPack [17], a crash benchmark used by previous crash reproduction studies [8]. We compare the performances in terms of *effectiveness in crash reproduction ratio* (*i.e.,* percentage of times that an approach can reproduce a crash) and *efficiency* (*i.e.,* time required by for reproducing a crash).

Our results show that *BBC* significantly improves the crash reproduction ratio over the 30 runs in our experiment for respectively 5 and 1 crashes when compared to using *STDistance* and *WeightedSum* without any secondary objective. Also, *BBC* helps these two fitness functions to reproduce 6 (for *STDistance*) and 1 (for *WeightedSum*) crashes that they could not be reproduced without secondary objective. Besides, on average, *BBC* increases the crash reproduction ratio of *STDistance* by 4%. Applying *BBC* also significantly reduces the time consumed for crash reproduction guided by *STDistance* and *WeightedSum* in 33 (26.6% of cases) and 14 (13.7% of cases) crashes, respectively, while it was significantly counter productive in only one case. In cases where *BBC* has a significant impact on efficiency, this secondary objective improves the average efficiency of *STDistance* and *WeightedSum* by 40.6% and 44.3%, respectively.

2 Background

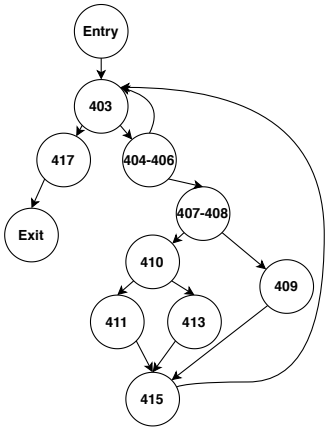


Fig. 1. CFG for method `fromMap`

2.1 Coverage Distance Heuristics

Many structural-based search-based test generation approaches mix the *branch distance* and *approach level* heuristics to achieve a high line and branch coverage [12]. These heuristics measure the distance between a test execution path and a specific statement or a specific branch in the software under test. For that, they rely on the coverage information of *control dependent basic blocks*, *i.e.,* basic blocks that have at least one outgoing edge leading the execution path toward the *target basic block* (containing the targeted statement) and at least another outgoing edge leading the execution path away from the target basic block. As an example, Listing 1.1 shows the source code of method `fromMap` in XWIKI[1], and Figure 1 contains the corresponding CFG. In this graph, the basic block `409` is control dependent on the basic block `407-408` because the execution of line 409 is dependent on the satisfacation of the predicate at line 408 (*i.e.,* line 409 will be executed only if elements of array `formvalues` are `String`).

The *approach level* is the number of uncovered control dependent basic blocks for the target basic block between the closest covered control dependent basic block and the target basic block. The *branch distance* is calculated from the predicate of the closest covered control dependent basic block, based on a set of predefined rules. Assuming that the test t covers only line `403` and `417`, and our target line is `409`, the approach level is 2 because two control dependent basic blocks (`404-406` and `407-408`) are not covered by t. The branch distance the predicate in line `403` (the closest covered control dependency of node `409`) is measured based on the rules from the establised technique [12].

To the best of our knowledge, there is no related work studying the extra heuristics helping the combination of approach level and branch distance to improve the coverage. Most related to our work, Panichella *et al.* [14] and Rojas *et al.* [15] introduced two heuristics called *infection distance* and *propagation distance*, to improve the weak mutation score of two generated test cases. However,

[1] https://github.com/xwiki

these heuristics do not help the search process to improve the general statement coverage (*i.e.,* they are effective only after covering a mutated statement).

In this paper, we introduce a new secondary objective to improve the statement coverage achieved by fitness functions based on the approach level and branch distance, and analyze the impact of this secondary objective on **search-based crash reproduction**.

Listing 1.2. XWIKI-13377 crash stack trace [17]

```
0  java.lang.ClassCastException: [...]
1      at [...].BaseStringProperty.setValue(BaseStringProperty.java:45)
2      at [...].PropertyClass.fromValue(PropertyClass.java:615)
3      at [...].BaseClass.fromMap(BaseClass.java:413)
4      [...]
```

2.2 Search-based Crash Reproduction

After a crash is reported, one of the essential steps of software debugging is to write a **Crash Reproducing Test case (CRT)** to make the crash observable to the developer and help them in identifying the root cause of the failure [22]. Later, this CRT can be integrated into the existing test suite to prevent future regressions. Despite the usefulness of a CRT, the process of writing this test can be labor-intensive and time-taking [19]. Various techniques have been introduced to automate the reproduction of a crash [5,13,16,19,21], and search-based approaches (EvoCrash [19] and ReCore [16]) yielded the best results [19].

