A Memetic Algorithm for Whole Test Suite Generation

Gordon Fraser[a], Andrea Arcuri[b], Phil McMinn[a]

[a] *University of Sheffield, Department of Computer Science, 211 Regent Court, Portobello, S1 4DP, Sheffield*
[b] *Certus Software V&V Center, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway*

Abstract

The generation of unit-level test cases for structural code coverage is a task well-suited to Genetic Algorithms. Method call sequences must be created that construct objects, put them into the right state and then execute uncovered code. However, the generation of primitive values, such as integers and doubles, characters that appear in strings, and arrays of primitive values, are not so straightforward. Often, small local changes are required to drive the value towards the one needed to execute some target structure. However, global searches like Genetic Algorithms tend to make larger changes that are not concentrated on any particular aspect of a test case. In this paper, we extend the Genetic Algorithm behind the EvoSuite test generation tool into a Memetic Algorithm, by equipping it with several local search operators. These operators are designed to efficiently optimize primitive values and other aspects of a test suite that allow the search for test cases to function more effectively. We evaluate our operators using a rigorous experimental methodology on over 12,000 Java classes, comprising open source classes of various different kinds, including numerical applications and text processors. Our study shows that increases in branch coverage of up to 53% are possible for an individual class in practice.

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

1. Introduction

Search-based testing uses optimization techniques such as Genetic Algorithms to generate test cases. Traditionally, the technique has been applied to test inputs for procedural programs, such as those written in C (McMinn, 2004). More recently, the technique has been applied to the generation of unit test cases for object-oriented software (Fraser and Arcuri, 2013b). The problem of generating such test cases is much more complicated than for procedural code.

Email addresses: gordon.fraser@sheffield.ac.uk (Gordon Fraser), arcuri@simula.no (Andrea Arcuri), p.mcminn@sheffield.ac.uk (Phil McMinn)

```
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.

To generate tests that cover all of the branches in a class, for example, the class must be instantiated, and a method call sequence may need to be generated to put the object into a certain state. These method calls may themselves require further objects as parameters, or primitive values such as integers and doubles, or strings, or arrays of values. The EvoSuite tool (Fraser and Arcuri, 2011) uses Genetic Algorithms to generate a whole test suite, composed of a number of test cases. Although empirical experiments have shown that it is practically usable on a wide range of programs (Fraser and Arcuri, 2012b), Genetic Algorithms are a global search technique, which tend to induce macro changes on the test suite. In order to cover certain branches, more focused changes are required. If, for example, somewhere in the test suite there is a particular integer variable, the probability of it being mutated during the search with a Genetic Algorithm is low, and so the optimization towards particular branches dependent on this value will take a long time. The difficulty of this problem becomes even more apparent when one takes into account string variables. Consider the example test case in Figure 1: Transformations of the string-based branching statement (see Section 3.3) provide guidance to the search for an input to reach the true-branch. However, even under very strong simplifications, a "basic" Genetic Algorithm 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 the approach may 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:

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

global search for test suites.

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

3. **Test suite improvement:** We define operators on test suites that allow test suites to improve themselves during phases of Lamarckian learning.

4. **Sensitivity analysis:** We have implemented the approach as an extension to the EvoSuite tool (Fraser and Arcuri, 2013b), and analyze the effects of the different parameters involved in the local search, and determine the best configuration.

5. **Empirical Evaluation:** We evaluate our approach in detail on a set of 16 open source classes as well as two large benchmarks (comprising more than 12,000 classes), and compare the results to the standard search-based approach that does not include local search.

This paper is an extension of (Fraser et al., 2013), and it is organized as follows: Section 2 presents relevant background to search-based testing, and the different types of search that may be applied, including local search and search using Genetic and Memetic Algorithms. Section 3 discusses the global search and fitness function applied to optimize test suites for classes towards high code coverage with EvoSuite. Section 4 discusses how to extend this approach with local operators designed to optimize primitive values such as integers and floating point values, Strings and arrays. Section 5 then presents our experiments and discusses our findings, showing how our local search operators, incorporated into a Memetic Algorithm, result in higher code coverage. A discussion on the threats to validity of this study follows in Section 6. Finally, Section 7 concludes the paper.

2. Search-Based Test Case Generation

Search-based testing applies meta-heuristic search techniques to the task of test case generation (McMinn, 2004). In this section, we briefly review local and global search approaches to testing, and the combination of the two in the form of Memetic Algorithms.

2.1. Local Search Algorithms

With local search algorithms (Arcuri, 2009) 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 are typically overcome by restarting the search with new random

values, or with some other form of escape mechanism (e.g., by accepting a worse solution temporarily, as with simulated annealing). Different types of local search algorithms exist, including simulated annealing, tabu search, iterated local search and variable neighborhood search (see Gendreau and Potvin (2010), for example, for further details). A popular version of local search often used in test data generation is Korel's Alternating Variable Method (Korel, 1990; Ferguson and Korel, 1996). The Alternating Variable Method (AVM) is a local search technique similar to hill climbing, and was introduced by Korel (1990). 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, the 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 in the direction of improvement with so-called "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 improvement of the variable value 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 to local search algorithms, global search algorithms try to overcome local optima in order to find more globally optimal solutions. Harman and McMinn (2010) 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., a maximum time limit or a certain 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 specific branches—to achieve branch coverage of a program— a common fitness function (McMinn, 2004) integrates the *approach level* (number of unsatisfied control dependencies) and the *branch distance* (estimation of how close a

branching condition is to being evaluated 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 (Tonella, 2004; Fraser and Zeller, 2012).

2.3. Memetic Algorithms

A Memetic Algorithm (MA) hybridizes global and local search, such that the individuals of a population in a global search algorithm have the opportunity for local improvement in terms of local search. With *Lamarckian*-style learning local improvement achieved by individuals is encoded in the genotype and thus passed on to the next generation. In contrast, with *Baldwinian* learning, the improvement is only encoded in terms of the fitness value, whereas the genotype remains unchanged. The Baldwin effect describes that as a result, individuals with more potential for improvement are favoured during evolution, which essentially smoothes the fitness landscape. Yao et al. (2005) report no difference between the two types of learning, whereas other experiments showed that Baldwinian learning may lead to better results but takes significantly longer (Whitley et al., 1994). The use of MAs for test generation was originally proposed by Wang and Jeng (2006) in the context of test generation for procedural code, and has since then been applied in different domains, such as combinatorial testing (Rodriguez-Tello and Torres-Jimenez, 2010). Harman and McMinn (2010) analyzed the effects of global and local search, and concluded that MAs achieve better performance than global search and local search. In the context of generating unit tests for object-oriented code, Arcuri and Yao (2007) combined a GA with hill climbing to form an MA when generating unit tests for container classes. Liaskos and Roper (2008) also confirmed that the combination of global and local search algorithms leads to improved coverage when generating test cases for classes. Baresi et al. (2010) 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. Although MAs have been already used in the past for unit test generation, their applications have been mainly focused on numerical data types and covering specific testing targets (e.g., a branch) with a single test case. In this paper, we rather provide a comprehensive approach for object-oriented software, targeting whole test suites, handling different kinds of test data like Strings and arrays, and also considering adaptive parameter control. Furthermore, we provide an extensive empirical evaluation to determine how to best combine the local and global search parts of the presented MA.

3. Whole Test Suite Generation

With 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.

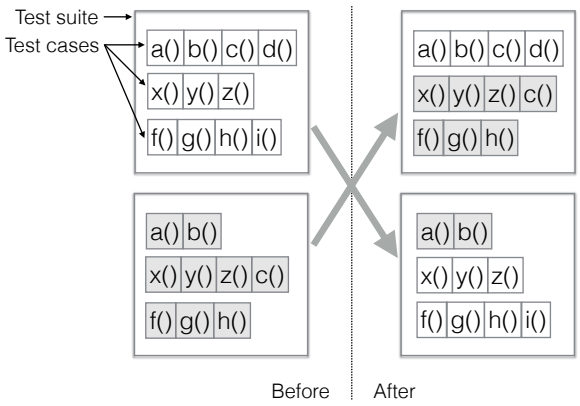

Figure 2: How the crossover operator works with test suites. Given two parent test suites (shown on the "before" side of the figure), two offspring test suites are produced (depicted on the "after" side of the figure) following splicing of the parent test suites at a given crossover point.

