Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?

Sina Shamshiri, José Miguel Rojas, Gordon Fraser, Phil McMinn
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello, Sheffield, UK, S1 4DP

ABSTRACT

Achieving high structural coverage is an important aim in software testing. Several search-based techniques have proved successful at automatically generating tests that achieve high coverage. However, despite the well-established arguments behind using evolutionary search algorithms (e.g., genetic algorithms) in preference to random search, it remains an open question whether the benefits can actually be observed in practice when generating unit test suites for object-oriented classes. In this paper, we report an empirical study on the effects of using a genetic algorithm (GA) to generate test suites over generating test suites incrementally with random search, by applying the EvoSuite unit test suite generator to 1,000 classes randomly selected from the SF110 corpus of open source projects. Surprisingly, the results show little difference between the coverage achieved by test suites generated with evolutionary search compared to those generated using random search. A detailed analysis reveals that the genetic algorithm covers more branches of the type where standard fitness functions provide guidance. In practice, however, we observed that the vast majority of branches in the analyzed projects provide no such guidance.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

Keywords. Automated Unit Test Generation; Random Testing; Genetic Algorithms; Search-Based Testing

1. INTRODUCTION

Automatically generating software test cases is an important task with the objective of improving software quality. Many different algorithms and techniques for different types of software testing problems have been proposed. One particular application area where search-based techniques have been successfully applied is the unit testing of object-oriented programs, where test cases are sequences of object constructor and method calls. There are various search-based tools available for languages such as Java and .NET, ranging from tools based on random search such as Randoop [21], JCrasher [5], JTExpert [23], NightHawk [2], T3 [22], or Yeti-Test [20], to tools based on evolutionary search such as EvoSuite [7], eToc [25] or Testful [4].

Although the tools based on evolutionary search techniques are commonly thought to be superior, it is unclear whether this is actually the case in practice. It could be that differences in performance across tools may be accounted for by the differences in the programming language that they target, or in the way they have been engineered, as opposed to any specific benefits of the particular search algorithm that they apply. In order to shed more light on these questions, in this paper we report on experiments to contrast the use of a genetic algorithm (GA) to optimize unit test suites for code coverage with an algorithm that optimizes code coverage by adding random tests to a test suite. Specifically, we make the following contributions in this study:

1. We report on the effectiveness of using a GA compared to an algorithm based on random search for the purpose of generating test suites with high branch coverage for object-oriented programs.

2. We investigate the influence of certain types of branches within the classes under test on the performance of each technique.

3. Finally, in order to better understand the effect of the search budget on the performance of each technique, we study the techniques' effectiveness at each time step during the search process.

To allow for a fair comparison, we use the GA and a common version of random test generation implemented in the same tool – EvoSuite, which generates branch covering test suites for Java classes. We run experiments on a random sample of 1,000 classes from the SF110 corpus of open source projects [11] and evaluate both techniques in terms of the achieved code coverage. Our results suggest that, in practice, there is little difference between the use of the GA and random search. While the two approaches have different performance profiles over time, the main reason for this finding is actually because of the types of branches that are prevalent in object-oriented programs. Fitness-guided searches like GAs work well when trying to cover branches that result in a smooth gradient of fitness values, which the search can "follow" to the required test case. These branches are typically characterized by numerical comparisons. However, our study found that in practice such "gradient branches" are relatively few in number; allowing random search to generate test cases without much relative disadvantage, and with a similar level of effectiveness.

GECCO '15, July 11 - 15, 2015, Madrid, Spain

ISBN 978-1-4503-3472-3/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739480.2754696

2. SEARCH-BASED TEST GENERATION

In this paper, we study the application of both random and GA search to automatic test suite generation, as implemented in the EvoSuite tool. EvoSuite aims to generate unit test suites that cover as many branches of a Java class as possible, while also executing all methods that are devoid of branches, referred to as "branchless" methods.

2.1 Random Search for Tests

One strategy for finding branch-covering test cases is simply to generate sequences of statements to the class under test at random, coupled with randomly-generated inputs. If a randomly-generated test case covers new branches that have not been executed before, it is added to a test suite for the class, else it can be discarded. One disadvantage of this approach is the size of the resulting test suite, which can be very large and therefore carry a high execution cost.

A further problem is finding inputs that need to be certain "magic" values required to execute certain branches, such as constant values, specific strings etc. that are unlikely to be generated fortuitously. One way of circumventing this problem is to enhance the algorithm through *seeding*.

2.1.1 Seeding

The process of *seeding* involves biasing the search process towards certain input values that are likely to improve the chances of executing more coverage goals [1,8,19]. EvoSuite obtains seeds both statically and dynamically (as documented by Fraser and Arcuri [8]). The static approach takes place before test generation: EvoSuite collects all literal primitive and string values that appear in the bytecode of the class of the test. Then, while tests are being generated, literals that are encountered at runtime may also be dynamically added to the pool of seeds. Some of these seeds are specially computed, according to a set of predefined rules. For instance, if the test case includes the statement "`foo.startsWith(bar)`", involving the strings `foo` and `bar`, the concatenation `bar + foo` will be added to the seed pool. During the search process, EvoSuite will then choose to use a seed from the pool instead of generating a fresh value, according to some probability.

We study random search with and without seeding enabled in this paper. We refer to the enhanced version of random search incorporating seeding as *Random+*, and the basic implementation without seeding as *Pure Random*.

