Noname manuscript No.
(will be inserted by the editor)

Parameter Tuning or Default Values? An Empirical Investigation in Search-Based Software Engineering [*]

Andrea Arcuri and Gordon Fraser

Received: date / Accepted: date

Abstract Many software engineering problems have been addressed with search algorithms. Search algorithms usually depend on several parameters (e.g., population size and crossover rate in genetic algorithms), and the choice of these parameters can have an impact on the performance of the algorithm. It has been formally proven in the No Free Lunch theorem that it is impossible to *tune* a search algorithm such that it will have optimal settings for all possible problems. So, how to properly set the parameters of a search algorithm for a given software engineering problem? In this paper, we carry out the largest empirical analysis so far on parameter tuning in search-based software engineering. More than one million experiments were carried out and statistically analyzed in the context of test data generation for object-oriented software using the EVOSUITE tool. Results show that tuning does indeed have impact on the performance of a search algorithm. But, at least in the context of test data generation, it does not seem easy to find good settings that significantly outperform the "default" values suggested in the literature. This has very practical value for both researchers (e.g., when different techniques are compared) and practitioners. Using "default" values is a reasonable and justified choice, whereas parameter tuning is a long and expensive process that might or might not pay off in the end.

Keyword: Search-based software engineering, test data generation, object-oriented, unit testing, tuning, EvoSuite, Java, response surface, design of experiments

[*] This paper is an extension of a conference paper published in the International Symposium on Search Based Software Engineering (SSBSE) [4], 2011.

A. Arcuri
Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
E-mail: arcuri@simula.no

G. Fraser
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
E-mail: Gordon.Fraser@sheffield.ac.uk

1 Introduction

Recent years have brought a large growth of interest in search-based software engineering (SBSE) [19], especially in software testing [1]. The field has even matured to a stage where industrial applications have started to appear [5, 36]. One of the key strengths of SBSE leading to this success is its ability of automatically solving very complex problems where exact solutions cannot be deterministically found in reasonable time. However, to make SBSE really usable in practice, no knowledge of search algorithms should be required from practitioners who want to use it, as such knowledge is highly specialized and might not be widespread. In other words, SBSE tools should be treated as "black boxes" where the internal details are hidden, otherwise technology transfer to industrial practice will hardly be feasible.

One of the main barriers to the use of a search algorithm in SBSE is *tuning*. A search algorithm can have many parameters that need to be set. For example, to use a genetic algorithm, one has to specify the population size, type of selection mechanism (e.g., roulette wheel, tournament, rank-based), type of crossover (e.g., single point, multi-point), crossover probability, type and probability of mutation, type and rate of elitism, etc. The choice of all these parameters might have a large impact on the performance of a search algorithm. In the worst case, an "unfortunate" parameter setting might make it impossible to solve the problem at hand.

Is it possible to find an *optimal* parameter setting, to solve this problem once and for all? Unfortunately, this is not possible, and this has been formally proven in the *No Free Lunch* (NFL) theorem [39]: On average, all algorithms perform equally on *all* possible problems. For any problem an algorithm is good at solving, there always exist problems for which that algorithm has worse performance than other algorithms. Because the same algorithm with different parameter settings can be considered as a family of different algorithms, the NFL theorem applies to tuning as well. However, the NFL is valid only when *all* possible search problems are considered. SBSE only represents a subset of all possible problems, so it could be possible to find "good" parameter settings that work well for this subset. Such a known good configuration is important when handing tools over to practitioners, because it is not reasonable to expect them to tune such tools as that would require deep knowledge of the tools and of search algorithms in general. Similarly, it is also important from a research perspective to avoid skewing results with improper parameter settings.

In this paper, we present the results of the largest empirical analysis of tuning in SBSE to date to address the question of parameter tuning. We chose the scenario of test data generation at unit test level because it is one of the most studied problems in SBSE [19]. In particular, we consider test data generation for object-oriented software using the EVOSUITE tool [13], where the goal is to find the minimal test suite that maximizes branch coverage (having a small test suite is important when no automated oracles are available and results need to be manually checked by software testers). Because an empirical study of the size conducted in this paper is extremely time consuming, it is necessary to focus on one particular problem. In principle the experiments could also be applied to other SBSE problems (e.g., regression testing [20]). However, the choice of test data generation allowed us to use an advanced SBSE tool like EVOSUITE. By using EVOSUITE on the SF100 corpus [16], we can have enough confidence to generalize our results to open source software, which is of high value for practitioners, rather than drawing conclusions from toy problems or small case studies. However, we would like to point out that the complexity of tuning in search-based test data generation, as witnessed in our experiments, does not mean that alternative approaches to test data generation not based on search are free of such problems. For example, in dynamic symbolic execution there are parameters such as the exploration strategy or the parameters of the employed constraint solver, and these parameters also need to be tuned. In general, this applies to any software engineering activity in which parameters need to be set.

