# Test Suite Generation with Memetic Algorithms

Gordon Fraser
University of Sheffield
Dep. of Computer Science
211 Regent Court, Portobello,
S1 4DP, Sheffield
gordon.fraser@sheffield.ac.uk

Andrea Arcuri
Certus Software V&V Center
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker,
Norway
arcuri@simula.no

Phil McMinn
University of Sheffield
Dep. of Computer Science
211 Regent Court, Portobello,
S1 4DP, Sheffield
p.mcminn@sheffield.ac.uk

## ABSTRACT

Genetic Algorithms have been successfully applied to the generation of unit tests for classes, and are well suited to create complex objects through sequences of method calls. However, because the neighborhood in the search space for method sequences is huge, even supposedly simple optimizations on primitive variables (e.g., numbers and strings) can be ineffective or unsuccessful. To overcome this problem, we extend the global search applied in the EvoSuite test generation tool with local search on the individual statements of method sequences. In contrast to previous work on local search, we also consider complex datatypes including strings and arrays. A rigorous experimental methodology has been applied to properly evaluate these new local search operators. In our experiments on a set of open source classes of different kinds (e.g., numerical applications and text processing), the resulting test data generation technique increased branch coverage by up to 32% on average over the normal Genetic Algorithm.

## Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search;
D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Algorithms, Experimentation, Reliability

## Keywords

EvoSuite, Search-based Software Engineering, Object-oriented, Evolutionary Testing

## 1. INTRODUCTION

Software testing is one of the most important techniques applied to improve software quality. When there is no automated test oracle available, as for example is often the case in white-box testing, test generation techniques aim to produce small test suites with high code coverage, such that these test suites can be analyzed by the developer in feasible time. In the case of object-oriented classes, test cases amount to sequences of method calls, and search-based testing [18] has been demonstrated to be an effective solution as it can handle not only optimizations on primitive datatypes, but also on complex data structures and sequences of method calls. The use of search-based techniques for optimizing entire test suites towards high branch coverage for classes [10] has been shown to be a successful technique for this purpose. Empirical experiments have shown that it is practically usable on a wide range of programs [9].

When an individual of the search is a test suite consisting of a variable number of test cases, each of which is a sequence of method calls of variable length, then the neighborhood in the search space is simply huge: Here, a neighbor is not only defined by the next successive value for primitive types, but also as all the possible calls that can be inserted on existing objects and all the possible changes that can be performed on the sequences of method calls. If somewhere in such a test suite there is an individual primitive value that needs to be optimized, then the probability of it being mutated during the search with a Genetic Algorithm is low, and so the optimization towards the targets dependent on this value can take long. The urgency of this problem becomes even more apparent when one also considers string variables as primitives, where again the neighborhood is huge. Consider

```
class Foo {
  boolean bar(String s) {
    if(s.equals("bar"))
      // target
 }
}
```

```
Foo foo = new Foo();
String s = "test";
foo.bar(s);
```

**Figure 1: Example class and test case: In theory, four edits of s can lead to the target branch being covered. However, with a Genetic Algorithm where each statement of the test is mutated with a certain probability (e.g., 1/3 when there are three statements) one would have to be really lucky: If the test is part of a test suite (size 10) of a Genetic Algorithm (population 50) and we only assume a character range of 128, then even if we ignore all the complexities of Genetic Algorithms, we would still need on average at least $50 \times 4 \times 1/(\frac{1}{10} \times \frac{1}{3} \times \frac{1}{128}) = 768{,}000$ fitness evaluations before covering the target branch.**


*GECCO'13,* July 6–10, 2013, Amsterdam, The Netherlands.

the example test case in Figure 1: Even under very strong simplifications, with a "basic" Genetic Algorithm we would need an average of at least 768,000 costly fitness evaluations (i.e., test executions) to cover the target branch. If the budget is limited, then indeed the approach might fail to cover such goals.

To overcome this problem, we extend the Genetic Algorithm used in the whole test suite generation approach to a Memetic Algorithm: At regular intervals, the search inspects the primitive variables and tries to apply local search to improve them. Although these extensions are intuitively useful and tempting, they add additional parameters to the already large parameter space. In fact, misusing these techniques can even lead to worse results, and so we conducted a detailed study to find the best parameter settings. In detail, the contributions of this paper are:

**Memetic algorithm for test suite optimization:** We present a novel approach to integrate local search on primitive values in a global search for test suites.

**Local search for complex values:** We extend the notion of local search as commonly performed on numerical inputs to string inputs, arrays, and objects.

**Sensitivity analysis:** We have implemented the approach as an extension to the EvoSuite tool [10], and analyze the effects of the different parameters involved in the local search, and determine the best configuration.

**Evaluation:** We evaluate our approach on a set of 16 open source classes, and compare the results to the standard search-based approach that does not include local search.

# 2. BACKGROUND

Search-based testing applies meta-heuristic search techniques to the task of test data generation [18].