EvoCrash. This approach utilizes a single-objective genetic algorithm to generate a CRT from a given stack trace and a *target frame* (*i.e.,* a frame in the stack trace that its class will be used as the class under test). The CRT generated by EvoCrash throws the same stack trace as the given one up to the target frame. For example, by passing the stack trace in Listing 1.2 and target frame 3 to EvoCrash, it generates a test case reproducing the first three frames of this stack trace (*i.e.,* thrown stack trace is identical from line 0 to 3).

EvoCrash uses a fitness function, called *WeightedSum*, to evaluate the candidate test cases. *WeightedSum* is the sum scalarization of three components: (i) the **target line coverage** (d_s), which measures the distance between the execution trace and the *target line* (*i.e.,* the line number pointed to by the target frame) using *approach level* and *branch distance*; (ii) the **exception type coverage** (d_e), determining whether the type of the triggered exception is the same as the given one; and (iii) the **stack trace similarity** (d_{tr}), which indicates whether the stack trace triggered by the generated test contains all frames (from the most in-depth frame up to the target frame) in the given stack trace.

Definition 1 (*WeightedSum* [19]). *For a given test case execution t, the WeightedSum (ws) is defined as follows:*

$$ws(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_{tr}) & \textit{if line not reached} \\ 3 \times min(d_s) + 2 \times d_e(t) + max(d_{tr}) & \textit{if line reached} \\ 3 \times min(d_s) + 2 \times min(d_e) + d_{tr}(t) & \textit{if exception thrown} \end{cases} \quad (1)$$

Where $d_s(t) \in [0,1]$ indicates how far t is from reaching the target line and is computed using the normalized approach level and branch distance: $d_s(t) = \|approachLevel_s(t) + \|branchDistance_s(t)\|\|$. Also, $d_e(t) \in \{0,1\}$ shows if the type of the exception thrown by t is the same as the given stack trace (0) or not (1). Finally, $d_{tr}(t) \in [0,1]$ measures the stack trace similarity between the given stack trace and the one thrown by t. $max(f)$ and $min(f)$ denote the maximum and minimum possible values for a function f, respectively. In this fitness function, $d_e(t)$ and $d_{tr}(t)$ are only considered in the satisfaction of two *constraints*: (i) *exception type coverage* is relevant only when we reach the target line and (ii) *stack trace similarity* is important only when we both reach the target line and throw the same type of exception.

As an example, when applying EvoCrash on the stack trace from Listing 1.2 with the target frame 3, *WeightedSum* first checks if the test cases generated by the search process reach the statement pointed to by the target frame (line 413 in class `BaseClass` in this case). Then, it checks if the generated test can throw a `ClassCastException` or not. Finally, after fulfilling the first two constraints, it checks the similarity of frames in the stack trace thrown by the generated test case against the given stack trace in Listing 1.2.

EvoCrash uses **guided** initialization, mutation and single-point crossover operators to ensure that the target method (*i.e.,* the method appeared in the target frame) is always called by the different tests during the evolution process.

According to a recent study, EvoCrash outperforms other non-search-based crash reproduction approaches in terms of *effectiveness in crash reproduction* and *efficiency* [19]. This study also shows the helpfulness of tests generated by EvoCrash for developers during debugging.

In this paper, we assess the impact of *BBC* as the secondary objective in the EvoCrash search process.

ReCore. This approach utilizes a genetic algorithm guided by a single fitness function, which has been defined according to the core dump and the stack trace produced by the system when the crash happened. To be more precise, this fitness function is a sum scalarization of three sub-functions: (i) **TestStackTraceDistance**, which guides the search process according to the given stack trace; (ii) **ExceptionPenalty**, which indicates whether the same type of exception as the given one is thrown or not (identical to ExceptionCoverage in EvoCrash); and (iii) **StackDumpDistance**, which guides the search process by the given core dump.