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 $t = \langle s_1,s_2,\ldots,s_l\rangle$, where the length of a test case is defined as $length(\langle s_1,s_2,\ldots,s_l\rangle) = l$. The length of a test suite is defined as the sum of the lengths of its test cases, i.e., $length(T) = \sum_{t\in T} length(t)$. 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 test cases from parent test suites, as Figure 2 shows. For two selected parents P_1 and P_2, a random value α 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 σ, a test case is added. If it is added, then a second test case is added with probability σ^2, and so on until the ith 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 ensures 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 the reader to (Fraser and Arcuri, 2013b).

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. Let $d(b,T)$ be the branch distance of branch b on test suite T:

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

We require each branching statement to be executed twice to avoid the situation where the search oscillates between the two potential evaluations of the branch predicate. ν is a normalization function (Arcuri, 2013) with the range [0,1]. Assuming the set of branches B for a given class under test, this leads to the following fitness function, which is to be minimized by the search:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b \in B} d(b,T)$$

Here, M is the set of methods in the class under test, while M_T is the set of methods executed by T.

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 (Li and Fraser, 2011). 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* (Harman et al., 2004), 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 (2006). 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 α^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 featuring in one of those test cases 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. Calculating the fitness value after a local search operator has been applied only requires partial fitness evaluation: EvoSuite stores the last execution trace with each test case, and from this the fitness value can be calculated. Whenever a test case is modified during the search, either by regular mutation or by local search, the cached execution trace is deleted. Thus, a fitness evaluation for local search only requires that one test out of a test suite is actually executed.

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 δ is subtracted. **Floating Point Datatypes:** For floating point variables (`float`, `double`) we use the same approach as originally defined by Harman and McMinn (2007) 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 δ.

4.1.2. String Statements

For string variables, exploratory moves are slightly more complicated than in the case of primitive statements: 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 modifying the string has some effect, regardless of whether the change resulted in an improvement in fitness 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

[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.

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. (Kapfhammer et al., 2013)):

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

4.1.3. 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 that 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.

4.1.4. 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:

```
int a = 10;
triType(a, a, a);
```

```
int a = 10;
int b = 10;
int c = 10;
triType(a, b, c);
```

Figure 3: Expanding primitive values. In the left-hand test case, only equilateral triangles are possible. Ensuring the method receives distinct primitive values for each of its parameters, as for the right-hand test case, allows for greater effectiveness when applying local search.

- 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.

4.2. Local Search on Test Suites

While the smallest possible search steps in the neighborhood of a test suite are defined by the tests' statements as discussed in the previous section, Lamarckian evolution in principle permits individuals to improve with any local refinements, and not just local search algorithms. This section describes some local improvements that can be performed on test suites with respect to the goal of achieving high code coverage.

4.2.1. Primitive Value Expansion

The search operators creating sequences of method calls allow variables to be reused in several statements. This is beneficial for certain types of coverage problems: For example, the case of an equilateral triangle (thus requiring three equal integer inputs) in the famous triangle example becomes a trivial problem when allowing variable reuse. However, variable reuse can also inhibit local exploration. In the case of the triangle example, given a test that creates an equilateral triangle using only a single variable, it is impossible for local search on the primitive values in the test to derive any other type of triangle. Therefore, a possible local improvement of test suite lies in making all variables uniquely used. That is, the triangle case would be converted to a test with three variables that have the same value, thus permitting local search to optimize each side independently again.

4.2.2. Ensuring Double Execution

The branch coverage fitness function (subsection 3.2) requires that each branching statement is executed twice, in order to avoid that the search oscillates between the true/false outcomes at the branching statement. If for a given test suite a branching predicate is covered only once, then it is possible to improve the test suite simply by duplicating the test that covers the predicate.

4.2.3. Restoring Coverage

The fitness function captures the overall coverage of a test suite, and how close it is to covering more branches. This means that the fitness value does not capture *which* branches are covered, and so a test suite with worse fitness than another might still cover some branches the "better" test suite does not cover. Again it is possible to apply a local improvement measure to counter this issue: We keep a global archive of tests, and whenever a new branch is covered for the first time, this test is added to the archive. If a test suite determines that it is not covering branches that have been covered in the past, it can take the according test cases from that archive.

4.3. A Memetic Algorithm for Test Suites

Given the ability to perform local search on the individuals of a global optimization there is the question of how to integrate these techniques. Considering the high costs of fitness evaluations in the test generation scenario, a generally preferred choice (El-Mihoub et al., 2006) is Lamarckian learning, i.e., the local search changes the genotype and its fitness value, rather than just the fitness value. A common implementation of MAs applies this learning immediately after reproduction (Moscato, 1989). However, there remain several different parameters (El-Mihoub et al., 2006): How often to apply the individual learning? On which individuals should it be applied? How long should it should be performed? In EvoSuite, the choice of how often to apply local learning depends on two parameters:

- **Rate:** The application of individual learning is only considered every r generations. For example, if $r = 1$, then it is considered in every generation.
- **Probability:** If the rate parameter decides that the current iteration is a candidate the local search, then this is done with a probability p. For example, if $p = 1$, then local search is applied at all generations selected by r.

Algorithm 1 shows how these parameters are used in the MA: Except for Lines 5 to 17, this algorithm represents a regular GA. If the current iteration matches the rate with which local search should be applied, then with a given probability the local search is applied to one individual of the current population after the other, until the local search budget is used up. One possible strategy to select individuals for local search is to apply it to the worst individuals (El-Mihoub et al., 2006), which supports exploration. However, we expect fitness evaluations and local search in the test generation scenario to be very expensive, such that it can be applied only to few individuals of the population. Furthermore, test suite generation is a scenario where the global optimization alone may not succeed in finding a solution (e.g., consider the string example in Figure 1). Therefore, we 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. The strategy implemented in EvoSuite is thus to start applying local search to the best individual of the population, then the second best, etc.,

Algorithm 1 A basic memetic algorithm, where a regular GA is extended by local search before regular reproduction on selected generations.

Require: Class under test C
Ensure: Test suite T

```
1: procedure MA(C)
2:     current_population ← random population
3:     iteration ← 1
4:     repeat
5:         if iteration mod local search rate = 0
6:           ∧ local search probability then
7:             while budget for local search available do
8:                 x ← select next individual from Z
9:                 x' ← local search on x
10:                if local search successful then
11:                    Z ← Z ∪ {x'}\{x}
12:                    Increase local search probability
13:                else
14:                    Decrease local search probability
15:                end if
16:            end while
17:        end if
18:        while |Z| ≠ |current_population| do
19:            P1,P2 ← select two parents
20:            if crossover probability then
21:                O1,O2 ← crossover P1,P2
22:            else
23:                O1,O2 ← P1,P2
24:            end if
25:            if mutation probability then
26:                O1,O2 ← mutate O1,O2
27:            end if
28:            f_P = min(fitness(P1),fitness(P2))
29:            f_O = min(fitness(O1),fitness(O2))
30:            if f_O ≤ f_P then
31:                Z ← Z ∪ {O1,O2}
32:            else
33:                Z ← Z ∪ {P1,P2}
34:            end if
35:            iteration ← iteration + 1
36:        end while
37:        current_population ← Z
38:    until solution found or maximum resources spent
39: end procedure
```

until the available budget for local search is used up. The local search budget in EvoSuite can be defined in terms of fitness evaluations, test executions, number of executed statements, number of individuals on which local search is applied, or time. Finally, a further parameter determines the *adaptation rate*: If local search was successful, then the probability of applying it at every r-th generation is increased, whereas an unsuccesful local search leads to a reduction of the probability. For this we use the approach that EvoSuite successfully applied to combine search-based testing and dynamic symbolic execution (Galeotti et al., 2013). The adaptation rate a updates the probability p after a successful (i.e., fitness was improved) local search as follows:

$$p = \min(p \times a,\, 1)\ , \tag{1}$$

whereas on unsuccessful local search it is updated to:

$$p = \frac{p}{a}\ . \tag{2}$$

Optionally, EvoSuite implements a strategy where local search is restricted to those statements where a mutation in the reproduction phase has led to a fitness change (Galeotti et al., 2013).