## 2.1 Local Search Algorithms

With *local* search algorithms [2] one only considers the neighborhood of a candidate solution. For example, a hill climbing search is usually started with a random solution, of which all neighbors are evaluated with respect to their fitness for the search objective. The search then continues on either the first neighbor that has improved the fitness, or the best neighbor, and again considers its neighborhood. The search can easily get stuck in local optima, which is typically overcome by restarting the search with new random values. A popular version of hill climbing often used in test data generation is Korel's Alternating Variable Method [7, 16].

The Alternative Variable Method (AVM) is a local search technique similar to hill climbing, and was introduced by Korel [16]. The AVM considers each input variable of an optimization function in isolation, and tries to optimize it locally. Initially, variables are set to random values. Then, AVM starts with exploratory moves on the first variable. For example, in the case of an integer an exploratory move consists of adding a delta of $+1$ or $-1$. If the exploratory move was successful (i.e., the fitness improved), then the search accelerates movement with pattern moves. For example, in the case of an integer, the search would next try $+2$, then $+4$, etc. Once the next step of the pattern search does not improve the fitness any further, the search goes back to exploratory moves on this variable. If successful, pattern search is again applied in the direction of the exploratory move. Once no further optimization of the variable is possible, the search moves on to the next variable. If no variable can be improved the search restarts at another randomly chosen location to overcome local optima.

## 2.2 Global Search Algorithms

In contrast, *global* search algorithms try to overcome local optima in order to find more globally optimal solutions. Harman and McMinn [14] recently determined that global search is more effective than local search, but less efficient, as it is more costly.

With evolutionary testing, one of the most commonly applied global search algorithms is a *Genetic Algorithm* (GA). A GA tries to imitate the natural processes of evolution: An initial population of usually randomly produced candidate solutions is evolved using search operators that resemble natural processes. Selection of parents for reproduction is based on their fitness (survival of the fittest). Reproduction is performed using crossover and mutation with certain probabilities. With each iteration of the GA, the fitness of the population improves, until either an optimal solution has been found, or some other stopping condition has been met (e.g. maximum time or number of fitness evaluations). In evolutionary testing, the population would for example consist of test cases, and the fitness estimates how close a candidate solution is to satisfying a coverage goal. The initial population is usually generated randomly, i.e., a fixed number of random input values is generated. The operators used in the evolution of this initial population depend on the chosen representation.

A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for branch coverage, a common fitness function [18] integrates the *approach level* (number of unsatisfied control dependencies) and the *branch distance* (estimation of how close the deviating condition is to evaluating as desired). Such search techniques have not only been applied in the context of primitive datatypes, but also to test object-oriented software using method sequences [11, 21].

## 2.3 Memetic Algorithms

A Memetic Algorithm (MA) hybridizes global and local search. The use of MAs for test generation was originally proposed by Wang and Jeng [22] in the context of test generation for procedural code. Arcuri [5] combined a GA with hill climbing to form a MA when generating unit tests for container classes. Harman and McMinn [14] analyzed the effects of global and local search, and concluded that MAs achieve better performance than global search and local search. Baresi et al. [6] also use a hybrid evolutionary search in their TestFul test generation tool, where at the global search level a single test case aims to maximize coverage, while at the local search level the optimization targets individual branch conditions.

# 3. WHOLE TEST SUITE GENERATION

In whole test suite generation, the optimization target is not to produce a test that reaches one particular coverage goal, but it is to produce a complete test suite that maximizes coverage, while minimizing the size at the same time.

## 3.1 Representation

An individual of the search is a *test suite*, which is represented as a set $T$ of test cases $t_i$. Given $|T| = n$, we have

$T = \{t_1, t_2, \ldots, t_n\}$. A test case is a sequence of statements $t = \langle s_1, s_2, \ldots, s_l \rangle$ of length $l$. The length of a test suite is defined as the sum of the lengths of its test cases, i.e., $length(T) = \sum_{t \in T} l_t$.

There are several different types of statements in a test case: *Primitive statements* define primitive values, such as Booleans, integers, or Strings; *Constructor statements* invoke constructors to produce new values; *Method statements* invoke methods on existing objects, using existing objects as parameters; *Field statements* retrieve values from public members of existing objects; *Array statements* define arrays; *Assignment statements* assign values to array indexes or public member fields of existing objects. Each of these statements defines a new variable, with the exception of void method calls and assignment statements. Variables used as parameters of constructor and method calls and as source objects for field assignments or retrievals need to be already defined by the point at which they are used in the sequence.

Crossover of test suites means that offspring recombine subsets from parent test suites. For example, for two selected parents $P_1$ and $P_2$, a random value $\alpha$ is chosen from [0,1], and on one hand, the first offspring $O_1$ will contain the first $\alpha|P_1|$ test cases from the first parent, followed by the last $(1-\alpha)|P_2|$ test cases from the second parent. On the other hand, the second offspring $O_2$ will contain the first $\alpha|P_2|$ test cases from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent.