2.2 Genetic Algorithm Search for Test Suites

While random search relies on encountering solutions by chance, guided searches aim to find solutions more directly by using a problem-specific "fitness function". A fitness function scores better potential solutions to the problem with better fitness values. A good fitness function will provide a gradient of fitness values so that the search can follow a "path" to increasingly better solutions that are increasingly fit for purpose. With a good fitness function, guided search-based approaches are capable of finding suitable solutions in extremely large or infinite search spaces (such as the space of possible test cases for a class as considered in this paper).

Genetic Algorithms (GAs) are one example of a directed search technique that uses simulated natural evolution as a search strategy. GAs evolve solutions to a problem based on their fitness. GAs evolve several candidate solutions at once in a "population". The initial population of candidate solutions is generated randomly. Each iteration of the algorithm seeks to adapt these solutions to ones with an increased fitness: "Crossover" works to splice two solutions to form new "offspring" while "mutation" randomly changes a component of a solution. The new solutions generated are taken forward to the next iteration depending on their fitness.

With EvoSuite's GA, a "solution" is a whole test suite, consisting of a series of test cases [10]. Whenever a new test case is generated at random (as with the construction of the initial population), it is done so as described in the last section, with seeding enabled. Crossover involves recombining test cases across two test suites while mutation works at two levels: at test case level and the test suite level. At the test case level, the mutation operator either randomly adds new statements, removes existing ones, or modifies them and their parameters. At the test suite level, it adds a fresh, randomly-generated test case to an existing test suite.

To guide the search towards achieving a high coverage test suite, the fitness value can be calculated based on the number of covered goals. However, a fitness function based solely on the number of *covered* goals provides no guidance to goals that remain *uncovered*. As with previous works in search-based test generation [18], EvoSuite incorporates branch distance metrics, which indicate how "far" a branch is from being executed. For example, if a conditional "`if (a == b)`" is to be executed as true, the "raw" distance can be computed as "$|\texttt{a} - \texttt{b}|$". In this way, the closer the values of `a` and `b` are to one another, the lower the branch distance is, and the closer the search is to covering the goal.

Since EvoSuite aims to evolve test suites where each test case covers as many branches as possible, the fitness function involves adding the distance value $d(b,T)$ for each branch b within a test suite T, computed as follows [10]:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

where $d_{min}(b,T)$ is the minimum raw distance value for the b for T, and ν is a function that normalizes a distance value between 0 and 1. Since the test suite must cover both the true and false outcomes of each individual branch, a distance value is not computed until the conditional is executed twice by the test suite. This is so that the initial execution of the predicate, with some specific true/false evaluation, is not lost in the process of pursuing the alternative outcome.

As longer test suites require more memory and execution time, controlling the length of the test suite can improve search performance [9]. Therefore, when deciding which test suites should proceed into the population for the next iteration of the search, EvoSuite prefers shorter test suites to test suites with the same fitness but whose test cases of composed of cumulatively higher number of statements.

Java programs are compiled to bytecode in order to be executed by the Java Virtual Machine (JVM), and it is at the level of the bytecode at which EvoSuite works – branch distances are computed by instrumenting and monitoring bytecode instructions. Different types of bytecode instruction can therefore give rise to different types of fitness landscape that may or may not be useful in guiding the search, as we discuss in the next section.