We chose to consider several parameter settings (e.g., population size and crossover rate). To make the experiments finish in a feasible amount of time, we had three different sets of classes for three different kinds of analyses. In the first, more exhaustive analysis, we only considered 20 software classes as case study. In the second more focused analysis, we had 609 classes randomly sampled from the SF100 corpus [16], which is a sample of 100 projects randomly selected from the open source software repository SourceForge. The last case study is based on only two classes, which are used to get more insight on some follow up questions coming from the analysis of the second case study. In the end, all of these empirical analyses led to more than *one million* experiments which took weeks to run, even on a cluster of computers.

Although it is well known that parameter tuning has impact on the performance of search algorithms, there is little empirical evidence in the literature of SBSE that tries to quantify its effects. The results of the large empirical analysis presented in this paper provide compelling evidence that parameter tuning is indeed critical, but "default" values in the literature already perform relatively well. On one hand, it makes sense to carry out a large tuning phase before releasing a SBSE tool to practitioners. On the other hand, tuning might become very expensive, but the improvements caused by it may not be so large. So, for example, for a researcher that faces time/resource constraints it might be justified to skip the tuning phase, if this allows him/her to use a larger case study or to compare more algorithms/techniques.

Another problem related to tuning that is often ignored is the *search budget*. A practitioner might not want to deal with the choice of a genetic algorithm population size, but the choice of the computational time (i.e., how long she/he is willing to wait before the tool gives an output) is something that has a strong impact on tuning. To improve performance, tuning should be a function of the search budget, as we will discuss in more details in the paper.

This paper is organized as follows. Section 2 discusses related work on tuning. The analyzed search algorithm (a genetic algorithm used in EvoSuite) is presented in Section 3 with a description of the parameters we investigate with respect to tuning. Section 4 presents the case studies and the empirical analyses. Guidelines on how to handle parameter tuning are discussed in Section 5. Threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

The problem of tuning is not only restricted to software engineering. Search algorithms can be applied to many other fields where optimizations are sought. For example, there are many successful applications of search algorithms in evolutionary computation (EC) [38]. On one hand, from the point of view of tuning, it might be that problems in the SBSE field share some commonalities among them. On the other hand, it may be that SBSE problems like test data generation are "closer" to other problems in EC rather than in SBSE. A cross-filed meta-analysis of parameter tuning would provide answers to these research questions, but it is not in the scope of this paper, as we focus on providing empirical evidence for test data generation. Whether our results carry over to other applications and fields will be an important future work.

2.1 Related Studies

Eiben *et al.* [11] presented a survey on how to control and set parameter values of evolutionary algorithms. In their survey, several techniques are discussed. Of particular interest is the distinction between *parameter tuning* and *parameter control*: The former deals with how to choose parameter values *before* running a search algorithm. For example, should we use a population size of 50 or 100? In contrast, parameter control deals with how to change parameter values *during* the run of a search algorithm. A particular value that is good at the beginning of the search might become sub-optimal in the later stages. For example, in a genetic algorithm one might want to have a high mutation rate (or large population size) at the beginning of the search, and then decrease it in the course of the evolution; this would be conceptually similar to temperature cooling in simulated annealing. A similar concept is to use feedback from the search to change the parameter values. For example, a search operator that generates a solution with higher fitness during the search might be rewarded by increasing its probability of occurrence, and penalized by decreasing it otherwise.

In this paper we only deal with parameter tuning. Parameter control is a promising area of research, and although there are some applications in SBSE (e.g., [27, 29]), it is not widely employed. It is important to note that parameter control does not completely solve the problems of parameter tuning. In fact, parameter control techniques usually lead to a higher number of parameters that need to be tuned. For example, if we want decrease the population size during the search, not only we still need to choose a starting value, but also we need to define by how much (e.g., 5%) it should be decreased (new parameter) and how often (e.g., every five generations) we decrease it (yet another new parameter). Even if we do not make these choices in a fixed way before the search and we rather use a feedback mechanism to adapt them at runtime, we still end up with a new set of parameters related to how to exploit such feedback information. Although parameter control usually results in more parameters that need to be tuned, empirical studies show that such an approach pays off in the end [11]. However, not everyone agrees on this point [10], and further research (especially in SBSE) is necessary.