Mutation of test suites means that test cases are inserted, deleted, or changed. With probability $\sigma$, a test case is added. If it is added, then a second test case is added with probability $\sigma^2$, and so on until the $i$th test case is not added (which happens with probability $1-\sigma^i$). Each test case is changed with probability $1/|T|$. There are many different options to change a test case: One can delete or alter existing statements, or insert new statements. We perform each of these three operations with probability 1/3; on average, only one of them is applied, although with probability $(1/3)^3$ all of them are applied. When removing statements from a test it is important that this operation must ensure that all dependencies are satisfied. Inserting statements into a test case means inserting method calls on existing calls, or adding new calls on the class under test. For details on the mutation operators we refer to [10].

## 3.2 Fitness Function

In this paper, we consider branch coverage as the optimization target, although the approach can be applied to any coverage criterion that can be expressed with a fitness function. Typically, fitness functions for other coverage criteria are based on the branch coverage fitness function. Branch coverage requires that for every conditional statement in the code there is at least one test that makes it evaluate to true, and one that makes it evaluate to false. For this, we can use a standard metric used in search-based testing, the *branch distance*.

For every branch, the branch distance estimates how close that branch was to evaluating to true or to false. For example, if we have the branch $x == 17$, and a concrete test case where $x$ has the value 10, then the branch distance to make this branch true would be $17 - 10 = 7$, while the branch distance to making this branch false is 0. To achieve branch coverage in whole test suite generation, the fitness function tries to optimize the sum of all normalized, minimal branch distances to 0 – if for each branch there exists a test such that the execution leads to a branch distance of 0, then all branches have been covered.

## 3.3 Search Guidance on Strings

The fitness function in whole test suite generation is based on branch distances. EvoSuite works directly on Java bytecode, where except for reference comparisons, the branching instructions are all based on numerical values. Comparisons on strings first map to Boolean values, which are then used in further computations; e.g., a source code branch like `if(string1.equals(string2))` consists of a method call on `String.equals` followed by a comparison of the Boolean return value with `true`. To offer guidance on string based branches we replace calls to the `String.equals` method with a custom method that returns a distance measurement [17]. The branching conditions comparing the Boolean with `true` thus have to be changed to check whether this distance measurement is greater than 0 or not (i.e., `== true` is changed to `== 0`, and `== false` is changed to `> 0`). The distance measurement itself depends on the search operators used; for example, if the search operators support inserts, changes, and deletions, then the Levenshtein distance measurement can be used. This transformation is an instance of *testability transformation* [12], which is commonly applied to improve the guidance offered by the search landscape of programs.

Search operators for string values have initially been proposed by Alshraideh and Bottaci [1]. Based on our distance measurement, when a primitive statement defining a string value is mutated, each of the following is applied with probability 1/3 (i.e., with probability $(1/3)^3$ all are applied):

**Deletion:** Every character in the string is deleted with probability $1/n$, where $n$ is the length of the string. Thus, on average, one character is deleted.

**Change:** Every character in the string is changed with probability $1/n$; if it is changed, then it is replaced with a random character.

**Insertion:** With probability $\alpha = 0.5$, a random character is inserted at a random position $p$ within the string. If a character was inserted, then another character is inserted with probability $\alpha^2$, and so on, until no more characters are inserted.

# 4. APPLYING MEMETIC ALGORITHMS

The whole test suite generation presented in the previous section is a global optimization technique, which means that we are trying to optimize an entire candidate solution towards the global optimum (maximum coverage). Search operations in global search can lead to large jumps in the search space. In contrast, local search explores the immediate neighborhood. For example, if we have a test suite consisting of $X$ test cases of average length $L$, then the probability of mutating one particular primitive value in there is $\frac{1}{X} \times \frac{1}{L}$. However, evolving a primitive value to a target value may require many steps, and so global search can easily exceed the search budget before finding a solution. This is a problem that local search can overcome.

## 4.1 Local Search on Method Call Sequences

The aim of the local search is to optimize the values in one particular test case of a test suite. When local search is applied to a test case, EvoSuite iterates over its sequence of

statements from the last to the first, and for each statement applies a local search dependent on the type of the statement. Local search is performed for the following types of statements: primitive statements, method statements, constructor statements, field statements and array statements.

### 4.1.1 Primitive Statements

**Booleans and Enumerations:** For Boolean variables the only option is to flip the value. For enumerations, an exploratory move consists of replacing the enum value with any other value, and if the exploratory move was successful, we iterate over all enumeration values.

**Integer Datatypes:** For integer variables (which includes all flavors such as `byte`, `short`, `char`, `int`, `long`) the possible exploratory moves are $+1$ and $-1$. The exploratory move decides the direction of the pattern move. If an exploratory move to $+1$ was successful, then with every iteration $I$ of the pattern search we add $\delta = 2^I$ to the variable. If $+1$ was not successful, $-1$ is used as exploratory move, and if successful, subsequently $\delta$ is subtracted.