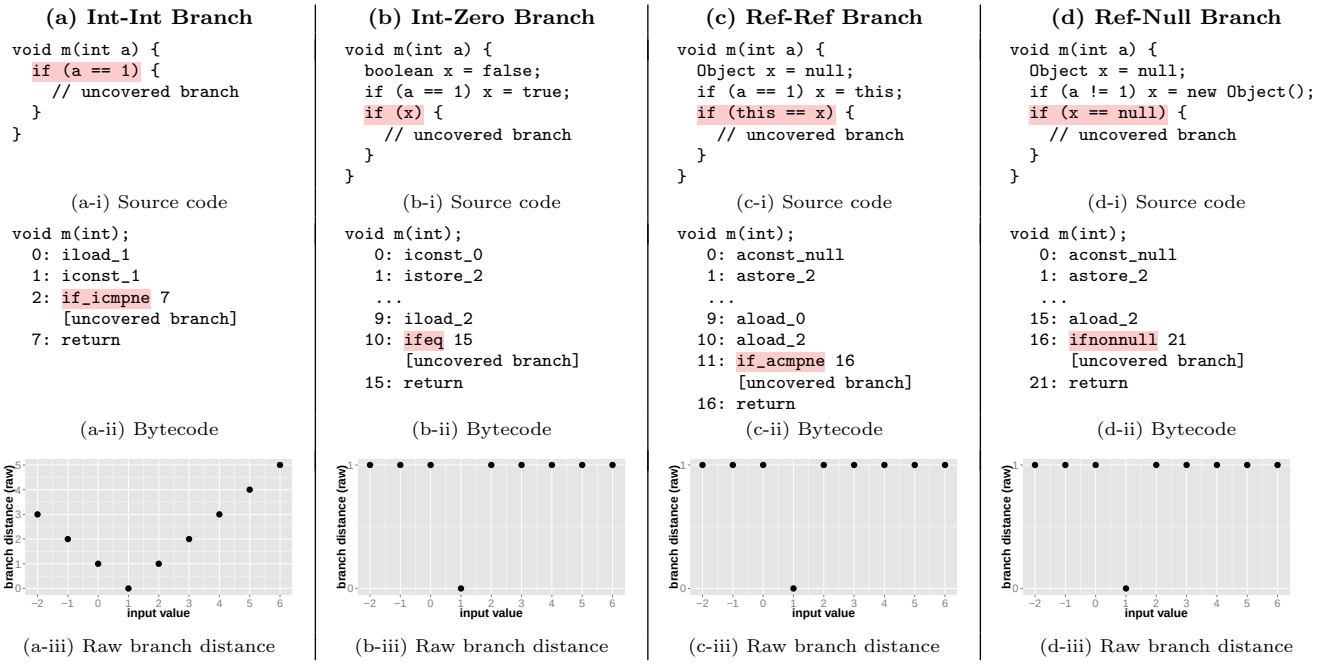


Figure 1: Examples of different branch types (denoted "uncovered branch") and their effect on the respective fitness landscape for the GA through raw (unnormalized) branch distance values. We show both the original Java source and the compiled bytecode, as processed by EvoSuite. Note that the target true/false evaluation of the branches is inverted by the Java compiler. The first column gives an example of a "gradient" branch, providing true guidance to the search. Conversely, the remaining examples do not provide good guidance, with the majority of inputs to the method resulting in the same distance value, and consequently a fitness landscape that is flat other than for the value required to execute the branch of concern.

3. BRANCH TYPES IN JAVA BYTECODE

Given that the fitness function is one of the key differences between the GA and random search, and that a major component of the fitness function is the calculation of distances for the branches in the class under test, we now classify the types of branches that occur in the bytecode of Java programs, and discuss the level of guidance they can potentially afford the GA search in EvoSuite.

This is important because it has been long known that not all branch predicates give "good" guidance, the archetypal example being that involving the boolean flag [3, 14]. Boolean conditions in branch predicates can only ever evaluate to true or false, offering one of only two distance values. Since one of these values corresponds to execution of the branch, no guidance is given to the search. Nevertheless, several branch predicates do indeed provide guidance, and result in a smooth "gradient" in the fitness landscape that a guided search can use to easily find test inputs.

3.1 "Integer-Integer" Branches

"Integer-Integer" branches involve the comparison of two integer values. The range of values possible for this comparison can potentially create a gradient for the search. Figure 1a shows an example of such a comparison, in which a method receives an integer parameter "`a`", and has a conditional statement on the parameter ("`a == 1`") (part a-i of the figure). The bytecode (part a-ii) shows this is compiled to a "`if_icmpne`" instruction, which compares the last two integers pushed to the stack, performing a jump to some other instruction in the bytecode if those two integers are not equal. Part a-iii of the figure shows how the distance value decreases as the chosen input value gets closer to the value that would execute the uncovered branch.

Of course, "Integer-Integer" branches may not always result in a gradient: it depends on the underlying program. One example of this is where two boolean values are compared, since boolean values are represented as the integer values 0 and 1 in Java bytecode. Therefore, source code comparisons involving two boolean values are compiled to an integer comparison involving the usage of the `if_icmpne` instruction. However, and as already discussed, boolean conditions do not induce any useful landscape gradient.

Furthermore, EvoSuite's special handling of `switch` statements falls into the "Integer-Integer" category. Java `switch` statements are compiled to either a `tableswitch` or `lookupswitch` bytecode instruction. These instructions pop the top of the stack to look up a "jump" target instruction in a map data structure, for which the keys are the values originally used in each `case` of the `switch`. For ease of fitness computation, EvoSuite simply instruments the bytecode by adding an explicit `if_icmpeq` for each case before the original `tableswitch` or `lookupswitch` instruction, comparing the top of the stack to each `case` value.

3.2 "Integer-Zero" Branches

"Integer-Zero" branches involve the comparison of an integer value with zero. One type of "Integer-Zero" branch occurs when boolean predicates are evaluated[1], for example as shown by Figure 1b. Here the branch involves the evaluation of the boolean value `x` (part b-i of the figure).

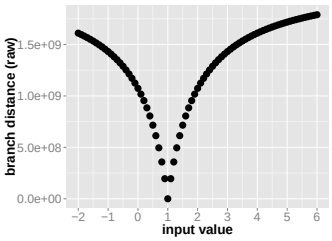
[1]Note that boolean predicate evaluations in branches differ in bytecode from comparing two boolean values – the latter type of branch falls into the "Integer-Integer" category.

```
void m(double a) {
  if (a == 1.0) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(double);
   0: dload_1
   1: dconst_1
   2: dcmpl
   3: ifne 10
   [uncovered branch]
  10: return
```

(ii) Bytecode



(iii) Raw branch distance

Figure 2: An example of handling a `double` comparison, showing the source code (i) and the bytecode (ii). Although these branches fall into the "Int-Zero" category, EvoSuite instruments the bytecode so that distances are recovered, resulting in a gradient landscape (iii).

The corresponding bytecode evaluates `x`, pushing the result (an integer, `0` or `1`) to the stack. The `ifeq` bytecode instruction then pops this value, performing a jump if it is zero. Such a condition can only be either true or false, and as such can only have one of two distance values, which, as shown by Figure 1b-iii, are not useful to guiding the GA to covering the branch. The "right" input must therefore be discovered purely by chance.