5. Evaluation

The techniques presented in this paper depend on a number of different parameters, and so evaluation needs to take these into account. As the problem is too complex to perform a theoretical runtime analysis (e.g., such as that presented by Arcuri (2009)), we therefore aim to empirically answer the following research questions:

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

RQ2: How does the configuration of the memetic algorithm influence the results?

RQ3: How does the available search budget influence the results?

RQ4: What is the influence of each individual type of local search operator?

RQ5: Which combination of local search operators achieves the best results?

RQ6: Does adaptive local search improve the performance?

RQ7: Do results generalize to other classes?

5.1. Case Study Selection

For **RQ1**–**RQ6** we need a small set of classes on which to perform extensive experiments with many different combinations of parameters. Therefore, we chose classes already used in previous experiments (Arcuri and Fraser, 2013), but excluded those on which EvoSuite trivially already achieves 100% coverage, as there was no scope for improvement with local search. In order to ensure that the set of classes for experimentation was variegated, we tried to strike a balance among different kinds of classes. To this end, beside classes taken from

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

Table 2: Details of the generated case study. For each project, we report how many classes it is composed of, and the total number of bytecode branches.

Name	# Classes	# Branches
tp1m	751	758,717
sp500k	301	307,546
tp10k	101	12,744
tp80k	81	61,560
tp50k	51	31,554
tp5k	31	2,850
tp7k	31	4,045
tp2k	21	1,041
tp1k	16	659
tp300	4	177
tp600	4	341
Total:	1,392	1,181,234

the case study in (Arcuri and Fraser, 2013), 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. For **RQ7** we need large sets of classes with different properties. First, we used the SF100 corpus of classes (Fraser and Arcuri, 2012b). The SF100 corpus is a random selection of 100 Java projects from SourceForge, one of the largest repositories of open source projects. In total, the SF100 corpus consists of 11,088 Java classes. On one hand, the SF100 corpus is an ideal case study to show how a novel technique would affect software engineering practitioners. On the other hand, there are several research questions in unit test generation that are still open and may influence the degree of achievable improvement, like handling of files, network connections, databases, GUI events, etc. Therefore, we used the case study of the Carfast (Park et al., 2012) test generation tool[3] as a second case study, as it represents a specific type of difficult classes that could be efficiently addressed with a hybrid local search algorithm. Table 2 summarizes the properties of this case study. Note that the Carfast paper mentions a second case study with about 1k LOC, which is not included in the archive on the website and therefore not part of our experiments.

5.2. Experiment Parameters

In addition to the parameters of the MA, local search is influenced by several other parameters of the search algorithm. Because how often we apply local search depends on the number X of generations, how much local search is actually performed 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 (Fraser and Arcuri, 2012a) as a further parameter to experiment with. In bytecode seeding, all constant values (e.g., numbers and strings) in the code of the class under test are added to a special pool of values that the search algorithm can employ when sampling new values, instead of doing that at random. We ran experiment with and without seeding because we expected it to have a large impact on the performance for case studies where local search is successful (as we later found to be confirmed by the experiments). In total, we ran four different sets of experiments to answer the different research questions, each requiring different parameter configurations: **Experiment 1 (RQ1 – RQ3):** For population size, local search budget and rate we considered five different values, i.e., {5, 25, 50, 75, 100}, while the interpretation chosen for the local search budget was "seconds". We controlled the use of constant seeding by setting the probability of EvoSuite using seeded constants to either 0.0 or 0.2. 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. In total, EvoSuite was run on $(2 \times 5^3) + (2 \times 5) = 260$ configurations. For each class we used an overall search budget of 10 minutes,

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

[3] Available at: http://www.carfast.org, accessed June 2013

but for **RQ3** we also look at intermediate values of the search. **Experiment 2 (RQ4 – RQ5):** We considered all possible combinations of the local search operators defined in Section 4, i.e., search on strings, numbers, arrays, reference types, as well as ensuring double execution, expanding test cases, and restoring coverage. Together with the seeding option, this results in $2^8 = 256$ different combinations. The values chosen for population size, rate, and budget are those that gave the best result in **RQ2**. As we do not consider the behavior of the search over time, we use a search budget of 2 minutes per class, a value which our past experience has shown to be a reasonable compromise between a runtime practitioners would accept and allowing for good coverage results. **Experiment 3 (RQ6):** We considered the probabilities {0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 0.7, 1.0} and adaptation rates of {1.0, 1.2, 1.5, 1.7, 2.0, 5.0, 10} (i.e., a suitable coverage of values between the minimum and maximum plausible values), while the local search rate is set to 1. We also experimented whether selective mode was active or not, as well as seeding, which led to $8 \times 7 \times 2^2 = 224$ configurations. The overall search budget is again 2 minutes per class. **Experiment 4 (RQ7):** For the experiments on the SF100 corpus and Carfast case study we only considered two configurations: default GA in EvoSuite and the best MA configuration from the analyses of the previous research questions. Search budget is 2 minutes per class also in this set of experiments.

5.3. Experiment Procedure

On each class, for each parameter combination and algorithm, we ran EvoSuite 10 times with different random seeds to take into account their random nature. This means that the first set of experiments required $(260 \times 16 \times 10 \times 10)/(60 \times 24) = 288$ days of computational time, the second required $(256 \times 16 \times 2 \times 10)/(60 \times 24) = 57$ days, the third $(224 \times 16 \times 2 \times 10)/(60 \times 24) = 50$ days, and finally the last set of experiments needed $(1{,}392 + 11{,}088) \times (2 \times 2 \times 10)/(60 \times 24) = 347$ days. Thus, in total all the experiments together took 742 days of computational time, which required the use of a cluster of computers. We used the University of Sheffield's Linux based high performance computing cluster which has a total of 1544 CPUs. The nodes of the cluster have either AMD or Intel CPUs at around 2.6GHz, and 4GB of memory per CPU core. During all these runs, EvoSuite was configured using the optimal configuration determined in our previous experiments on tuning (Arcuri and Fraser, 2013). To evaluate the statistical and practical differences among the different settings, we followed the guidelines by Arcuri and Briand (2014). Statistical difference is evaluated with a two-tailed Mann-Whitney-Wilcoxon U-test (at 0.05 significant level), whereas the magnitude of improvement is quantified with the Vargha-Delaney standardized effect size $\hat{A}_{12}$. In some cases, it is sufficient to determine which configuration gives best result. In other cases, it is useful to analyse trends among the different configuration parameters and their combinations. However, when there are hundreds of configuration settings based on several parameters, the issue of how to visualize them is not so straightforward. In this paper, when we do this kind of analysis, we create *rank* tables, in a similar way as we did in previous work (Fraser and Arcuri, 2013a). In a rank table, we compare the

effectiveness of each configuration against all other configurations, one at a time. For example, if we have $X = 250$ configurations, we will have $X \times (X - 1)$ comparisons, which can be reduced by half due to the symmetric property of the comparisons. Initially, we assign a score of 0 to each configuration. For each comparison in which a configuration is statistically better (using a U-test at 0.05 level), we increase its score by one, and we reduce it by one in case it is statistically worse. Therefore, in the end each configuration has a score between $-X$ and $+X$. The higher the score, the better the configuration is. After this first phase, we rank these scores, such that the highest score has the best rank, where better ranks have lower values. In case of ties, we average the ranks. For example, if we have five configurations with scores {10, 0, 0, 20, -30}, then their ranks will be {2, 3.5, 3.5, 1, 5}. We repeat this procedure for all the Z classes in the case study, and we calculate the average of these ranks for each configuration, for a total of $Z \times X \times (X - 1)/2$ statistical comparisons. For example, if we consider $X = 250$ configurations and $Z = 16$ classes, this results in 498,000 statistical comparisons. This a very large number of comparisons, which can lead to a high probability of Type I error if we consider the hypothesis that *all* tests are significant at the same time. The issue of applying adjustments such as Bonferroni corrections, however, is a complex one, and there is no full consensus amongst statisticians as to their application. In this paper we have not to applied such corrections, for reasons discussed by Arcuri and Briand (2014); Perneger (1998); Nakagawa (2004), with which we are in agreement with.

5.4. RQ1: Does local search improve the performance?

For both cases where seeding was applied and where it was not, we analyzed the $5^3 = 125$ configurations using the MA, and chose the one that resulted with highest average coverage over the 16 classes in the case study. We did the same 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 3 shows the comparison between the tuned MA and tuned GA configuration based on whether seeding was used. Results in Table 3 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 $98 - 45 = 53\%$ average coverage difference (when seeding is not used).