Definition 2 (*TestStackTraceDistance* [16]). *For a given test case execution t, the TestStackTraceDistance (STD) is defined as follows:*

$$STD(R,t) = |R| - lcp - (1 - StatementDistance(s)) \tag{2}$$

Where $|R|$ is the number of frames in the given stack trace. And lcp is the longest common prefix frames between the given stack trace and the stack trace thrown by t. Concretely, $|R| - lcp$ is the number of frames not covered by t. Moreover, $StatementDistance(s)$ is calculated using the sum of the approach level and the normalized branch distance to reach the statement s, which is pointed

to by the first (the utmost) uncovered frame by t: $StatementDistance(s) = approachLevel_s(t) + \|branchDistance_s(t)\|$.

Since using runtime data (such as core dumps) can cause significant overhead [5] and leads to privacy issues [13], the performance of ReCore in crash reproduction was not compared with EvoCrash in prior studies [19]. Although, two out of three fitness functions in ReCore use only the given stack trace to guide the search process. Hence, this paper only considers *TestStackTraceDistance + ExceptionPenalty* (called *STDistance* hereafter).

As an example, when applying ReCore with *STDistance* on the stack trace in Listing 1.2 with target frame 3, first, *STDistance* determines if the generated test covers the statement at frame 3 (line 413 in class `BaseClass`). Then, it checks the coverage of frame 2 (line 615 in class `PropertyClass`). After covering the first two frames by the generated test case, it checks the coverage of the statement pointed to by the deepest frame (line 45 in class `BaseStringProperty`). For measuring the coverage of each of these statements, *STDistance* uses the approach level and branch distance. After covering all of the frames, this fitness function checks if the the generated test throws `ClassCastException` in the deepest frame.

In this study, we perform an empirical evaluation to assess the performance of crash reproduction using *STDistance* with and without *BBC* as the secondary objective in terms of *effectiveness in crash reproduction* and *efficiency*.

3 Basic Block Coverage

3.1 Motivating Example

During the search process, the fitness of a test case is evaluated using a fitness function, either *WeightedSum* or *STDistance*. Since the search-based crash reproduction techniques model this task to a minimization problem, the generated test cases with lower fitness values have a higher chance of being selected and evolved to generate the next generation. One of the main components of these fitness functions is the coverage of specific statements pointed by the given stack trace. The distance of the test case from the target statement is calculated using the approach level and branch distance heuristics. As we have discussed in Section 2.1, the approach level and branch distance cannot guide the search process if the execution stops because of implicit branches in the middle of basic blocks (*e.g.,* a thrown `NullPointerException` during the execution of a basic block). As a consequence, these fitness functions may return the same fitness value for two tests, although the tests do not cover the same statements in the block of code where the implicit branching happens.

For instance, assume that the search process for reproducing the crash in Figure 1.2 generates two test cases T_1 and T_2. The first step for these test cases is to cover frame 3 in the stack trace (line `413` in BaseClass). However, T_1 stops the execution at line `404` due to a `NullPointerException` thrown in method `getName`, and T_2 throws a `NullPointerException` at line `405` because

Listing 1.3. *BBC* secondary objective computation algorithm

```
1  input: test T1, test T2, String method, int line
2  output: int
3  begin
4      FCB1 ← fullyCoveredBlocks(T1,method,line);
5      FCB2 ← fullyCoveredBlocks(T2,method,line);
6      SCB1 ← semiCoveredBlocks(T1,method,line);
7      SCB2 ← semiCoveredBlocks(T2,method,line);
8
9      if (FCB1 ⊂ FCB2 ∧ SCB1 ⊂ SCB2) ∨ (FCB2 ⊂ FCB1 ∧ SCB2 ⊂ SCB1):
10         return size(FCB2 ∪ SCB2) - size(FCB1 ∪ SCB1)
11     else if FCB1 = FCB2 ∧ SCB1 = SCB2:
12         closestBlock ← closestSemiCoveredBlocks(SCB1, method, line);
13         coveredLines1 ← getCoveredLines(T1,closestBlock);
14         coveredLines2 ← getCoveredLines(T2,closestBlock);
15         return size(coveredLines2) - size(coveredLines1);
16     else:
17         return 0;
18 end
```

it passes a null value input argument to `map`. Even though T_2 covers more lines, the combination of approach level and branch distance returns the same fitness value for both of these test cases: approach level is 2 (nodes `407-408` and `410`) and branch distance is measured according to the last predicate. This is because these two heuristics assume that each basic block is atomic, and by covering line `404`, it means that lines `405` and `406` are covered, as well.