A further type of "Integer-Zero" branch occurs as result of comparisons involving values of `float`, `double` and `long` primitive Java types. Figure 2 shows an example of a double comparison. The original source (part i of the figure) performs the comparison in the branch predicate. This is decomposed into a sequence of bytecode instructions shown by part ii of the figure. The comparison is performed by the `dcmpl` in relation to the top two double values pushed to the stack. The `dcmpl` instruction pushes an integer to the stack: `-1` if the first value is greater than the second, `1` if the first is less than the second, else `0` if they are equal. The `ifne` then performs a jump if the top of the stack is not `0`.

Since the original numerical comparison in the source code is transformed to a boolean comparison in the bytecode, a significant amount of useful distance information is "lost" in the compilation process that would have been useful in guiding the search. EvoSuite therefore instruments the bytecode so that distance information can be recovered. The branch distance plot for the example, shown by Figure 2, therefore restores a gradient that can be used to optimize input values towards execution of the uncovered branch.

3.3 "Reference-Reference" branches

"Reference-Reference" branches are where two object references are compared for equality. Since references are not ordinal types, no meaningful distance metric can be applied, and the situation is similar to boolean flags – either the references are the same or they are not. Figure 1c shows an example of this. The original source code conditional is "`if (this == x)`" (part c-i of the figure), which Java compiles to the bytecode instructions 9–11 in part c-ii of the figure. Instructions 9 and 10 push the references onto the stack. Instruction 11 is the branching point in the bytecode, with "`if_acmpne`" popping the top two stack references and performing a jump if they are not equal. The resulting plot of branch distances (part c-iii) shows the resulting plateau, providing no guidance to the required input that makes the references equal and executes the uncovered branch.

3.4 "Reference-Null" branches

"Reference-Null" branches are similar to "Reference-Reference" branches, except one side of the comparison is `null`. Again, no meaningful distance metric can be applied. Figure 1d shows an example. The source code compares `x` with `null`. In the bytecode, `x` is pushed onto the stack by instruction 15. Instruction 16 is the branching point, where the `ifnonnull` instruction performing the jump if the element popped off the top of the stack is not `null`.

3.5 Summary

We have summarized and classified the different types of branches that can occur in Java bytecode. Some of these instructions will potentially give rise to a "gradient" in the fitness landscape, while others will not. We now study the prevalence of these types of branches in real-world code, whether they potentially involve a gradient, and their potential impact on the relative performance of random search and fitness-guided GA search.

4. EXPERIMENTAL SETUP

We designed an empirical study to test the relative effectiveness of test case generation using random and GA search, with the aim of answering the following research questions:

RQ1: Is the use of GA search more effective at generating unit tests than random search?

RQ2: How do the results of the comparison depend on the types of branches found in the code under test?

RQ3: How do the results of the comparison depend on the time allowed for the search?

To answer the research questions, we performed experiments on an initial sample of classes from SourceForge.

4.1 Subjects

In order to compare and contrast the relative effectiveness and performance of random and GA search, we selected a sample of classes from the SF110 corpus of open source projects [11]. The SF110 corpus is made up of 110 open source projects from the SourceForge open source repository (http://sourceforge.net), where 10 of the projects were the most popular by download at the time at which the corpus was constructed (June 2014) and the remaining 100 projects selected at random. Due to the large variation in the number of classes available in each project, we stratified our random sampling over the 110 projects, such that our sample involved at least one class from each of the 110 projects in the corpus, and comprised 1000 classes in total. However, 22 classes were removed from the sample for reasons such as not having any testable methods (e.g., they consisted purely of enumerated types, or did not have any public methods) or test suites could not be generated for some other reason that would allow us to sensibly compare the techniques (e.g., the class contained a bug or other issue that meant it could not be loaded independently without causing an exception).

The final number of classes in the study therefore totaled 978, comprising small classes with just a single coverage goal to larger classes with over 1,000 coverage goals, as shown by Table 1. In this table, *Branchless Methods* indicates the number of methods without conditional statements that can be covered by simply calling the method concerned.

Table 1: Statistics for the sample of 978 classes.

	Min	Avg	Max	SD
Total Branches	0	26.91	1,020	79.2
Branchless Methods	0	7.2	155	11.4
Total Goals	1	34.1	1,030	84.0

4.2 Collation of Branch Type Statistics

So that we could answer RQ2, we collated a series of statistics on the types of branches in the bytecode of each class.

Firstly, we simply collected the numbers of branches that fall into each of the categories detailed in Section 3 (i.e., "Integer-Integer" etc.) by simply analyzing the bytecode of each class in turn.

Secondly, we attempted to classify each branch as either *potentially* having a gradient distance landscape ("Gradient Branches"'), or, a plateau landscape ("Plateau Branches"). We programmed EvoSuite so that during test suite generation it would monitor the distance value of the predicate leading to the branch. If in any of the searches in the experiments, a value other than 0 or 1 is observed, we assume a wider range of distance values is available for fitness computation and label the branch as a "Gradient Branch". Otherwise the search is labelled as a "Plateau Branch". Clearly, this analysis is only indicative (but helps in understanding our results, as we will show in the answer to RQ2). This is because a range of values does not necessarily imply a gradient that will be useful for guiding the search. Nor does only finding the distances 0 and 1 for a branch mean that there are not further distance values that could be encountered. For instance, given a branch predicate $x > 5$, if for the whole duration of the search only the values $x \in \{4, 5, 6\}$ are used, then this will result in the distance values of 1, 0, and 1 respectively; and the branch will be incorrectly classified as a plateau branch. However, it is quite unlikely that the branch would only be attempted with these values over the course of several searches.