> **RQ1:** *The MA achieved up to 53% higher branch coverage than the standard GA.*

When considering the case without seeding enabled, there are no classes where the MA resulted in significantly lower coverage; however, the effect size is worse for the MA for the classes `IntArrayWithoutExceptionsWithArrayParameters`, `XMLElement`, `ArrayList` and `Bessj`, although difference in coverage are no more than 1%. Some local search operators may thus lead to lower coverage, and this will be analyzed in detail as part of **RQ4**. With seeding enabled MA is still

Table 3: Results for **RQ1**. Average coverage obtained by the tuned MA is compared with the tuned GA, using a 10 minutes search budget. 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	$\hat{A}_{12}$
IntArrayWithoutExceptions	0.93	0.88	**1.00**	0.93	0.93	0.50
LinearWithoutOverflow	0.99	0.73	**1.00**	0.99	0.91	**1.00**
FloatArithmetic	0.65	0.49	**1.00**	0.86	0.86	0.50
IA.WithArrayParameters	1.00	1.00	0.49	1.00	1.00	0.50
Cookie	0.98	0.45	**0.99**	1.00	0.95	**0.85**
DateParse	0.97	0.77	**1.00**	1.00	1.00	0.50
Stemmer	0.76	0.72	**0.82**	0.76	0.71	**0.89**
Ordered4	1.00	0.97	**0.85**	1.00	0.98	0.65
XMLElement	0.98	0.98	0.27	0.98	0.99	0.27
CommandLine	0.98	0.98	0.50	0.98	0.98	0.50
Attribute	0.86	0.76	**1.00**	0.89	0.85	**1.00**
DoubleMetaphone	0.76	0.75	0.72	0.84	0.80	**1.00**
ArrayList	0.91	0.92	0.45	0.93	0.93	0.55
Bessj	0.95	0.96	0.45	0.96	0.95	0.55
FastFourierTransformer	0.74	0.71	0.64	0.76	0.78	0.53
DateTimeFormat	0.80	0.78	0.74	0.79	0.77	**0.86**
Average	0.89	0.80	0.75	0.92	0.90	0.67

clearly better overall. Only for `XMLElement` the results are slightly worse, but these are not statistically significant.

5.5. RQ2: How does the configuration of the MA influence the results?

One thing that is clearly visible in Table 3 is that seeding, as expected (Fraser and Arcuri, 2012a), leads to higher results. On one hand, when seeding is not used, the difference in average coverage between the MA and the GA is $89 - 80 = 9\%$. On the other hand, when seeding is used, the difference is $92 - 90 = 2\%$. At a first look, such an improvement might be considered low. But the statistics in Table 3 point out a relatively high average effect size of 0.67, with six classes having a strong statistical difference. This is not in contrast with the 2% difference in the raw values of the achieved coverage. What the data in Table 3 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, large population size (100 individuals), low rate of local search (every 100 generations) and a small budget of 25 seconds for local search. In other words, on average the best result is achieved using local search infrequently

Table 4: Rank analysis for **RQ2**. Out of the 250 configurations for MA, only the top 25 using seeding and the 25 without are displayed.

Seeding	Population	Rate	Budget	Rank	Coverage
√	100	100	25	55.062	0.917
√	100	100	50	55.531	0.915
√	75	100	50	58.625	0.910
√	100	75	50	58.875	0.914
√	100	75	75	60.438	0.910
√	75	100	75	61.438	0.912
√	100	75	25	61.656	0.912
√	25	75	5	65.750	0.909
√	100	100	5	67.938	0.910
√	50	100	5	68.281	0.911
√	100	75	5	68.438	0.910
√	75	100	5	68.562	0.910
√	75	75	5	69.531	0.909
√	50	100	25	69.938	0.908
√	25	100	5	70.219	0.908
√	75	100	100	70.875	0.908
√	25	100	25	70.906	0.908
√	50	75	5	71.312	0.912
√	75	100	25	71.500	0.907
√	50	50	5	71.750	0.907
√	50	100	75	72.062	0.906
√	75	50	5	73.094	0.909
√	75	75	50	73.188	0.908
√	100	100	75	73.750	0.908
√	100	50	5	74.031	0.908
	25	100	5	98.250	0.889
	100	75	25	99.438	0.889
	100	75	75	104.688	0.885
	100	100	25	105.688	0.887
	100	100	50	106.500	0.887
	75	100	5	106.688	0.889
	50	100	25	107.125	0.888
	50	75	5	110.188	0.885
	100	100	5	110.625	0.887
	25	75	5	110.906	0.884
	75	100	25	111.281	0.891
	50	25	5	113.500	0.883
	50	75	25	114.688	0.885
	75	75	5	114.719	0.884
	50	100	5	114.938	0.883
	75	100	50	115.344	0.885
	25	100	25	116.375	0.885
	25	100	50	116.406	0.884
	25	75	100	117.156	0.886
	75	75	25	117.250	0.884
	50	100	50	118.031	0.887
	100	100	100	118.312	0.884
	100	75	100	119.719	0.885
	75	75	50	120.000	0.885
	75	50	25	121.000	0.881

and with not a large budget. This is different from the result of our initial experiment (Fraser et al., 2013), where the best configuration used 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. Table 4 shows the results of the rank analysis on those 250 MA configurations. For space reasons, we only show the results of 50 configurations: the 25 top configurations using seeding, and the top 25 that do not use seeding. The results in Table 4 clearly show that, on average, seeding has a strong impact on performance (all the 25 top configurations using seeding achieve better results than the top 25 not using seeding). Among the top configurations, there is a clear trend pointing to large population values, local search applied infrequently, and for a short period of time (i.e., low budget). This may seem in slight contrast to the results of our initial experiments in Fraser et al. (2013), where the best result was achieved with seeding, small population size (5), low rate of local search (every 75 generations), and a small budget (5) for local search. However, this difference can be attributed to a) the variance in the results (the top configurations are all very similar in terms of achieved coverage), b) differences in the local search operators, c) optimisations introduced in this paper that make it feasible to apply local search on larger populations, e.g. the original experiments did not include primitive value expansion and restoring coverage. In general, these results suggest that, although local search does improve performance, one has to strike the right balance between the effort spent on local search and the one spent on global search (i.e., the search operators in the GA). Considering Table 3, 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 (Arcuri and Fraser, 2013). This is a common problem that in Machine Learning is called *overfitting* (Mitchell, 1997). 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 (Mitchell, 1997)). 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?). Note, we used a 16-fold cross validation instead of a typical 10-fold cross validation as we have only 16 classes, and dividing them into 10 groups would had partitioned them in very unbalanced groups (i.e., some groups with only one element whereas others

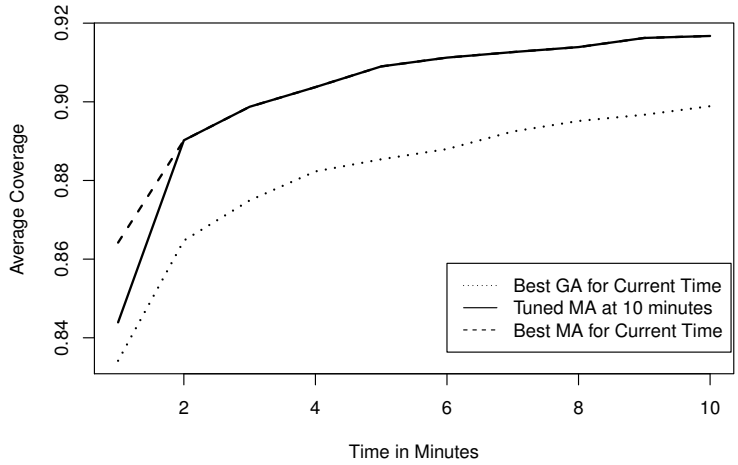

Figure 4: Average coverage at different points in time for the "best" GA (at each minute interval), the MA tuned at 10 minutes, and "best" MA configuration at each minute interval.

with twice as much). The obtained estimate for the best MA configuration was 0.91, which is close to the average value 0.92 in Table 3. Therefore, the best parameter configuration we found is not overfitted to the case study examples.