Recently, Smit and Eiben [34] carried out a series of experiments on parameter tuning. They consider the tuning of six parameters of a genetic algorithm applied to five numerical functions, comparing three settings: a default setting based on "common wisdom", the best tuning averaged on the five functions (which they call *generalist*), and the best tuning for each function independently (*specialist*). Only one fixed search budget (i.e., maximum number of fitness evaluations as stopping criterion) was considered. Our work shares some commonalities with these experiments, but more research questions and larger empirical analysis are presented in this paper (details will be given in Section 4).

In order to find the best parameter configuration for a given case study, one can run experiments with different configurations, and then the configuration that gives the highest results on average can be identified as best for that case study. However, evaluating all possible parameter combinations is infeasible in practice. Techniques to select only a subset of configurations to test that have high probability of being optimal exist, for example regression trees (e.g., used in [6, 8]) and response surface methodology (e.g., used in [12, 27]). However, as for any complex heuristics, such techniques might fail in some contexts, and proper care needs to be taken to apply them properly on the problem at hand.

2.2 Practical Considerations

The goal of this paper is to study the effects of parameter tuning, which includes also the cases of sub-optimal choices. To obtain accurate results, this type of analysis requires large empirical evaluations. This is done only for the sake of answering research questions (as for example to study the effects of a sub-optimal tuning). In general, a practitioner would be interested only in the best configuration for his problem at hand.

If a practitioner wants to use a search algorithm on an industrial problem (not necessarily in software engineering) that has not been studied in the literature, then she/he would need to tune the algorithm by herself, as default settings might lead to poor performance. To help practitioners in making such tuning, there exist frameworks such as GUIDE [9]. The scope of this paper is different: we tackle *known* SBSE problems (e.g., test data generation for object-oriented software). For known problems, it is possible to carry out large empirical analyses in laboratory settings.

There might be cases in which, even on known problems, it might be useful to let the practitioners perform/improve tuning (if they have enough knowledge about search algorithms), and tools like EvoTest support this [36]. As an example, an SBSE problem instance type might need to be solved several times (e.g., a software system that is slightly modified during time). Another example could be to do tuning on a sub-system before tackling the entire system (which for example could be millions of lines of code). Whether such cases occur in practice, and whether the tuning can be safely left to practitioners, would require controlled empirical studies in industrial contexts. As such empirical evidence is currently lacking in the literature of SBSE, we can assume that parameter tuning is needed before releasing SBSE tool prototypes.

3 Search Algorithm Setting

We performed our experiments in a domain of test generation for object-oriented software using genetic algorithms. In this domain, the objective is to derive test suites (sets of test cases) for a given class, such that the test suite maximizes a chosen coverage criterion while minimizing the number of tests and their length. A test case in this domain is a sequence of method calls that constructs objects and calls methods on them. The resulting test suite is presented to the user, who usually has to add test oracles that check for correctness when executing the test cases.

3.1 Representation

The test cases may have variable length [2], and so earlier approaches to testing object-oriented software made use of method sequences [17, 35] or strongly typed genetic programming [30,37]. In our experiments, we used the EvoSuite [13] tool, in which one individual is an entire test suite of variable size. The entire search space of test suites is composed of all possible test suites of sizes from 1 to a predefined maximum N. Each test case can have a size (i.e., number of statements) from 1 to L. For each position in the sequence of statements of a test case, there can be up to I_{max} possible statements, depending on the SUT and the position within the test case (later statements can reuse objects instantiated in previous statements). The search space is hence extremely large, although finite because N, L and I_{max} are finite.

3.2 Search Operators

Crossover between test suites generates two offspring O_1 and O_2 from two parent test suites P_1 and P_2. A random value r is chosen from $[0,1]$, and the first offspring O_1 contains the first $r|P_1|$ test cases from the first parent, followed by the last $(1-r)|P_2|$ test cases from the second parent. The second offspring O_2 contains the first $r|P_2|$ test cases from the second parent, followed by the last $(1-r)|P_1|$ test cases from the first parent.

The mutation operator for test suites works both at test suite and test case levels: When a test suite $\mathcal{T}$ is mutated, each of its test cases is mutated with probability $1/|\mathcal{T}|$. Then, with probability $\sigma = 0.1$, a new test case is added to the test suite. 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 $1 - \sigma^i$). Test cases are added only if the limit N has not been reached.