4.3 Experimental Procedure

We applied EvoSuite to conduct our experiments, with implementations of the *GA*, *Random+* and *Pure Random* as described in Section 2. We used *Pure Random* in RQ1 only, in order to analyze for possible effects with *Random+* due to its seeding mechanism.

For RQ1 we applied each technique with a search time of two minutes (which has been shown to be a suitable stopping condition in previous work [11]). To answer RQ2, we investigated the influence of the type of conditional predicates on the outcome of each technique. To do so, we used the statistics on branch types, collected as we described in the last section. To better understand the influence of the search budget over the outcome of the techniques for RQ3, we executed EvoSuite using the *GA* and *Random+* configurations with an increased search time of ten minutes and measured the level of coverage at one minute intervals.

We conducted the University of Sheffield's HPC Cluster (http://www.shef.ac.uk/wrgrid/iceberg). Each node has a Sandy-bridge Intel Xeon processor with 3GB real and 6GB virtual memory. We used EvoSuite's default configuration and ran it under Oracle's JDK 7u55.

4.4 Threats to Validity

Threats to the *internal validity* of our study include its usage of only one test generation tool (EvoSuite). While this was deliberate to facilitate a more controlled, fair comparison, it is plausible that specific implementation choices made in EvoSuite may limit the extent to which our results generalize (an associated external threat). The size of the test suites, for example, may influence the comparison; whereas *Random+* has no constraint in the test suite size, the GA evolves test suites with limited size (100 test cases by default) which imposes boundaries in the search space.

Another threat to internal validity stems from the branch-classification analysis described in Section 4.2, which can miscategorize branches in certain cases. While we acknowledge the results of this analysis may only be approximate, however during testing the analysis categorized all branches correctly. Furthermore, chance can affect the results of randomized search algorithms. To mitigate this threat, we repeated all experiments 50 times.

Threats to *external validity* affect the generalization of our results. While we used a randomly selected sample of Java classes as subjects, our results may not generalize beyond the SourceForge project repository or to other programming languages/paradigms. Furthermore, we also used branch coverage as a proxy measure of the quality of the resulting test suites: results may vary for other test suite properties (e.g., size, length or fault detection ability).

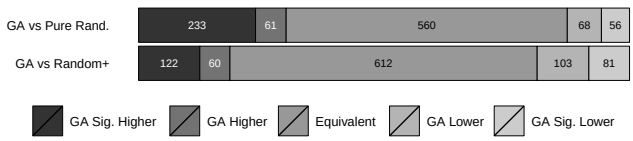
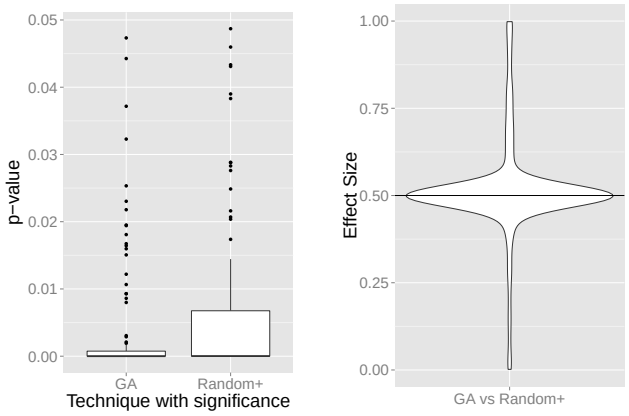


Figure 3: Comparing *GA* performance with *Pure Random* and *Random+* over the 978 SourceForge classes.
("GA Sig. Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 50 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same, etc.)

5. RESULTS

RQ1: Coverage Effectiveness. On average, *GA* attains 67.84% branch coverage, *Random+* achieves 67.94%, while *Pure Random* obtains 65.22%. Figure 3 summarizes the number of classes for which *GA* achieved a significantly higher or lower level of coverage than *Pure Random* and *Random+* over the 50 repetitions of the experiments. We computed significance using the Mann-Whitney U test at a level of $\alpha = 0.05$. While there are 122 classes for which the *GA* achieves significantly higher coverage, there are also 81 classes on which *Random+* attains significantly higher coverage than *GA*. Figure 4a plots the p-values for the significant cases for the *GA* and *Random+* comparison showing that the majority of cases are highly significant (particularly in the *GA* case) and thus unlikely to represent type-I errors.

We observe further similarities in the coverage achieved by *GA* and *Random+* with Figure 4b, which shows effect sizes computed with Vargha-Delaney's $\hat{A}_{12}$ statistic [26]. Here, the effect size estimates the probability that a run of *GA* achieves higher coverage than *Random+*. A value of $\hat{A}_{12} = 0.5$ indicates that both search strategies perform equally, $\hat{A}_{12} = 1$ indicates that all runs of *GA* will achieve higher coverage than *Random+*, and vice versa for $\hat{A}_{12} = 0$. The overall average effect size amounts to 0.51, which indicates that *GA* is only very marginally more effective.