3.2 Secondary Objective

The goal of the Basic Block Coverage (*BBC*) secondary objective is to prioritize the test cases with the same fitness value according to their coverage within the basic blocks between the closest covered control dependency and the target statement. At each iteration of the search algorithm, test cases with the same fitness value are compared with each other using *BBC*. Listing 1.3 presents the pseudo-code of the *BBC* calculation. Inputs of this algorithm are two test cases T_1 and T_2, which both have the same fitness value (calculated either using *WeightedSum* or *STDistance*), as well as line number and method name of the target statement. This algorithm compares the coverage of basic blocks on the path between the entry point of the CFG of the given method and the basic block that contains the target statement (called *effective blocks* hereafter) achieved by T_1 and T_2. If *BBC* determines there is no preference between these two test cases, it returns 0. Also, it returns a value < 0 if T_1 has higher coverage compared to T_2, and vice versa. A higher absolute value of the returned integer indicates a bigger distance between the given test cases.

In the first step, *BBC* detects the effective blocks fully covered by each given test case (*i.e.,* the test covers all of the statements in the block) and saves them in two sets called $\texttt{FCB}_1$ and $\texttt{FCB}_2$ (lines 4 and 5 in Listing 1.3). Then, it detects the effective blocks semi-covered by each test case (*i.e.,* blocks where the test covers the first line but not the last line) and stores them in $\texttt{SCB}_1$ and $\texttt{SCB}_2$ (lines 6 and 7). The semi-covered blocks indicate the presence of implicit branches. Next,

BBC checks if both fully and semi-covered blocks of one of the tests are subsets of the blocks covered by the other test (line 9). In this case, the test case that covers the most basic blocks is the winner. Hence, *BBC* returns the number of blocks only covered by the winner test case (line 10). If *BBC* determines T_2 wins over T_1, the returned value will be positive, and vice versa.

If none of the test cases subsumes the coverage of the other one, *BBC* checks if the blocks covered by T_1 and T_2 are identical (line 11). If this is the case, *BBC* checks if one of the tests has a higher line coverage for the semi-covered blocks closest to the target statement (lines 12 to 15). If this is the case, *BBC* will return the number of lines in this block covered only by the winning test case. If the lines covered are the same for T_1 and T_2 (*i.e.,* `coveredLines1` and `coveredLines2` have the same size), there is no difference between these two test cases and *BBC* returns value 0 (line 15). Finally, if each of the given tests has a unique covered block in the given method (*i.e.,* the tests cover different paths in the method), *BBC* cannot determine the winner and returns 0 (lines 16 and 17) because we do not know which path leads to the crash reproduction.

Example. When giving two tests with the same fitness value (calculated by the primary objective) T_1 and T_2 from our motivation example to *BBC* with target method `fromMap` and line number 413 (according to the frame 3 of Figure 1.2), this algorithm compares their fully and semi-covered blocks with each other. In this example both T_1 and T_2 cover the same basic blocks: the fully covered block is `403` and the semi-covered block is `404-406`. So, *BBC* checks the number of lines covered by T_1 and T_2 in block `404-406`. Since T_1 stopped its execution at line 404, the number of lines covered by this test is 1. In contrast, T_2 managed to execute two lines (404 and 405). Hence, *BBC* returns $size(coveredLines2) - size(coveredLines1) = 1$. The positive return value indicates that T_2 is closer to the target statement and therefore, it should have higher chance to be selected for the next generation.

Branchless Methods. *BBC* can also be helpful for branchless methods. Since there are no control dependent nodes in branchless methods, approach level and branch distance cannot guide the search process in these cases. For instance, methods from frames 1 and 2 in Figure 1.2 are branchless. So, we expect that *BBC* can help the current heuristics to guide the search process toward covering the most in-depth statement.