**Floating Point Datatypes:** For floating point variables (`float`, `double`) we use the same approach as originally defined by Harman and McMinn [13] for handling floating point numbers with the AVM. Exploratory moves are performed for a range of precision values $p$, where the precision ranges from 0–7 for `float` variables, and from 0–15 for `double` values. Exploratory moves are applied using $\delta = 2^I \times dir \times 10^{-p}$. Here $dir$ denotes either $+1$ or $-1$, and $I$ is the number of the iteration, which is 0 during exploratory moves. If an exploratory move was successful, then pattern moves are made by increasing $I$ when calculating $\delta$.

**Strings:** For string variables, exploratory moves are slightly more complicated: To determine if local search on a string variable is necessary, we first apply $n$ random mutations on the string[1]. These mutations are the same as described in Section 3.3. If any of the $n$ probing mutations changed the fitness, then we know that the string has some effect on it, regardless of whether the change resulted in an improvement or not. As discussed in Section 3.3, string values affect the fitness through a range of Boolean conditions that are used in branches; these conditions are transformed such that the branch distance also gives guidance on strings. If the probing on a string showed that it affects the fitness, then we apply a systematic local search on the string. The operations on the string must reflect the distance estimation applied on string comparisons:

**Deletion:** First, every single character is removed and the fitness value is checked. If the fitness did not improve, the character is kept in the string.

**Change:** Second, every single character is replaced with every possible other character; for practical reasons, we restrict the search to ASCII characters. If a replacement is successful, we move to the next character. If a character was not successfully replaced, the original character stays in place.

**Insertion:** Third, we insert new characters. Because the fitness evaluation requires test execution, trying to insert every possible character at every possible position would be too expensive – yet this is what would be required when using the standard Levenshtein distance (edit distance) as distance metric. Consequently, we only attempt to insert characters at the front and the back, and adapt the distance function for strings accordingly.

The distance function for two strings $s1$ and $s2$ used during the search is (c.f. [15]):

$$\begin{aligned}\text{distance}(s1,s2) &= |\text{length}(s1) - \text{length}(s2)| + \\ &\sum_{i=0}^{\min(\text{length}(s1),\text{length}(s2))} \text{distance}(s1[i],s2[i])\end{aligned}$$

### 4.1.2 Array Statements

Local search on arrays concerns the length of an array and the values assigned to the slots of the array. To allow efficient search on the array length, the first step of the local search is to try to remove assignments to array slots. For an array of length $n$, we first try to remove the assignment at slot $n-1$. If the fitness value remains unchanged, we try to remove the assignment at slot $n-2$, and so on, until we find the highest index $n'$ for which an assignment positively contributes to the fitness value. Then, we apply a regular integer-based local search on the length value of the array, making sure the length does not get smaller than $n'+1$. Once the search has found the best length, we expand the test case with assignments to all slots of the array which are not already assigned in the test case (such assignments may be deleted as part of the regular search). Then, on each assignment to the array we perform a local search, depending on the component type of the array.

### 4.1.3 Reference Type Statements

Statements related to reference values (method statement, constructor statement, field statement) do not allow traditional local search in terms of primitive values. The neighborhood of a complex type in a sequence of calls is huge (e.g., all possible calls on an object with all possible parameter combinations, etc.), such that exhaustive search is not a viable option. Therefore, we apply randomized hill climbing on such statements. This local search consists of repeatedly applying random mutations to the statement, and it is stopped if there are $R$ consecutive mutations that did not improve the fitness (in our experiments, $R = 10$).

We use the following mutations for this randomized hill climbing:

- Replace the statement with a random call returning the same type.
- Replace a parameter (for method and constructor statements) or the receiving object (for field and method statements) with any other value of the same type available in the test case.
- If the call creates a non-primitive object, add a random method call on the object after the statement.

The fitness function described in Section 3 requires that every branch is executed twice, such that there is at least one run we can optimize towards executing the branch to one direction, without losing the other direction. If during local search we determine that there is a branch that is only executed once, as a further optimization we duplicate its test case within the test suite.

## 4.2 Memetic Algorithm

Given the ability to perform local search on the individuals of a global optimization there is the question of how

[1]In theory, static analysis could also be used to determine when a string is a data dependency of one of the target branches; however, as the method sequences may use many different classes that are not known ahead of time, this is non-trivial.