RQ2*: The MA gives the best results when local search is applied infrequently with a small search budget.*

5.6. RQ3: How does the search budget influence the results?

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 (Arcuri and Fraser, 2013). 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

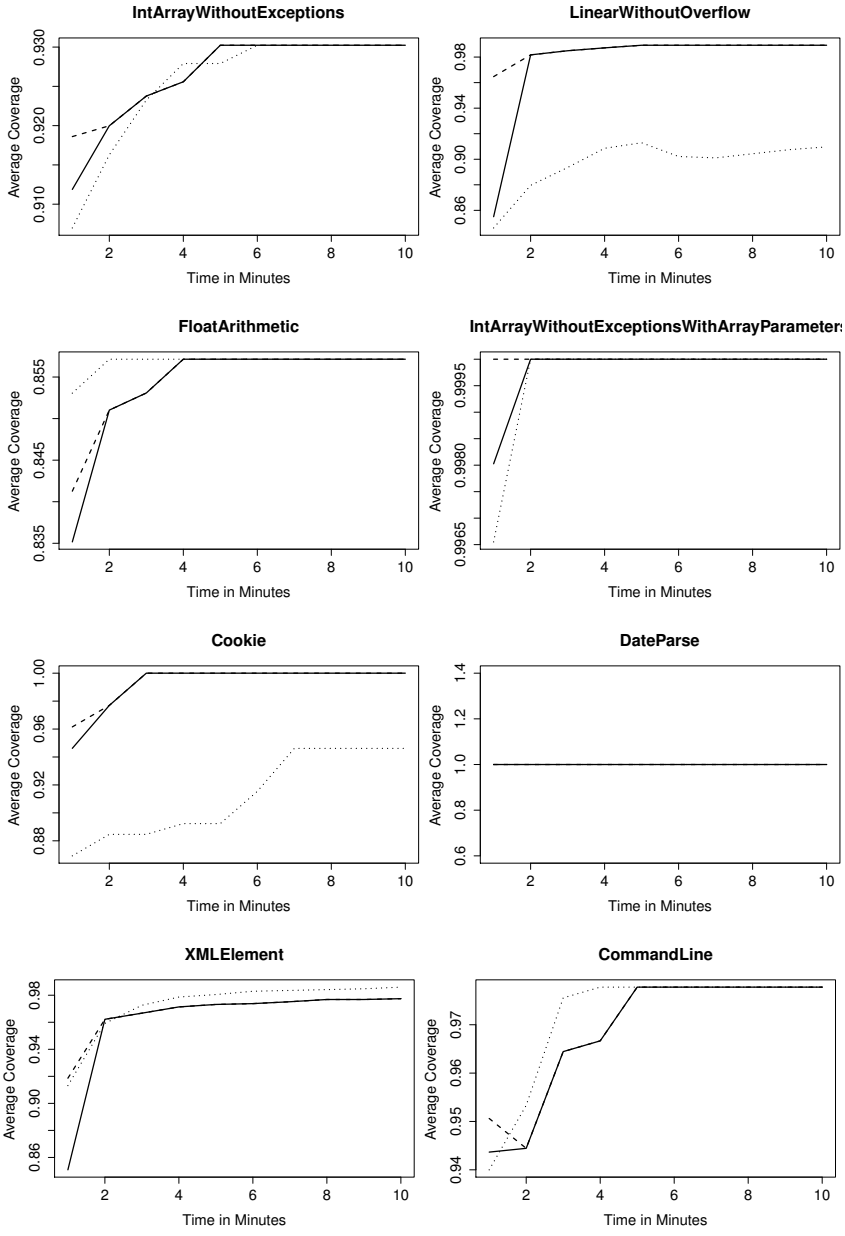

Figure 5: For the first eight classes in the case study, average coverage at different points in time for the "best" GA (dotted line), "best" MA (dashed line) and "tuned" MA at 10 minutes (solid line).

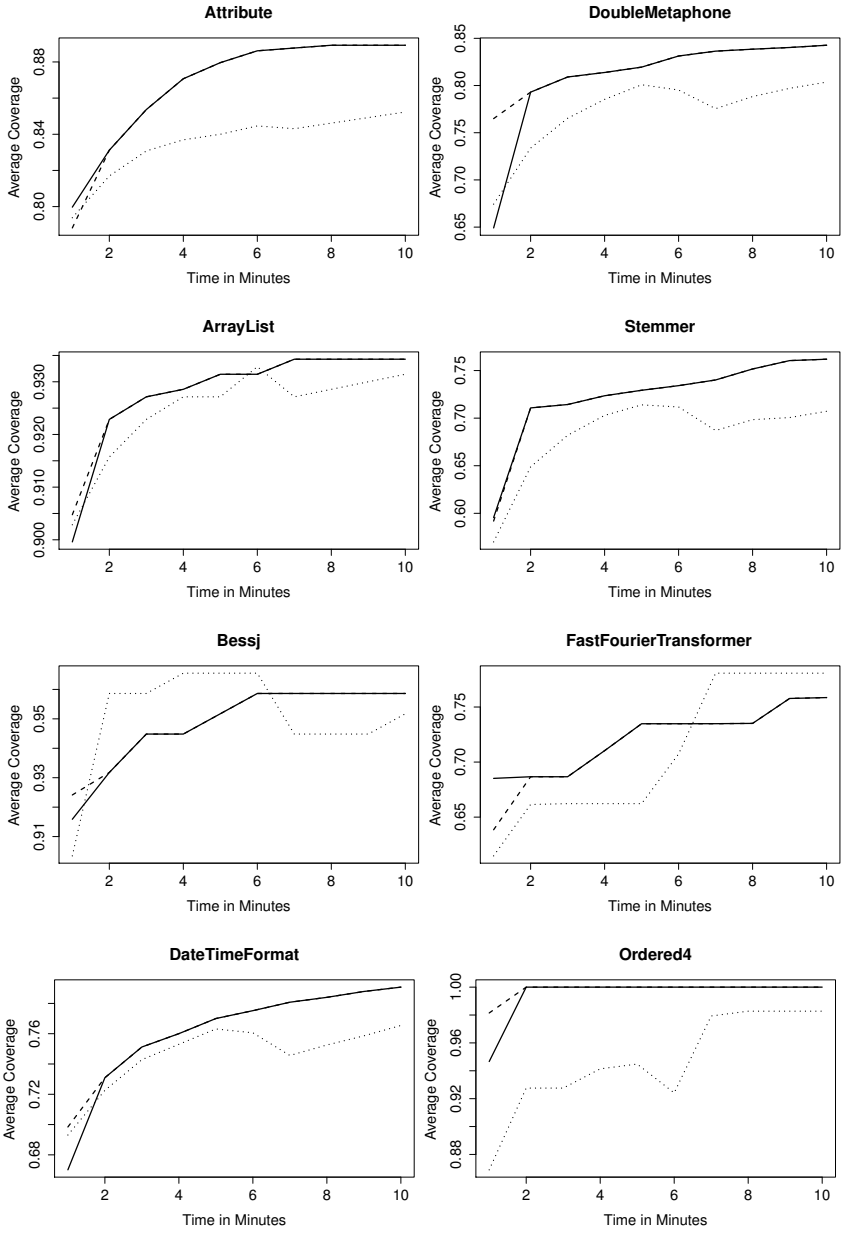

Figure 6: For the second eight classes in the case study, average coverage at different points in time for the "best" GA (dotted line), "best" MA (dashed line) and "tuned" MA at 10 minutes (solid line).

just a lucky choice? Figure 4 answers these questions by showing that, already from two 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 4 shows the results averaged on all 16 classes in the case study. Thanks to the relatively small number of classes, in Figure 5 and Figure 6 we can show the time analysis for each class individually. The results provide interesting further insight by showing very different behaviors among classes. For example, a peculiar result in Figure 5 and Figure 6 is that, for the best MA, the performance is not monotonically increasing through time (as it is in Figure 4). 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 4), on single classes in isolation everything could in theory happen (Figure 5 and Figure 6).

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

5.7. RQ4: What is the influence of each individual type of local search operator?

Table 5 shows the average coverage achieved for each individual type of local search. To study the effects individually and not conflate them with the effects of seeding, all results shown in the table are based on runs without seeding activated. If applied independently, then the techniques of ensuring double execution and expanding test cases have only a minor effect. However, they can be beneficial for all other types of local search. In the table they are activated for all types of local search. In other words, results presented Table 5 are based only on six configurations out of the $2^8 = 256$ we ran. In all these six configurations, seeding was off, whereas double execution and test expansion were on. In the "Base" configuration, all the five local search operators were off. For each of the remaining five configurations, one local search operator was on, whereas the other four were off.