(a) Box Plot (b) Violin Plot

Figure 4: Comparing *GA* performance with *Random+*. (a) Box plot of p-values for classes where a significantly higher level of coverage was achieved with either the *GA* or *Random+*. (b) Violin plot of the effect sizes obtained using Vargha-Delaney's $\hat{A}_{12}$ statistic, here computing the proportion of the 50 repetitions for which the *GA* scores a higher level of coverage than *Random+* for each class; thereby reflecting its relative effectiveness.

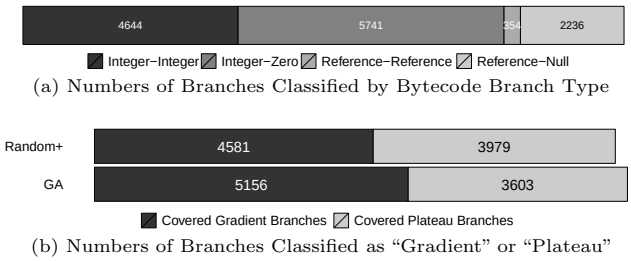
For 612 classes *GA* and *Random+* achieve identical coverage. To a large extent, this can likely be attributed to the simplicity of these classes: *GA* achieves 100% coverage on 456 classes, and *Random+* on 440 classes. Classes with lower but identical coverage are possibly due to problems that EvoSuite cannot overcome regardless of search algorithm (e.g., due to environmental factors such as classes depending on databases or web services that were not available during the experiments).

The comparison between *GA* and *Pure Random* shows larger differences, with 233 classes where *GA* achieves significantly higher coverage. This indicates that optimizations such as constant and dynamic seeding which are used in *Random+* are effective and help covering non-trivial classes.

RQ1. Our experiments showed no significant difference between *GA* and *Random+* in 78% of the classes, and only slightly more classes with increase for *GA* over *Random+* than classes with a decrease.

RQ2: Influence of Branch Types. Although the comparison between *GA* and *Random+* showed 612 classes with no difference in coverage, there were also 203 classes with significant differences. RQ2 aims to shed light on this by studying the influence of different types of branches in a class on the effectiveness of the search algorithms.

Figure 5a shows the distribution of different branch types as taken from the bytecode of the classes. In total, there are 11,677 branches in the 987 classes. "Reference-Reference" branches are rare: this is not surprising as in most cases in Java a comparison is performed using the `equals` method on the objects, rather than comparing references. "Reference-Null" comparisons are more common accounting for approximately one quarter of the branches. Almost half of the branches (5,741) are "Integer-Zero" branches, from which only 303 involve `double`, `float` or `long` comparisons. Only these 303 branches, along with the 3,346 "Integer-Integer" branches have the potential to provide gradients.



(a) Numbers of Branches Classified by Bytecode Branch Type

(b) Numbers of Branches Classified as "Gradient" or "Plateau"

Figure 5: Numbers of different branch types in the classes under test.

Effectiveness on Gradient Branches. Intuitively, one would expect that the *GA* should achieve higher coverage on gradient branches, as the branch distance values will influence the search operators and guide the search towards covering additional branches. Figure 6a compares *GA* and *Random+* in terms of the coverage achieved when only considering gradient branches; that is, the coverage is only calculated for classes that have at least one gradient branch, and the coverage values exclude non-gradient branches. There are 105 classes where *GA* achieves significantly higher coverage of the gradient branches, with only eight classes where the coverage is significantly lower. Figure 5b shows that overall the *GA* covered 5,156 gradient branches, whereas *Random+* covered only 4,581. This confirms that the *GA* benefits from the branch distances provided by the gradient branches.

The eight cases where *Random+* has significantly higher coverage can be explained by their large number of branches (69 total goals and 18 gradient branches on average): The fitness function that guides the *GA* considers all branches at the same time; this means that a test suite that is close to covering many gradient branches may have a better fitness value than a test suite that fully covers fewer branches. In these cases, the *GA* would simply require more time to eventually fully cover all these branches.

Effectiveness on Plateau Branches. Figure 6b compares *GA* and *Random+* when only considering the coverage of plateau branches. There are 158 classes in which the *GA* has significantly lower coverage compared to *Random+*, and 70 classes with significantly higher coverage. Figure 5b shows that overall the *GA* covered 3,603 plateau branches, whereas *Random+* covered 3,979; that is, even though the *GA* covered more branches overall, it covered fewer plateau branches. Since the branch distance for these branches only has two values there is no guidance that the *GA* could exploit – a plateau branch is either covered or it is not covered. The lower coverage of the *GA* can be attributed to a loss of diversity: Over time, the *GA* in EvoSuite prefers smaller test suites and gets rid of random "noise", focusing the search operators on the exploitation of achieved coverage and mutating existing objects. On the other hand, *Random+* is not biased by the search and continuously creates independent new objects and call sequences.

Effectiveness on Branchless Methods. Branchless methods represent a special case similar to plateau branches, and intuitively they are simple to cover – they just require test cases to call the method. Figure 6c compares *GA* and *Random+* with respect to the coverage of methods. Although *GA* achieves significantly higher cover than *Random+* in 18 cases, there are 75 classes where the *GA* results in lower coverage, which is similar in proportions to the plateau

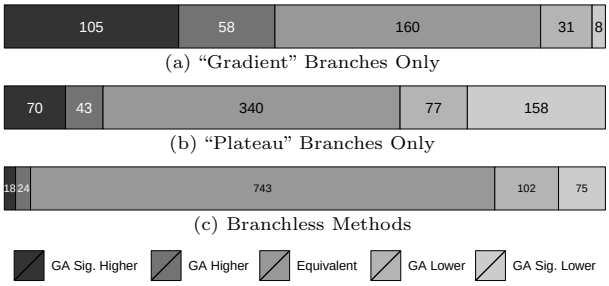


Figure 6: Comparing *GA* performance with *Random+* for different types of branch and with branchless methods. ("GA Sig. Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 50 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same, etc.)