**Table 1: Case Study Classes**

"Branches" is the number of branches reported by EvoSuite; "LOC" refers to the number of non-commenting source code lines reported by JavaNCSS (*http://www.kclee.de/clemens/java/javancss*).

| Project | Class | LOC | Branches |
|---|---|---|---|
| Roops | IntArrayWithoutExceptions | 64 | 43 |
| Roops | LinearWithoutOverflow | 223 | 93 |
| Roops | FloatArithmetic | 68 | 49 |
| Roops | IA.WithArrayParameters | 30 | 29 |
| SCS | Cookie | 18 | 13 |
| SCS | DateParse | 32 | 39 |
| SCS | Stemmer | 345 | 344 |
| SCS | Ordered4 | 11 | 29 |
| NanoXML | XMLElement | 661 | 310 |
| Commons CLI | CommandLine | 87 | 45 |
| JDOM | Attribute | 138 | 65 |
| Commons Codec | DoubleMetaphone | 579 | 504 |
| java.util | ArrayList | 151 | 70 |
| NCS | Bessj | 80 | 29 |
| Commons Math | FastFourierTransformer | 290 | 135 |
| Joda Time | DateTimeFormat | 356 | 434 |

to integrate these techniques. Often, MAs are implemented such that individuals can perform Lamarckian or Baldwinian learning immediately after reproduction [20]. This, however, raises the questions of how often to apply the individual learning, on which individuals it should be applied, and how long it should be done. Because local search can be very expensive, we would like to direct the learning towards the better individuals of the population, such that newly generated genetic material is more likely to directly contribute towards the solution.

In EvoSuite, local search is applied at regular intervals; the rate at which it is applied is the first parameter of local search. When local search is applied, we iterate over the population ranked by their fitness, such that the first individual to be improved is the best individual of the search, then the second best, and so on. Thus, as a second parameter, there is a search budget for this local search.

# 5. EVALUATION

The presented techniques depend on a number of parameters, and so evaluation needs to be carefully done with respect to these. We therefore aim to empirically answer the following three research questions:

**RQ1:** Does local search improve the performance of whole test suite generation?

**RQ2:** Which parameter combination gives the best results?

**RQ3:** How do the results vary based on the available search budget?

## 5.1 Experimental Setup

To answer the research questions, we first need to decide for how long and how often to run local search in the MA; these are the two parameters discussed in Section 4.2. Because how often we apply local search depends on the number $X$ of generations, how much local search is actually done is dependent on the population size. Consequently, we also had to consider the population size when designing the experiments. We also considered seeding from bytecode [8] as a further parameter to experiment with, as we expected it to have a large impact on the performance in the cases in which local search is successful (and this is confirmed in the experiments). In total, we had four different parameters to experiment with.

For our evaluation, we used the classes already used in previous experiments [4], but had to exclude those on which EvoSuite trivially achieves 100% coverage. In the choice of a variegated set of classes to experiment with, we tried to strike a balance among the different kinds of classes. To this end, beside classes coming from the case study in [4], we also included four benchmark classes on integer and floating point calculations from the Roops[2] benchmark suite for object-oriented testing, This results in a total of 16 classes, of which some characteristics are given in Table 1.

The use of only 16 classes was necessitated by the complex evaluation setup. In this paper, we study the effects of four different parameters. For population size, local search budget and rate we considered five different values, i.e., {5, 25, 50, 75, 100}, and for seeding two (on/off). We also included further configurations without local search (i.e., the default GA in EvoSuite), but still considering the different combinations of population size and seeding.

On each class, for each parameter, we ran EvoSuite 30 times with different random seeds to take into account their random nature. In each run, the stopping criterion was a 10 minute timeout. Thus, in total the experiments took $((2 \times 5^3) + (2 \times 5)) \times (30 \times 10 \times 16)/(60 \times 24) = 866$ days of computational time, which required the use of a cluster of computers. During these runs, EvoSuite was configured using the optimal configuration determined in our previous experiments on tuning [4].

## 5.2 Results

For both the cases in which seeding was used and not, we analyzed the 125 configurations using MA, and chose the one that resulted with highest average coverage over the 16 classes in the case study. The same is done for the basic GA, i.e., we evaluated which configuration of the population size gave best results. We call these four configurations (two for MA, and two for GA) "tuned". Table 2 shows the comparison between the tuned MA and tuned GA configuration based on whether seeding was used.

To evaluate the statistical and practical differences among the different settings, we followed the guidelines in [3]. Statistical difference is evaluated with a two-tailed Mann-Whitney U-test, whereas the magnitude of improvement is quantified with the Vargha-Delaney standardized effect size $\hat{A}_{12}$.

Results in Table 2 answer **RQ1** by clearly showing, with high statistical confidence, that the MA outperforms the standard GA in many, but not all, cases. For classes such as `Cookie`, improvements are as high as a $87 - 55 = 32\%$ average coverage difference (when seeding is not used).