- `IntArrayWithoutExceptions` benefits mainly from numeric search, and the local search on arrays has no benefit. Indeed, as long as there are explicit assignments to array elements in the tests then numeric local search can improve array contents as well, whereas search on all array elements may waste resources.
- `LinearWithoutOverflow` is a class that consists almost exclusively of numerical constraints, thus numeric search brings the most benefits.
- `FloatArithmetic` represents numeric problems with floating point inputs; numeric search brings the expected improvement.
- `IntArrayWithoutExceptionsWithArrayParameters` repeats the pattern seen in `IntArrayWithtoutExceptions`: search on numbers improved coverage, search on arrays made things worse.

Table 5: Results for **RQ4**. For each class, the table reports the coverage effectiveness of each operator in isolation. Values in bold are the maxima for each class.

Case Study	Base	Primitives	Strings	Arrays	References	Restore
IntArrayWithoutExceptions	0.84	**0.91**	0.84	0.84	0.84	0.83
LinearWithoutOverflow	0.68	**0.98**	0.68	0.68	0.68	0.68
FloatArithmetic	0.48	**0.57**	0.48	0.48	0.48	0.48
IA.WithArrayParameters	0.94	**0.99**	0.93	0.93	0.94	0.92
Cookie	0.33	0.30	**0.94**	0.33	0.30	0.30
DateParse	0.54	0.53	**0.85**	0.54	0.52	0.53
Stemmer	0.56	**0.68**	0.55	0.56	0.56	0.58
Ordered4	**0.99**	**0.99**	0.98	**0.99**	**0.99**	**0.99**
XMLElement	0.86	0.84	0.86	0.87	0.90	**0.94**
CommandLine	**0.97**	0.96	0.96	0.96	**0.97**	**0.97**
Attribute	0.71	0.71	**0.74**	0.72	0.72	0.72
DoubleMetaphone	0.62	0.62	0.64	0.62	0.65	**0.70**
ArrayList	0.88	0.88	0.89	0.88	**0.90**	0.89
Bessj	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**
FastFourierTransformer	0.63	0.63	0.63	0.61	**0.65**	0.61
DateTimeFormat	0.63	0.62	0.71	0.63	0.68	**0.72**

- `Cookie` is a pure string problem, and string local search behaves as expected.
- `DateParse` is also a string problem (which becomes trivial with seeding – see the flat-lined graph in Figure 5); string local search works as expected.
- `Stemmer` is a class that works with text input, yet it takes its input in terms of character arrays and integers. Consequently, string local search does not help, whereas numeric search improves it a lot.
- `Ordered4` is a surprising case: It is a string problem, yet the only type of local search that achieves a worse result than pure global search is search on strings. The reason for this is that the string constraints in this class are based on the `compareTo` method, which returns -1, 0, or 1. While EvoSuite transforms all boolean string comparison operators and replaces them with functions that provide guidance, it currently does not do this for `compareTo`. Consequently, local search on the strings will in many cases not get beyond exploration, which nevertheless consumes search budget.
- `XMLElement` has strings dependencies, yet the are few constraints on these strings; they mainly represent the names of tags. However, some string-related inputs are represented as character arrays (`char[]`), which explains why the array search is more beneficial than the string search for this example. The class has many methods, which is likely why reference search is beneficial, as is restoring coverage.
- Most methods of `CommandLine` have either string or character parameters, which offers potential to apply local search on strings and numbers.

However, again this is a class where the actual values of these strings and characters do not matter, and so these types of local search have a negative effect.

- `Attribute` has several string dependencies, for example one can set a string value for an XML attribute and then call methods to convert it to numbers or booleans. Consequently, local search on strings is beneficial.
- `DoubleMetaphone` has many string related parameters, given that it implements an algorithm to encode strings. String local search has a small benefical effect, as does search on references.
- `ArrayList` has methods with string and numerical inputs, yet only few branches depend on these parameters (e.g., the capacity of an `ArrayList` needs to be larger than 0). Consequently, the only type of local search that has an effect on this class is search on references.
- `Bessj` is a class with many branches on numerical dependencies; however, even with significantly higher search budget EvoSuite is not able to achieve higher coverage than 91%, therefore it is likely that this is already the maximum possible, and none of the types of local search have a negative impact on reaching this.
- `FastFourierTransformer` has many array parameters, yet it seems to perform more transformative calculations on these arrays rather than depending on their content. Consequently, the array local search has a negative effect.
- `DateTimeFormat` has functionality to parse date formatting patterns, and consequently benefits significantly from string local search. It also has many methods, which is reflected in the improvement with reference local search.

Restoring coverage had a negative effect only in five out of the 16 cases, whereas it had a very strong effect in many of them.

RQ4*: Numeric and string local search work well on their relevant problem instances, whereas array search can have negative impact. Reference local search is beneficial for large classes.*

5.8. RQ5: Which combination of local search operators achieves the best results?

There can be subtle effects and interactions between different types of local search. Consequently, for **RQ5** we looked at all possible combinations of the local search operators. Table 6 presents a rank analysis where we list the top 25 configurations with seeding enabled and 25 configurations without seeding. All top ranked configurations restore coverage, most of them apply numeric local search, and most of them apply primitive value expansion (Section 4.2.1).

Table 6: Rank analysis for **RQ5**. Out of the 256 configurations, only the top 25 using seeding and the 25 without are displayed. The table considers whether seeding was employed (Seeding). It also considers the four types of local search operators at test case level: on Primitives (Section 4.1.1), on Strings (Section 4.1.2), on Arrays (Section 4.1.3) and on References (Section 4.1.4). For the test suite level, it considers primitive value expansion (Expand, Section 4.2.1), double execution (Double, Section 4.2.2) and restoring coverage (Restore, Section 4.2.3).

Seeding	Primitives	Strings	Arrays	References	Expand	Double	Restore	Rank	Coverage
√	√			√	√	√	√	45.250	0.883
√	√			√	√		√	45.469	0.883
√	√	√	√	√	√	√	√	50.469	0.885
√	√		√	√	√	√	√	54.344	0.879
√	√		√		√		√	56.812	0.882
√	√		√		√	√	√	58.062	0.882
√	√				√		√	59.188	0.882
√			√				√	60.844	0.864
√	√	√		√	√	√	√	62.250	0.881
√	√	√	√		√		√	63.406	0.882
√			√	√	√		√	65.406	0.875
√	√	√	√		√	√	√	65.469	0.881
√	√		√	√	√		√	65.625	0.880
√	√	√	√	√	√		√	65.812	0.881
√	√	√	√	√		√	√	65.938	0.875
√	√				√	√	√	66.938	0.881
√				√			√	68.562	0.867
√	√	√			√	√	√	68.781	0.881
√	√	√	√	√			√	70.125	0.874
√	√	√		√	√		√	70.312	0.880
√			√			√	√	71.219	0.863
√		√	√	√	√		√	71.562	0.872
√	√			√		√	√	72.719	0.871
√	√		√				√	73.062	0.867
√	√		√			√	√	73.188	0.867
	√	√			√	√	√	94.656	0.851
	√	√	√		√	√	√	103.625	0.847
	√	√		√	√	√	√	106.656	0.843
	√	√	√	√	√	√	√	109.625	0.843
	√	√	√	√	√		√	112.250	0.843
	√	√	√		√		√	116.469	0.841
	√	√			√		√	116.656	0.848
	√	√		√	√		√	116.719	0.842
	√	√	√	√		√	√	122.688	0.835
	√			√	√		√	122.969	0.779
	√		√	√	√		√	124.156	0.777
	√	√		√			√	124.500	0.832
	√		√			√	√	128.031	0.781
	√		√	√	√	√	√	128.156	0.780
	√			√	√	√	√	128.250	0.779
	√			√		√	√	131.562	0.773
	√	√				√	√	132.031	0.836
	√				√	√	√	132.875	0.782
	√	√	√			√	√	133.156	0.832
	√		√		√		√	133.719	0.777
	√		√	√		√	√	135.094	0.776
	√				√		√	135.594	0.777
	√		√		√	√	√	136.406	0.780
	√	√		√		√	√	136.562	0.831
	√	√					√	137.062	0.832

This confirms the intuition that expansion is important to make local search on primitive values effective. The table clearly shows how seeding influences the search, as all seeding configurations are ranked higher than those without seeding. All top ranked configurations without seeding apply numerical local search, whereas there exist some in the seeding ranks that do not use numerical search. The top ranked configurations without seeding use string local search, whereas fewer of the top ranked configurations with seeding use string local search. Indeed, in several of the 16 example classes the string constraints are partially trivially solved with seeding, such that string local search in conjunction with seeding seems to waste resources and has a negative effect. The top ranked configuration without seeding excludes array local search, as one would expect from the analysis of **RQ4**. However, surprisingly reference search is also excluded, whereas in **RQ4** we saw that there were only two cases where reference local search applied individually led to a worse result, suggesting interactions with the other operators. However, the configurations with array search and reference search enabled are ranked directly below that configuration with only marginally lower coverage, suggesting that the impact is only minor. The top ranked configuration with seeding also excludes array local search as expected, but it does include reference local search. However, the configuration with all types of local search enabled ranks third, with even a minimally higher average coverage.