4 Empirical Evaluation

To assess the impact of *BBC* on search-based crash reproduction, we perform an empirical evaluation to answer the following research questions:

RQ_1: *What is the impact of BBC on crash reproduction in terms of effectiveness in crash reproduction ratio?*

RQ_2: *What is the impact of BBC on the efficiency of crash reproduction?*

In these two RQs we want to evaluate the effect of *BBC* on the existing fitness functions, namely *STDistance* and *WeightedSum*, from two perspectives: effectiveness on crash reproduction ratio and efficiency.

4.1 Setup

Implementation. Since ReCore and EvoCrash are not openly available, we implement *BBC* in Botsing, an extensible, well-tested, and open-source search-based crash reproduction framework already implementing the *WeightedSum* fitness function and the guided initialization, mutation, and crossover operators. We also implement *STDistance* (ReCore fitness function) in this tool. Botsing relies on EvoSuite [9], an open-source search-based tool for unit test generation, for code instrumentation and test case generation by using `evosuite-client` as a dependency. We also implement the *STDistance* fitness function used as baseline in this paper.

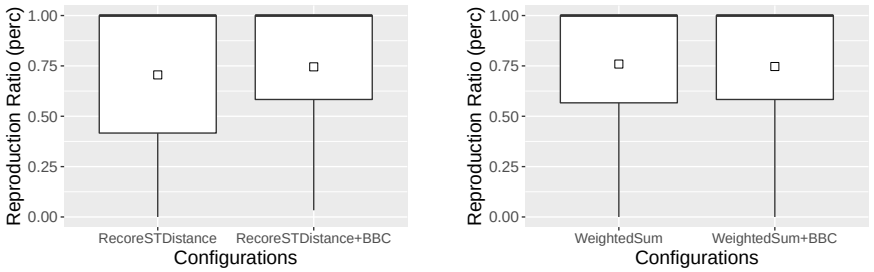
Crash selection. We select crashes from JCrashPack [17], a benchmark containing hard-to-reproduce Java crashes. We apply the two fitness functions with and without using *BBC* as a secondary objective to 124 crashes, which have also been used in a recent study [8]. These crashes stem from six open-source projects: JFreeChart, Commons-lang, Commons-math, Mockito, Joda-time, and XWiki. For each crash, we apply each configuration on each frame of the crash stack traces. We repeat each execution 30 times to take randomness into account, for a total number of 114,120 independent executions. We run the evaluation on two servers with 40 CPU-cores, 128 GB memory, and 6 TB hard drive.

Parameter settings. We run each search process with five minutes budget and set the population size to 50 individuals, as suggested by previous studies on search-based test generation [14]. Moreover, as recommended in prior studies on search-based crash reproduction [19], we use the *guided mutation* with a probability $p_m = 1/n$ (n = length of the generated test case), and the *guided crossover* with a probability $p_c = 0.8$ to evolve test cases. We do note that prior studies do not investigate the sensitivity of the crash reproduction to these probabilities. Tuning these parameters should be undertaken as future works.

4.2 Data Analysis

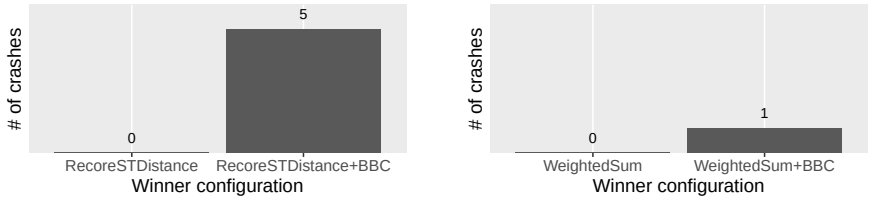
To evaluate the crash reproduction ratio (*i.e.,* the ratio of success in crash reproduction in 30 rounds of runs) of different assessed configurations ($\mathbf{RQ}_1$), we follow the same procedure as previous studies [8,18]: for each crash C, we detect the highest frame that can be reproduced by at least one of the configurations (r_{max}). We examine the crash reproduction ratio of each configuration for crash C targeting frame r_{max}. Since crash reproduction data has a dichotomic distribution (*i.e.,* an algorithm reproduces a crash C from its r_{max} or not), we use the Odds Ratio (OR) to measure the impact of each algorithm in crash reproduction ratio. A value $OR > 0$ in a comparison between a pair of factors (A, B) indicates that the application of factor A increases the crash reproduction ratio, while $OR < 0$ indicates the opposite. Also, a value of $OR = 0$ indicates that both of the factors have the same performance. We apply Fisher's exact test, with $\alpha = 0.01$ for the Type I error, to assess the significance of results.