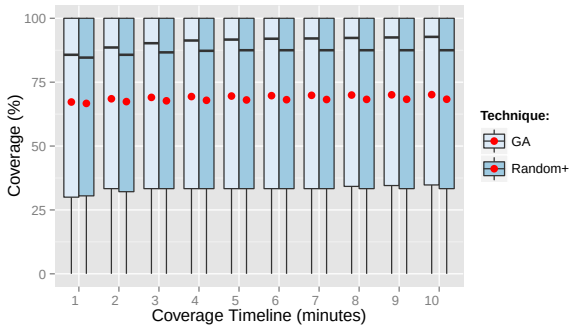


Figure 7: Branch coverage comparison between *GA* vs. *Random+* over 10 minutes with one minute intervals. Dots represent mean averages.

branches. It is maybe surprising that there can be difference in so simple coverage goals in the first place. Our conjecture is that this is because *Random+* has a higher probability of inserting new method calls: The *GA* only mutates a test suite with a certain probability, and then each test in turn is only mutated with a certain probability, and finally insertion of new statements again does not always happen. In contrast, *Random+* generates tests by repeatedly adding new statements. Again it would only be a matter of time for the *GA* to fully cover all branchless methods, although possibly more time than for *Random+*. Interestingly, classes on which the *GA* achieved more than 90% coverage have a median proportion of 100% branchless methods out of all coverage goals, providing further evidence that many classes in practice are trivial.

RQ2. Our experiments show that *GA* achieves higher coverage of gradient branches compared to *Random+*, but lower coverage of plateau branches, which constitute the majority of branches.

RQ3: Effect of the Time Allowed For the Search. The results so far have shown that *GA* and *Random+* perform similarly for the majority of classes after two minutes of search, with some differences in performance on plateau and gradient branches. This raises the question whether the results are influenced by the allocated search budget – given more time, do the results change?

To analyze the impact of the search budget, we repeated the experiments with *GA* and *Random+* using a search budget of 10 minutes, and measured the coverage values at one minute intervals. Figure 7 compares the average coverage per class for each interval: There is a slight increase of coverage for both *GA* and *Random+* over time, and after 10 minutes *GA* achieves an overall average of 70% branch coverage, while *Random+* achieves 68.3%.

Figure 8: Branch coverage comparison between *GA* and *Random+* using a search budget of 10 minutes. (Legend is as for Figure 6.)

Given more time, *GA* will catch up on branchless methods and plateau branches covered compared to *Random+*. Figure 8 compares *GA* with *Random+* after 10 minutes, and shows that the *GA* has significantly lower coverage on only 41 classes after 10 minutes, compared to 81 after two minutes (Note that the number of classes with coverage data after 10 minutes is only 955, as there were 32 additional classes for which EvoSuite did not produce any data after 10 minutes). The *GA* will also continue to optimize gradient branches; however, the dynamic seeding used in EvoSuite will also help *Random+* in many cases to cover gradient branches. Figure 8 shows that there are 127 classes where *GA* has higher coverage after 10 minutes, compared to 122 after two minutes. For 699 classes the coverage is identical, which is likely because the maximum achievable level of coverage has been reached by both algorithms.

RQ3. Our experiments show a slightly higher increase of coverage over time for *GA* compared to *Random+*, but equivalent coverage in the majority of cases.

6. DISCUSSION

Our results indicate that, while the techniques have a similar outcome for the majority of classes, there are differences that influence the effectiveness. However, the fact that *Random+* can outperform the *GA* in a number of subjects raises the question of why this happens in practice. In this section we look at some of the factors in the search algorithms that may be the cause of these surprising results.

Although *GA* and *Random+* were applied with the same time limit in our experiments, *Random+* executed 1.3 times as many statements as the *GA* in the same time. This can be attributed to the search operators of the *GA*, and the resulting frequent copy operations on test cases and cached execution traces as implemented in EvoSuite. Without such an overhead, *Random+* can spend more time executing tests and exploring the search landscape. Nevertheless, as shown in Figure 7 even increasing the *GA*'s budget by 30% to match the number of executed statements can only result in small improvements, and does not affect the overall findings.

The analysis of RQ2 also suggests that the search operators of the *GA* have an effect on the diversity: The *GA* has a lower probability of generating new tests, and may thus be slower at covering plateau branches or branchless methods (cf. Figure 6, part b and c).

A further influencing factor is that a test suite produced by the *GA* may not cover all branches that were covered throughout the search. This is because the fitness function aims to maximize coverage: For instance, given a test suite T_1 that covers goals $\{A, B\}$, and another test suite T_2 that covers goals $\{B, C, D\}$, assuming T_2 has a better fitness

value it will be selected as the best solution. As a result, although goal A was covered by T_1, it remains uncovered in the resulting test suite. In contrast, *Random+* generates a new test case on each iteration, and if the new test covers any new goals, it is added to the test suite. This suggests that creating an archive of solutions that cover new coverage goals would be important for the *GA*.