RQ5*: Applying all local search operators leads to good results, although string, array, and reference search can have minor negative effects.*

5.9. RQ6: Does adaptive local search improve the performance?

RQ4 showed how different classes influence the effectiveness of local search. Consequently, instead of applying local search with a fixed configuration, we next consider how doing so in an adaptive way influences results. As described in Section 4.3, we use the adaptive methods introduced by Galeotti et al. (2013). Table 7 shows the top ranked configurations for the combinations of adaptiveness parameters we considered. Again the configurations using seeding are ranked higher than those without. Applying local search selectively, i.e., only on statements that led to a fitness change after the last mutation, is not included in the top configurations. The likely reason for this is that this optimization is designed for a scenario where DSE is applied to the primitive values in a test suite (cf. (Galeotti et al., 2013)) and will thus only select some of the cases where local search can lead to an improvement. As seen in the discussion of **RQ4**, local search operators that are not directly related to primitive values can still have a strong positive influence on the performance, and these would not benefit from this selective strategy. The probabilities in the top ranks confirm the results of **RQ1**, where in the best configuration local search was applied every 100 generations. With a probability of 0.01, on average local search will also be

Table 7: Rank analysis for **RQ6**. Out of the 224 configurations, only the top 25 using seeding and the 25 without are displayed.

Seeding	Selective	Rate	Probability	Rank	Coverage
√		10.0	0.01	42.688	0.881
√		5.0	0.01	45.812	0.877
√		1.5	0.02	47.594	0.877
√		10.0	0.02	48.219	0.880
√	√	1.0	0.05	51.750	0.868
√		1.0	0.01	52.781	0.876
√	√	1.2	0.01	52.781	0.868
√	√	10.0	0.01	54.031	0.866
√	√	1.2	0.05	54.500	0.865
√		5.0	0.02	55.219	0.875
√	√	1.5	0.05	56.156	0.867
√	√	1.7	0.05	56.656	0.866
√		2.0	0.01	57.156	0.873
√		1.7	0.01	59.344	0.874
√	√	1.7	0.01	59.531	0.866
√	√	5.0	0.01	60.500	0.865
√		1.5	0.01	60.781	0.877
√	√	1.0	0.10	62.594	0.864
√	√	2.0	0.01	62.688	0.865
√		1.2	0.01	63.125	0.869
√		10.0	0.05	63.250	0.875
√	√	1.7	0.10	65.781	0.861
√		1.7	0.02	67.031	0.874
√	√	10.0	0.05	67.156	0.860
√	√	2.0	0.05	67.500	0.862
		1.2	0.01	80.750	0.853
		1.7	0.01	85.625	0.845
		1.5	0.01	86.594	0.850
		1.5	0.02	87.469	0.854
		10.0	0.02	88.438	0.842
		1.0	0.01	93.812	0.845
		5.0	0.01	95.375	0.840
		5.0	0.02	95.625	0.841
		2.0	0.01	98.656	0.842
	√	1.0	0.01	98.656	0.774
		1.7	0.05	99.531	0.838
		1.7	0.02	102.500	0.842
		10.0	0.05	102.906	0.838
	√	1.0	0.10	103.250	0.804
		1.2	0.02	103.781	0.839
		10.0	0.01	105.438	0.836
		2.0	0.05	105.500	0.839
		1.0	0.02	106.062	0.840
		1.5	0.05	106.406	0.832
		2.0	0.02	111.156	0.839
		1.5	0.10	111.281	0.833
		1.0	0.05	111.875	0.834
	√	1.0	0.20	112.219	0.801
	√	1.7	0.01	113.250	0.769
		1.2	0.10	113.312	0.830

Table 8: For each class, comparisons without seeding of Base GA configuration with best Non-Adaptive MA from Table 6 and with best Adaptive MA from Table 7. Effect sizes $\hat{A}_{12}$ are calculated for when Non-Adaptive is compared with Base ($\hat{A}_{nb}$), and Adaptive compared to Base ($\hat{A}_{ab}$) and to Non-Adaptive ($\hat{A}_{an}$). Effect sizes that are statistically significant at 0.05 level are in bold.

Case Study	Base	Non-Adaptive	$\hat{A}_{nb}$	Adpative	$\hat{A}_{ab}$	$\hat{A}_{an}$
IntArrayWithoutExceptions	0.85	0.92	**1.00**	0.93	**1.00**	0.65
LinearWithoutOverflow	0.69	0.98	**1.00**	0.99	**1.00**	0.65
FloatArithmetic	0.49	0.59	**1.00**	0.63	**1.00**	0.62
IA.WithArrayParameters	0.98	1.00	0.67	1.00	0.67	0.50
Cookie	0.26	0.95	**1.00**	0.98	**1.00**	0.61
DateParse	0.55	0.89	**1.00**	0.91	**1.00**	0.58
Stemmer	0.59	0.70	**1.00**	0.70	**1.00**	0.48
Ordered4	0.92	1.00	**0.99**	1.00	**1.00**	0.55
XMLElement	0.91	0.95	**0.89**	0.91	0.43	**0.21**
CommandLine	0.96	0.96	0.45	0.97	0.54	0.59
Attribute	0.72	0.76	**0.99**	0.78	**1.00**	0.69
DoubleMetaphone	0.63	0.72	**1.00**	0.71	**0.98**	0.44
ArrayList	0.90	0.90	0.56	0.88	0.33	0.29
Bessj	0.91	0.91	0.50	0.91	0.50	0.50
FastFourierTransformer	0.66	0.64	0.44	0.61	0.40	0.45
DateTimeFormat	0.69	0.74	**1.00**	0.76	**1.00**	0.74

applied every 100 generations. The best configuration shows a high adaptation rate of 10, followed by the second best configuration with the second highest configuration rate used in the experiment. Consequently, we can conclude that adaptation is an important factor in achieving high coverage. To compare the results between the adaptive configurations, the GA, and the tuned MA, Table 8 summarizes the coverage and $\hat{A}_{12}$ for each pair of configurations. To show the effects of adaptiveness more clearly without interference of other optimizations, this table shows the results without seeding. Note that in contrast to Table 3, the non-adaptive configuration is for two minutes of search time using the best configuration of local search operators as obtained in **RQ5**. For this particular configuration, the non-adaptive MA is significantly better than the GA in 11 out of the 16 cases. Interestingly, it is even better on `XMLElement`, whereas in **RQ1** the MA showed a slightly worse result after 10 minutes using all local search operators. Comparing the adaptive MA to GA shows significantly better results in 10 out of 16 cases, but interestingly slightly worse results in `ArrayList` and `FastFourierTransformer`, although not significant in any of the cases. The adaptive MA achieves higher average coverage than the non-adaptive tuned MA in eight cases, although none of them are statistically significant, and the coverage loss in `XMLElement` is statistically significant (however, on average it is still the same as the base GA). Thus, on average the adaptive configuration is only slightly better than the best fixed configuration (average coverage of 85.44% for adaptive vs. 85.06% for tuned fixed configuration, and the average effect size is 0.53). However, the implementation of adaptiveness used in these experiments

Table 9: Comparison of results of default GA with best MA for both SF100 and Carfast case studies. The average number of covered branches is reported, and the difference between the two configurations.