To evaluate the efficiency of different configurations ($\mathbf{RQ}_2$), we analyze the time spent by each configuration on generating a crash reproducing test case.



(a) *STDistance* (b) *WeightedSum*

Fig. 2. Crash reproduction ratio (out of 30 executions) of fitness functions with and without *BBC*. (□) denotes the arithmetic mean and the bold line (—) is the median.



(a) *STDistance* (b) *WeightedSum*

Fig. 3. Pairwise comparison of impact of *BBC* on each fitness function in terms of crash reproduction ratio with a statistical significance < 0.01.

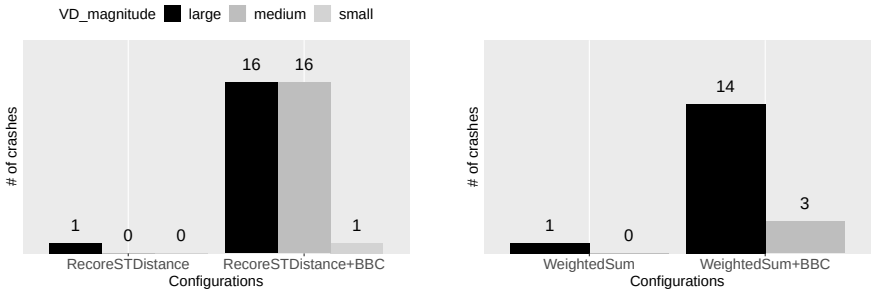
We do note that the extra pre-analysis and basic block coverage in *BBC* is considered in the spent time. Since measuring efficiency is only possible for the reproduced crashes, we compare the efficiency of algorithms on the crashes that are reproduced at least once by one of the algorithms. In executions that an algorithm failed to reproduce a crash, we assume that it reached the maximum allowed budget (5 minutes).

In this study, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [20] to examine the effect size of differences between using and not using *BBC* for efficiency. For a pair of factors (A, B) a value of $\hat{A}_{12} > 0.5$ indicates that A reproduces the target crash in a longer time, while a value of $\hat{A}_{12} < 0.5$ shows the opposite. Also, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. In addition, to assess the significance of effect sizes ($\hat{A}_{12}$), we utilize the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.01$ for the Type I error.

A replication package of this study has been uploaded to Zenodo [6].

5 Results

Crash reproduction effectiveness ($\mathbf{RQ_1}$) Figure 2 presents the crash reproduction ratio of the search processes guided by *STDistance* (Figure 2a) and *WeightedSum* (Figure 2b), with and without *BBC* as a secondary objective. This



(a) *STDistance* (b) *WeightedSum*

Fig. 4. Pairwise comparison of impact of *BBC* on each fitness function in terms of efficiency with a small, medium, and large effect size $\hat{A}_{12} < 0.5$ and a statistical significance < 0.01.

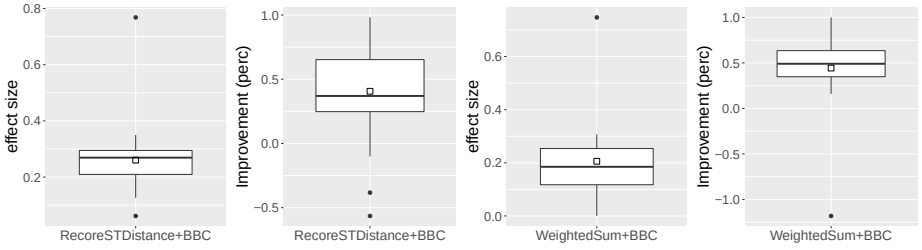
figure shows that the crash reproduction ratio of *WeightedSum* improves slightly when using *BBC*. However, on average, the crash reproduction ratio achieved by *STDistance* + *BBC* is 4% better than *STDistance* without *BBC*. Also, the lower quartile of crash reproduction ratio using *STDistance* has been improved by about 30% by utilizing *BBC*.