> **RQ1:** *The MA achieved up to a 32% higher branch coverage than the standard GA.*

One thing that is clearly visible in Table 2 is that seeding, as expected [8], leads to higher results. On one hand, when seeding is not used, the difference in average coverage between the MA and the GA is $86 - 79 = 7\%$. On the other hand, when seeding is used, the difference is $89 - 88 = 1\%$. At a first look, such an improvement might be considered low. But the statistics in Table 2 points out a relatively high average 0.61 effect size, with four classes having a strong statistical difference. This is not in contrast with the 1% difference in the raw values of the achieved coverage. What the

[2]http://code.google.com/p/roops/

**Table 2: Comparison of average coverage obtained by the tuned MA and by the tuned GA. Effect sizes ($\hat{A}_{12}$) with statistically significant difference at 0.05 level are shown in bold. Data are divided based on whether seeding was used or not.**

| Case Study | Without Seeding | | | With Seeding | | |
|---|---|---|---|---|---|---|
| | MA | GA | $\hat{A}_{12}$ | MA | GA | $\hat{A}_{12}$ |
| roops.core.bv32.arr.noex.IntArrayWithoutExceptions | 0.90 | 0.86 | **0.91** | 0.92 | 0.92 | 0.57 |
| roops.core.bv32.linear.noex.gods.LinearWithoutOverflow | 0.98 | 0.74 | **1.00** | 0.98 | 0.92 | **1.00** |
| roops.extended.bv32.floats.FloatArithmetic | 0.65 | 0.49 | **1.00** | 0.86 | 0.86 | 0.50 |
| scs.Cookie | 0.87 | 0.55 | **0.94** | 0.95 | 0.95 | 0.52 |
| scs.DateParse | 0.87 | 0.69 | **1.00** | 1.00 | 1.00 | 0.50 |
| net.n3.nanoxml.XMLElement | 0.98 | 0.98 | 0.52 | 0.98 | 0.98 | 0.54 |
| org.apache.commons.cli.CommandLine | 0.98 | 0.98 | 0.50 | 0.97 | 0.97 | 0.49 |
| org.jdom.Attribute | 0.84 | 0.79 | **0.99** | 0.86 | 0.86 | 0.57 |
| org.apache.commons.codec.language.DoubleMetaphone | 0.75 | 0.71 | **0.98** | 0.84 | 0.82 | **0.76** |
| java2.util2.ArrayList | 0.94 | 0.94 | 0.48 | 0.94 | 0.94 | 0.50 |
| scs.Stemmer | 0.72 | 0.73 | 0.47 | 0.74 | 0.72 | **0.77** |
| ncs.Bessj | 0.97 | 0.96 | **0.57** | 0.97 | 0.97 | 0.50 |
| org.apache.commons.math.transform.FastFourierTransformer | 0.63 | 0.62 | 0.42 | 0.62 | 0.63 | 0.53 |
| org.joda.time.format.DateTimeFormat | 0.81 | 0.74 | **1.00** | 0.83 | 0.77 | **1.00** |
| scs.Ordered4 | 0.97 | 0.95 | 0.61 | 0.95 | 0.95 | 0.46 |
| roops.extended.bv32.arr.noex.IntArrayWithoutExceptionsWithArrayParameters | 0.90 | 0.90 | 0.50 | 0.90 | 0.90 | 0.50 |
| Average | 0.86 | 0.79 | 0.74 | 0.89 | 0.88 | 0.61 |

data in Table 2 suggest is that, when seeding is employed, there are still some branches that are not covered with the GA, and so require the local search of the MA to be reached.

To answer **RQ2** we can look at the configuration that gave the best result on average. This configuration uses an MA algorithm with seeding, small population size (five individuals), low rate of local search (every 75 generations), and a small budget of five fitness evaluations for local search. In other words, on average the best result is achieved using local search infrequently and with a low budget. This is surprising, and in Table 2 we see that the results change significantly between individual classes. This suggests that the benefit of local search is highly dependent on the problem at hand. For example, in a class with many string inputs, much of the budget may be devoted to local search, even if the input strings have no effect on code coverage levels. Although we do see an improvement, even on average, this clearly points out the need for parameter control—in order to adaptively change the local search configuration to the class under test and current state of the search.

At any rate, one problem with parameter tuning is that, given a large set of experiments from which we choose the best configuration, such a configuration could be too specific for the employed case study [4]. This is a common problem that in Machine Learning is called *overfitting* [19].

To reduce the threats of this possible issue, we applied a k-fold cross validation on our case study (for more details, see for example [19]). Briefly, we divided the case study in $k = 16$ groups, chose the best configuration out of the 250 on $k - 1$ groups (training), and calculated its performance on the remaining group (validation). This process is then repeated $k$ times, each time using a different group for the validation. Then, the average of these $k$ performance values on the validation groups is used as an estimate of actual performance of tuning on the entire case study (the "tuned" configuration) when applied on other new classes (i.e., does the tuning process overfit the data?).

The obtained estimate for best MA configuration was 0.89, which is the same as the average value 0.89 in Table 2. Therefore, the best parameter configuration we found is not overfitted to the case study examples.

> **RQ2***: The best configuration is the MA with small population size and local search applied infrequently with small search budget.*

The time spent for test data generation (i.e., the testing budget) is perhaps the only parameter that practitioners would need to set. For a successful technology transfer from academic research to industrial practice, the internal details (i.e., how often and how long to run local search inside EvoSuite) of a tool should be hidden from the users, and thus this choice should be made before the tools are released to the public. However, usually the best parameter configuration is strongly related to the testing budget [4].