Case Study	Classes	Total	GA	MA	Diff.
Carfast	1,392	1,181,234	513,669	558,521	44,853
SF100	11,088	238,760	93,600	94,240	639

is of course rather simplistic, and ideally one would apply adaptiveness also to the choice of operators. With this in mind, and considering that adaptive configurations have a higher chance of generalising to new classes, it is fair to assume that adaptivity in local search is beneficial in the general case.

RQ6*: In our experiments, basic adaptive control improved performance slightly, but the improvement is more likely to generalize than the fixed configuration.*

5.10. RQ7: Do results generalize to other classes?

All experiments so far were conducted on 16 classes selected under the assumption that they are representative of difficult search problems. However, there remains the question on how these findings generalize (**RQ7**). To answer this question, we take the overall best configuration of local search, and apply EvoSuite with that configuration to two different benchmarks. The SF100 corpus of classes is a random sample of 100 SourceForge open source projects. A particular aspect of this real-life, unbiased sample of classes is that the problems it represents are quite different to those considered as difficult search problems (Fraser and Arcuri, 2012b): For example, a large share of the classes have environmental dependencies that make high coverage with EvoSuite impossible. In contrast, the Carfast (Park et al., 2012) case study is devoid of such environmental dependencies, but still consists of a set of automatically generated software projects that are intended to be realistic. We applied EvoSuite on both benchmarks for two minutes per class with 10 iterations to accommodate for randomness. Table 9 summarizes the results: On the CarFast benchmark, the use of local search covers on average 44,853 more branches than pure global search. On SF100 the increase is smaller; 639 additional branches were covered by the memetic algorithm. This is not unexpected; SF100 consists of many trivial classes and many branches cannot be covered until the test generator can handle the environmental dependencies, so the potential for improvement is smaller in the first place.

RQ7*: The improvements with local search generalize to other classes, but in practice other technical problems may be prevalent to pure search problems.*

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 the generated 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 10 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 (in the order of hundreds of days of computational resources), we only used 16 classes for our in depth evaluations. Those classes were manually chosen. To reduce this threat to validity, we also carried out a set of experiments with best found settings on the SF100 corpus, which is a random selection of 100 projects from SourceForge. We also carried out further experiments on a large case study (Carfast) previously used in the literature.

7. Conclusions

The EvoSuite tool applies Genetic Algorithms to the problem of generating unit-level test suites for Java classes with high branch coverage. However, genetic mutations on particular parts of the test cases tend to be undirected. This means that for variables of primitive types, strings, and arrays, small adjustments needed for certain branches to be covered are unlikely to occur. This paper therefore defined a series of local search operators, extending the Genetic Algorithm used in EvoSuite to a Memetic Algorithm. Although Memetic Algorithms have already been used in the past for unit test generation, this paper is the first to provide a comprehensive approach for object-oriented software, targeting whole test suites, handling different kinds of test data like strings and arrays. Our empirical study shows that, using these local search operators, branch coverage of classes may be significantly improved, in some cases even up by 53%. A sound evaluation on more than 12,000 Java classes confirms the results are of practical value for practitioners. Adding an adaptive parameter control technique showed improvements in our experiments. However, the technique we applied in our experiments was simple, and there is potential for further improvements using more advanced parameter control techniques (Eiben et al., 1999). 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 EPSRC projects "EXOGEN" (EP/K030353/1) and "RE-COST" (EP/I010386), and the Norwegian Research Council.

References

Alshraideh, M., Bottaci, L., 2006. Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification, and Reliability 16, 175–203.

Arcuri, A., 2009. Theoretical analysis of local search in software testing, in: Symposium on Stochastic Algorithms, Foundations and Applications (SAGA), pp. 156–168.

Arcuri, A., 2013. It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (STVR) 23, 119–147.

Arcuri, A., Briand, L., 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) 24, 219–250.

Arcuri, A., Fraser, G., 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering (EMSE) , 1–30.

Arcuri, A., Yao, X., 2007. A memetic algorithm for test data generation of object-oriented software, in: IEEE Congress on Evolutionary Computation (CEC), pp. 2048–2055.

Baresi, L., Lanzi, P.L., Miraz, M., 2010. Testful: an evolutionary test approach for java, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 185–194.

Eiben, A.E., Hinterding, R., Michalewicz, Z., 1999. Parameter control in evolutionary algorithms. IEEE Trans. on Evolutionary Computation 3, 124–141.

El-Mihoub, T.A., Hopgood, A.A., Nolle, L., Battersby, A., 2006. Hybrid genetic algorithms: A review. Engineering Letters 13, 124–137.

Ferguson, R., Korel, B., 1996. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology 5, 63–86.

Fraser, G., Arcuri, A., 2011. EvoSuite: Automatic test suite generation for object-oriented software., in: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 416–419.

Fraser, G., Arcuri, A., 2012a. The seed is strong: Seeding strategies in search-based software testing, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 121–130.

Fraser, G., Arcuri, A., 2012b. Sound empirical evidence in software testing, in: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 178–188.

Fraser, G., Arcuri, A., 2013a. Handling test length bloat. Software Testing, Verification and Reliability (STVR) DOI:10.1002/stvr.1495.

Fraser, G., Arcuri, A., 2013b. Whole test suite generation. IEEE Transactions on Software Engineering 39, 276–291.

Fraser, G., Arcuri, A., McMinn, P., 2013. Test suite generation with memetic algorithms, in: Genetic and Evolutionary Computation Conference (GECCO), pp. 1437–1444.

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

Galeotti, J.P., Fraser, G., Arcuri, A., 2013. Improving search-based test suite generation with dynamic symbolic execution, in: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 360–369.

Gendreau, M., Potvin, J.Y.E., 2010. Handbook of Metaheuristics. International Series in Operations Research & Management Science, Vol. 146. 2nd ed., Springer.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M., 2004. Testability transformation. IEEE Transactions on Software Engineering 30, 3–16.

Harman, M., McMinn, P., 2007. 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), pp. 73–83.

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

Kapfhammer, G., McMinn, P., Wright, C.J., 2013. Search-based testing of relational schema integrity constraints across multiple database management systems, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 31–40.

Korel, B., 1990. Automated software test data generation. IEEE Transactions on Software Engineering , 870–879.

Li, Y., Fraser, G., 2011. Bytecode testability transformation, in: Search Based Software Engineering. Springer. volume 6956 of *Lecture Notes in Computer Science*, pp. 237–251.

Liaskos, K., Roper, M., 2008. Hybridizing evolutionary testing with artificial immune systems and local search, in: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, IEEE Computer Society, Washington, DC, USA. pp. 211–220.

McMinn, P., 2004. Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14, 105–156.

Mitchell, T., 1997. Machine Learning. McGraw Hill.

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

Nakagawa, S., 2004. A farewell to Bonferroni: the problems of low statistical power and publication bias. Behavioral Ecology 15, 1044–1045.

Park, S., Hossain, B.M.M., Hussain, I., Csallner, C., Grechanik, M., Taneja, K., Fu, C., Xie, Q., 2012. Carfast: achieving higher statement coverage faster, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA. pp. 35:1–35:11. doi:10.1145/2393596.2393636.

Perneger, T., 1998. What's wrong with Bonferroni adjustments. British Medical Journal 316, 1236–1238.

Rodriguez-Tello, E., Torres-Jimenez, J., 2010. Memetic algorithms for constructing binary covering arrays of strength three, in: Proceedings of the 9th International Conference on Artificial Evolution, Springer-Verlag, Berlin, Heidelberg. pp. 86–97.

Tonella, P., 2004. Evolutionary testing of classes, in: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 119–128.

Wang, H.C., Jeng, B., 2006. Structural testing using memetic algorithm, in: Proceedings of the Second Taiwan Conference on Software Engineering.

Whitley, D., Gordon, V.S., Mathias, K., 1994. Lamarckian evolution, the baldwin effect and function optimization, in: Parallel Problem Solving from Nature-PPSN III. Springer, pp. 5–15.

Yao, X., Wang, F., Padmanabhan, K., Salcedo-Sanz, S., 2005. Hybrid evolutionary approaches to terminal assignment in communications networks, in: Recent Advances in Memetic Algorithms. Springer, pp. 129–159.