The large number of plateau branches could potentially be reduced by introducing testability transformation [15]; although EvoSuite implements certain transformations (e.g., on floating point numbers or string comparisons) it does not apply a transformation of boolean flags.

7. RELATED WORK

There have been several papers that have compared GAs with random search in the procedural domain (e.g. [16], [27]). This work has found guided search to always outperform random. In general, procedural code tends to consist of larger functions than methods in OO code, and each function tends to involve more parameters. While random search typically covers a large percentage of the branches involved, the GA covers significantly more.

Sharma et al. [24] showed on 13 examples that random testing of OO container classes achieves the same coverage as shape abstraction, a systematic technique specific for container classes. The results of our experiments suggest that in practice, many OO classes are, similarly to container classes, simple in nature and thus well suited for random testing.

Earlier experiments with EvoSuite on the SF100 corpus [11] showed that a large number of classes are either trivially covered, or uncoverable without providing the test generator with additional features (e.g. to handle environmental inputs such as web services or databases). This finding is in line with our results; however, a comparison with Randoop [21] in the same study suggested a large improvement of *GA* over random testing. The results of our experiments suggest that this improvement is largely due to the engineering of the tool rather than the search algorithm; for example, Randoop does not use seeding.

Eler et al. [6] analyzed the SF100 corpus from the point of view of test data generation using dynamic symbolic execution. They also reported the large number of reference comparisons and the challenges of handling those in a constraint solver. They further reported the relatively low number of branches involving integer comparisons, which result in constraints that DSE is typically strong at handling.

8. CONCLUSIONS

In this paper, we presented an empirical study comparing the effectiveness of a GA and a random search-based algorithm for generating branch coverage test suites for real-world Java classes. One might expect the GA to be more suitable than random search for this task, but surprisingly we observed that both algorithms behaved similarly on the majority of classes, in particular when applying optimisations such as seeding. Although a GA can exploit the guidance provided by certain types of branches, in practice there are many more branches that provide no such guidance, and on some classes with many such branches the GA resulted in lower coverage than random search – even when a large search budget was used.

There certainly is some room for improving the GA for the task, as for example suggested by different optimisations not included in our experiments [12, 13, 17]. However, if the objective is to quickly achieve a decent level of branch coverage on object-oriented classes, then using random search with seeding may be sufficient.

Acknowledgments. This work is supported by the EPSRC project "EXOGEN" (EP/K030353/1). The authors would like to thank Chris Wright for help with R.

9. REFERENCES

[1] Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: ASE. IEEE (2011)
[2] Andrews, J.H., Li, F.C., Menzies, T.: Nighthawk: A two-level genetic-random unit test data generator. In: ASE. ACM (2007)
[3] Baresel, A., Sthamer, H.: Evolutionary testing of flag conditions. In: GECCO. Springer (2003)
[4] Baresi, L., Lanzi, P.L., Miraz, M.: Testful: an evolutionary test approach for Java. In: ICST. IEEE (2010)
[5] Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Exper. 34(11) (2004)
[6] Eler, M., Endo, A., Durelli, V.: Quantifying the characteristics of Java programs that may influence symbolic execution from a test data generation perspective. In: COMPSAC. IEEE (2014)
[7] Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: FSE. ACM (2011)
[8] Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: ICST. IEEE (2012)
[9] Fraser, G., Arcuri, A.: Handling test length bloat. Softw. Test., Verif. Reliab. 23(7) (2013)
[10] Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. on Software Engineering 39(2) (2013)
[11] Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite. ACM TOSEM 24(2) (2014)
[12] Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. J. Syst. Software 103(0) (2014)
[13] Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: ISSRE. IEEE (2013)
[14] Harman, M., Hu, L., Hierons, R., Baresel, A., Sthamer, H.: Improving evolutionary testing by flag removal. In: GECCO. MK Pub. (2002)
[15] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. on Software Engineering 30(1) (2004)
[16] Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. on Software Engineering 36(2) (2010)
[17] Li, Y., Fraser, G.: Bytecode testability transformation. In: SSBSE, pp. 237–251. Springer (2011)
[18] McMinn, P.: Search-based software test data generation: a survey. Softw. Test., Verif. Reliab. 14(2) (2004)
[19] McMinn, P., Shahbaz, M., Stevenson, M.: Search-based test input generation for string data types using the results of web queries. In: ICST. IEEE (2012)
[20] Oriol, M., Tassis, S.: Testing .NET code with YETI. In: ICECCS. IEEE (2010)
[21] Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: OOPSLA. ACM (2007)
[22] Prasetya, I.W.B.: T3, a combinator-based random testing tool for Java: benchmarking. In: FITTEST'13. Springer (2014)
[23] Sakti, A., Pesant, G., Gueheneuc, Y.G.: Instance generator and problem representation to improve object oriented code coverage. IEEE Trans. on Software Engineering 41(3) (2015)
[24] Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., Marinov, D.: Testing container classes: Random or systematic? In: FASE. Springer (2011)
[25] Tonella, P.: Evolutionary testing of classes. ACM SIGSOFT Softw. Eng. Notes 29(4) (2004)
[26] Vargha, A., Delaney, H.D.: A critique and improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. Educational and Behavioral Statistics 25(2) (2000)
[27] Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14) (2001)