Figure 3 depicts the number of crashes, for which *BBC* has a significant impact on the effectiveness of crash reproduction guided by *STDistance* (Figure 3a) and *WeightedSum* (Figure 3b). *BBC* significantly improves the crash reproduction ratio in 5 and 1 crashes for fitness functions *STDistance* and *WeightedSum*, respectively. Importantly, the application of this secondary objective does not have any significant negative effect on crash reproduction. Also, *BBC* helps *STDistance* and *WeightedSum* to reproduce 6 and 1 new crashes, respectively (in at least one out of 30 runs), that could not be reproduced without *BBC*.

Summary. *BBC* slightly improves the crash reproduction ratio when using the *WeightedSum* fitness function. However, on average, *BBC* achieves a higher improvement when used as a secondary objective with the *STDistance* function.

Crash reproduction efficiency ($\mathbf{RQ_2}$) Figure 4 illustrates the number of crashes, in which *BBC* significantly affects the time consumed by the crash reproduction search process. As Figure 4b shows, *BBC* significantly improves the speed of crash reproduction guided by *WeightedSum* in 17 crashes (13.7% of cases), while it lost efficiency in the reproduction of only one crash. In cases that *BBC* significantly improves the efficiency of *WeightedSum*, on average, the efficiency is improved for about 40%. Moreover, Figure 4a shows that *BBC* has a higher positive impact on the efficiency of the search process guided by *STDistance*. It significantly reduces the time consumed by the search process in 33 crashes (26.6% of cases), while it had an adverse impact on the reproduction efficiency of only one crash. In cases that *BBC* significantly improves the efficiency of *STDistance*, on average, the efficiency is improved for about 53%.

Figure 5 depicts the average improvements in the efficiency and effect sizes for crashes where the difference in the consumed budget when using *BBC* or not



(a) *STDistance* (b) *WeightedSum*

Fig. 5. The effect size and the average improvement achieved by *BBC* on each of the fitness functions in cases that *BBC* makes a significant difference in terms of efficiency.

was significant. According to the right-side plot in Figure 5a, *BBC* reduces the time consumed by the search process guided by *STDistance* up to 98% (being 40.6% on average). Also, the left-side plot indicates that the average effect size of differences between *STDistance* and *STDistance* +*BBC* (calculated by Vargha-Delaney) is 0.26 (lower than 0.5 indicates that *BBC* improved the efficiency). Figure 5b shows that the average improvement (right-side plot) achieved by using *BBC* as the second objective of *WeightedSum* is 44.3%, and the average effect size (left-side plot), in terms of the crash reproduction efficiency, is 20.5.

Summary. *BBC* improves the efficiency of the search process with both of the crash reproduction fitness functions.

6 Discussion

Generally, using *BBC* as secondary objective leads to a better crash reproduction ratio and higher efficiency in search-based crash reproduction. This improvement is achieved thanks to the additional ability to guide the search process when facing implicit branches during the search. Combining *BBC* with *STDistance* shows an important improvement compared to the combination of *BBC* with *WeightedSum*. This result was expected, since only one (out of three) component in *WeightedSum* is allocated to line coverage, and thereby most parts of the fitness function do not use the approach level and branch distance heuristics. In contrast, *STDistance* uses the approach level and branch distance to cover each of the frames in the given stack trace incrementally.

Our results show that *BBC* helps the crash reproduction process to reproduce new crashes. For instance, the crash that we used in this study (XWIKI-13377) can be reproduced only by *STDistance* + *BBC*. Considering our results, we believe that the usage of approach level and branch distance can be improved in other areas of search-based test generation (*e.g.,* unit testing) by taking the *implicit branches* into account. However, it can be expensive to apply this secondary objective in cases where the search process tries to cover multiple paths. Assessing the impact of *BBC* on other search-based test generation techniques is part of our future research agenda.

Threats to validity. We cannot guarantee that our implementation of Botsing is bug-free. However, we mitigated this threat by testing our tool and manually examining some samples of the results. We cannot ensure that our results are generalizable to all crashes. However, we used an earlier established benchmark for crash reproduction containing 124 hard-to-reproduce crashes provoked by real bugs in a variety of open-source applications. Moreover, by following the guidelines of the related literature [3], we executed each configuration 30 times to take the randomness of the search process into account. Finally, we provide Botsing as an open-source tool. Also, the data and the processing scripts used to present the results are available as a replication package on Zenodo [6].