To answer **RQ3**, we studied the performance of the tuned MA and the tuned GA at different time intervals. In particular, during the execution of EvoSuite, for all the configurations we kept track of the best solution found so far at every minute (for both the GA and the MA). With all these data, at every minute we also calculated the "best" MA configuration (out of 250) and the "best" GA (out of 10) at that particular point in time. By definition, the performance of the "tuned" MA is equal or lower than the one of the "best" MA. Recall that "tuned" is the configuration that gives the "best" results at 10 minutes.

From a practical stand point, it is important to study whether the "tuned" MA is stable compared to the "best" MA. In other words, if we tune a configuration considering a 10 minute timeout, are we still going to get good results (compared to the "best" MA and GA) if the practitioner decides to stop the search beforehand? Or was 10 minutes just a lucky choice? Figure 2 answers these questions by showing that, already from three minutes on, "tuned" performs very similar to the "best" configuration. Furthermore, regardless of the time, there is always a large gap between the "tuned" MA and GA.

Figure 2 shows the results averaged on all 16 classes in the case study. Thanks to the relatively small number of classes, in Figure 3 we can show the time analysis for each class

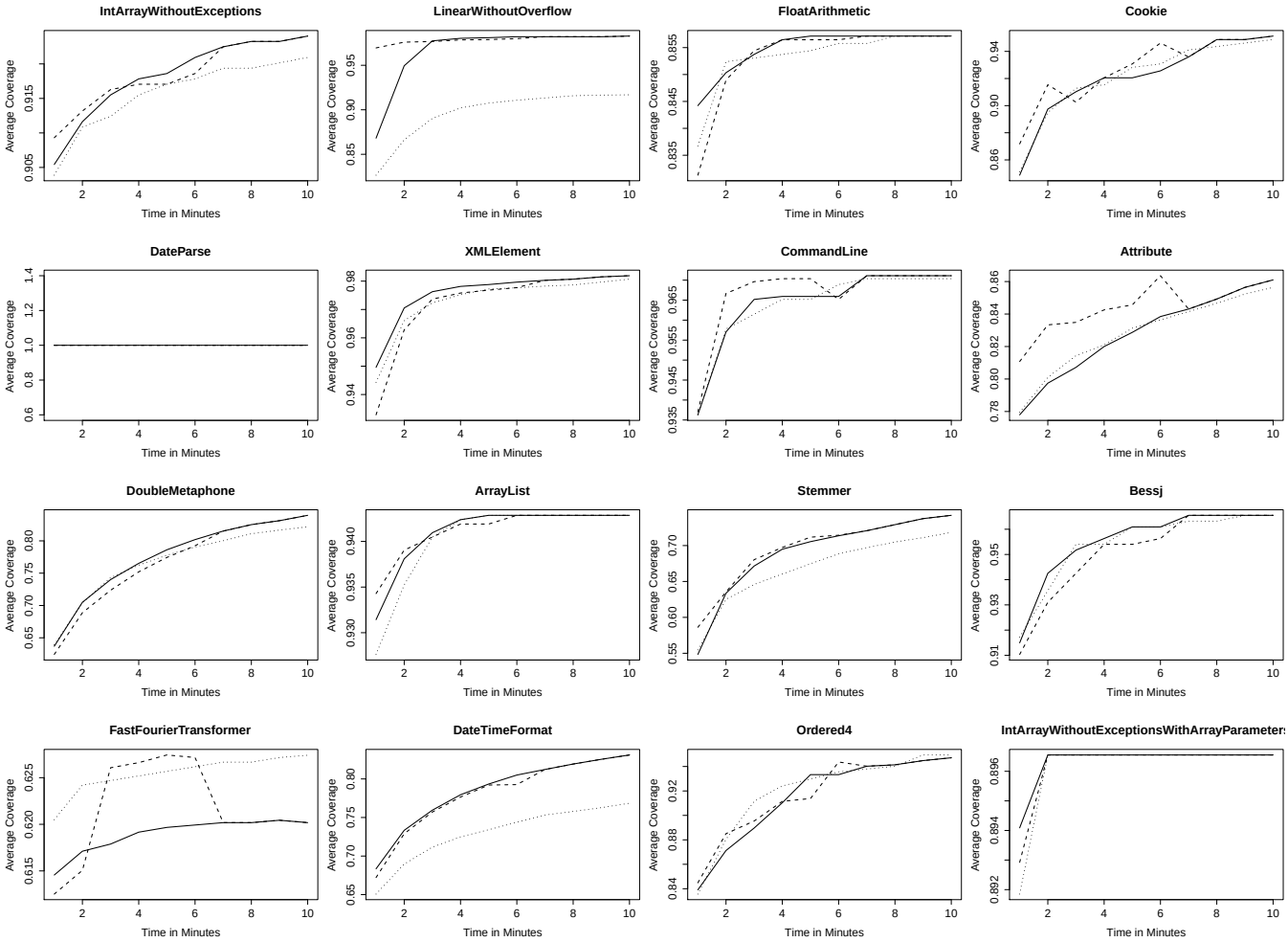


**Figure 3: For each class in the case study, average coverage at different points in time for the "best" GA (dotted line), "best" MA (dashed line) and "tuned" MA at 10 minutes (solid line).**

individually. The results provide interesting further insight by showing very different behaviors among classes. This stresses the importance of a rigorous empirical procedure when testing techniques are analyzed.