7 Conclusion and Future work

Approach level and branch distance are two well-known heuristics, widely used by search-based test generation approaches to guide the search process towards covering target statements and branches. These heuristics measure the distance of a generated tests from covering the target using the coverage of control dependencies. However, these two heuristics do not consider implicit branches. For instance, if a test throws an exception during the execution of a non-branch statement, approach level and branch distance cannot guide the search process to tackle this exception. In this paper, we introduced a secondary objective called *BBC* to address this issue. To assess *BBC*, we used it for search-based crash reproduction due to the high chance of implicit branch occurrence and the limited number of basic blocks that should be covered. Our results show that *BBC* helps *STDistance* and *WeightedSum* to reproduce 6 and 1 new crashes, respectively. Also, *BBC* significantly improves the efficiency in 26.6% and 13.7% of the crashes using *STDistance* and *WeightedSum*, respectively.

In our future work, we will investigate the application of *BBC* for other search-based test generation techniques (such as unit and integration).

Acknowledgements

The authors would like to thank Carolin Brandt for her valuable feedback on the paper. This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No.731529).

References

1. Allen, F.E.: Control flow analysis. vol. 5, pp. 1–19. ACM, New York, NY, USA (1970). https://doi.org/10.1145/390013.808479
2. Arcuri, A.: RESTful API automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM) **28**(1), 1–37 (2019)
3. Arcuri, A., Briand, L.: A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability **24**(3), 219–250 (2014). https://doi.org/10.1002/stvr.1486

4. Borba, P., Cavalcanti, A., Sampaio, A., Woodcook, J.: Testing techniques in software engineering: Second pernambuco summer school on software engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures, vol. 6153. Springer (2010)
5. Chen, N., Kim, S.: STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. on Software Engineering **41**(2), 198–220 (2015)
6. Derakhshanfar, P., Devroey, X.: Replication package of Basic Block Coverage for Search-Based Crash Reproduction. https://doi.org/10.5281/zenodo.3953519, https://doi.org/10.5281/zenodo.3953519
7. Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., van Deursen, A.: Towards integration-level test case generation using call site information. arXiv preprint arXiv:2001.04221 (2020)
8. Derakhshanfar, P., Devroey, X., Perrouin, G., Zaidman, A., van Deursen, A.: Search-based crash reproduction using behavioural model seeding. Software Testing Verification and Reliability (2020). https://doi.org/10.1002/stvr.1733
9. Fraser, G., Arcuri, A.: 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. pp. 416–419 (2011)
10. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2012)
11. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empirical software engineering **20**(3), 611–639 (2015)
12. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)
13. Nayrolles, M., Hamou-Lhadj, A., Tahar, S., Larsson, A.: Jcharming: A bug reproduction approach using crash traces and directed model checking. In: Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). pp. 101–110. IEEE (2015)
14. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2018)
15. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: International Symposium on Search Based Software Engineering (SSBSE). pp. 93–108. Springer (2015)
16. Rößler, J., Zeller, A., Fraser, G., Zamfir, C., Candea, G.: Reconstructing core dumps. In: Proc. International Conference on Software Testing, Verification and Validation (ICST). pp. 114–123. IEEE (2013)
17. Soltani, M., Derakhshanfar, P., Devroey, X., van Deursen, A.: A benchmark-based evaluation of search-based crash reproduction. Empirical Software Engineering **25**(1), 96–138 (jan 2020)
18. Soltani, M., Derakhshanfar, P., Panichella, A., Devroey, X., Zaidman, A., van Deursen, A.: Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In: Symposium on Search-Based Software Engineering (SSBSE). LNCS, vol. 11036, pp. 325–340. Springer (2018)
19. Soltani, M., Panichella, A., Van Deursen, A.: Search-based crash reproduction and its impact on debugging. IEEE Trans. on Software Engineering pp. 1–1 (2018)
20. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)

21. Xuan, J., Xie, X., Monperrus, M.: Crash reproduction via test case mutation: Let existing test cases help. In: Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 910–913. ACM (2015)
22. Zeller, A.: Why programs fail, Second Edition: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edn. (2009)