A peculiar result in Figure 3 is that, for the best MA, the performance is not monotonically increasing through time (as it is in Figure 2). This is particularly evident for the class `FastFourierTransformer`. The reason is that, at each point (minute) in time, we are considering the configuration with highest coverage averaged over all the 16 classes. Although on average the performance improves monotonically (Figure 2), on single classes in isolation everything could in theory happen (Figure 3).

**RQ3***: The best configuration only differs for small search budgets, and is consistent across higher budgets.*

# 6. THREATS TO VALIDITY

This paper compares the whole test suite generation approach based on a Genetic Algorithm to a hybrid version that uses a Memetic Algorithm with local search. Threats to *construct validity* are on how the performance of a testing technique is defined. We measured the performance in terms of branch coverage. However, in practice we might not want a much larger test suite if the achieved coverage is only slightly higher. Furthermore, this performance measure does not take into account how difficult it will be to manually evaluate the test cases and to add assert statements (i.e., to check the correctness of the outputs).

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 30 times, and we followed rigorous statistical procedures to evaluate their results.

There is also the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Because of the large number of experiments required, we only used 16 classes for our evaluation. To make claims on generalization we will need to conduct further studies on representative sets of classes.

# 7. CONCLUSIONS

Whole test suite generation has been shown to be effective at producing test suites with high coverage for object-oriented classes. However, as mutations on any particular

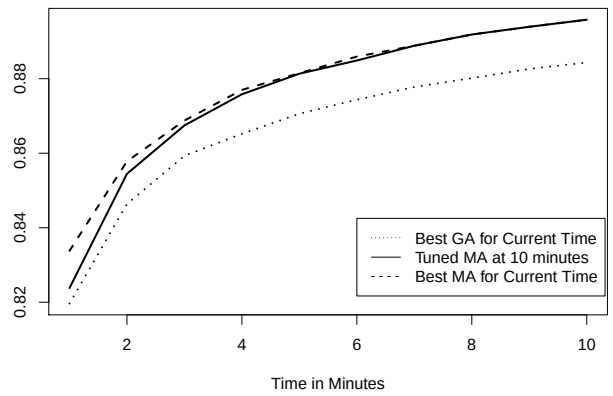


**Figure 2: Average coverage at different points in time for the "best" GA (at each minute interval), the MA tuned at 10 minutes, and "best" MA configuration at each minute interval.**

statement in a test suite of sequences of method calls have a low probability of occurring, it is not necessarily efficient. To overcome this problem, we have defined a set of local search operators, and extended the Genetic Algorithm used in the EvoSuite test generation tool to a Memetic Algorithm. Experiments on a set of case study classes have demonstrated that this approach results in higher coverage given a fixed search budget. We have observed that the effect is very dependent on the class on which test generation is applied, which makes it difficult to find an optimal parameter configuration. It will therefore be important for future work to make the local search *adaptive*, such that local search operators that lead to success on a particular problem instance are applied more frequently than those that are not.

For more information about EvoSuite please visit:

`http://www.evosuite.org/`

**Acknowledgments.** This project has been funded by a Google Focused Research Award on "Test Amplification", the Norwegian Research Council, and the EPSRC project "RE-COST" (EP/I010386).

## 8. REFERENCES

[1] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability*, 16(3):175–203, 2006.

[2] A. Arcuri. Theoretical analysis of local search in software testing. In *Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, pages 156–168, 2009.

[3] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)*, 2012. (to appear).

[4] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 33–47, 2011.

[5] A. Arcuri and X. Yao. A memetic algorithm for test data generation of object-oriented software. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2048–2055, 2007.

[6] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for java. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.

[7] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[8] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 121–130, 2012.

[9] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[10] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[11] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[12] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[13] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 73–83, 2007.

[14] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering (TSE)*, 36(2):226–247, 2010.

[15] G. Kapfhammer, P. McMinn, and C. J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013.

[16] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[17] Y. Li and G. Fraser. Bytecode testability transformation. In *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2011.

[18] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[19] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[20] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report 826*, 1989.

[21] P. Tonella. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[22] H.-C. Wang and B. Jeng. Structural testing using memetic algorithm. In *Proceedings of the Second Taiwan Conference on Software Engineering*, 2006.