When a test case is chosen to be mutated, we apply a number of mutations at random in between 1 and m, for some constant m (which is a parameter that needs to be tuned). For each of these mutations on a test case (which are applied sequentially), we apply three different operations with probability $1/3$ in order: remove, change and insert.

When removing statements out of a test case of length l, each statement is removed with probability $1/l$. Removing a statement might invalidate dependencies within the test case, which we attempt to repair; if this repair fails, then dependent statements are also deleted. When applying the change mutation, each statement is changed with probability $1/l$. A change means it is replaced with a different statement that retains the validity of the test case; e.g., a different method call with the same return type. When inserting statements, we first insert a new statement with probability $\sigma' = 0.5$ at a random position. If it is added, then a second statement is added with probability σ'^2, and so on until the ith statement is not inserted. If after applying these mutation operators a test case t has no statement left (i.e., all have been removed), then t is removed from $\mathcal{T}$.

In EVOSUITE, we also consider a set of *seeding* strategies [15]. For example, we can statically analyze the byte-code of the SUT and collect all the constants in it (e.g., numbers and strings), and store them in a pool of constants. Every time we need to sample a number/string at random (e.g., during population initialization and mutations), with probability P_s we can rather take one constant (of the appropriate kind) from that pool. This has been shown to be a very effective technique, particularly for SUTs using string objects.

When we start the search, the initial population of test cases is generated randomly, by repeatedly performing the insertion operator also used to mutate test cases.

3.3 Fitness Function

The search objective we chose is branch coverage at the byte-code level, which requires that a test suite exercises a program in such a way that every condition (e.g., if, while) evaluates to true and to false. The fitness function is based on the well-established branch distance [24], which estimates the distance towards a particular evaluation of a branch predicate. The overall fitness of a test suite with respect to all branches is measured as the sum of the normalized branch distances of all branches in the program under test. Using a fitness function that considers all the testing targets at the same time has been shown to lead to better results than the common strategy of considering each target individually [13]. Such an approach is particularly useful to reduce the negative effects of infeasible targets for the search. Furthermore, we applied several bloat control techniques [14] to avoid that the size of individuals becomes bloated during the search.

4 Experiments

In this paper, we present the results of three different sets of experiments using three different sets of classes as case studies.[1] The design of the first and second case study were independent. The first case study is much smaller than the second, which allowed a more extensive analysis on different properties of parameter tuning. The third case study was designed after the analysis of the second case study, and serves to get more insight on questions arising from the second case study. In the following, we

[1] All experimental data is available at http://www.evosuite.org/experimental-data/

start from describing the first case study and its related research questions, followed by the second and then the third case study.

4.1 First Case Study – Exhaustive Analysis

4.1.1 Experimental Setup

Our first case study is a subset of 20 Java classes out of those previously used to evaluate EvoSuite [13]. In choosing the case study, we tried to balance the different types of classes: historical benchmarks, data structures, numerical functions, string manipulations, classes coming from open source applications and industrial software. Apart from historical benchmarks, our criterion when selecting individual classes was that classes are non-trivial, but EvoSuite may still achieve high coverage on them, to allow for variation in the results. We therefore selected classes where EvoSuite used up its entire search budget without achieving 100% branch coverage, but still achieved more than 80% coverage.

4.1.2 Considered Parameters

In the experiments presented in this section, we investigated five parameters of the search, which are not specific to the test-data generation domain. The first parameter is the *crossover rate*: Whenever two individuals are selected from the parent generation, this parameter specifies the probability with which they are crossed over. If they are not crossed over, then the parents are passed on to the next stage (mutation), else the offspring resulting from the crossover are used at the mutation stage.

The second parameter is the *population size*, which determines how many individuals are created for the initial population. The population size does not change in the course of the evolution, i.e., reproduction ensures that the next generation has the same size as the initial generation.

The third parameter is the *elitism rate*: Elitism describes the process that the best individuals of a population (its elite) automatically survive evolution. The elitism rate is sometimes specified as a percentage of the population that survives, or as the number of individuals that are copied to the next generation. For example, with an elitism rate set to 1 individual, the best individual of the current population is automatically copied to the next generation. In addition, it is still available for reproduction during the normal selection/crossover/mutation process.

In a standard genetic algorithm, elitism, selection and reproduction is performed until the next population has reached the desired population size. A common variant is *steady state* genetic algorithms, in which after the reproduction the offspring replace their parents in the current population. As the concept of elitism does not apply to steady state genetic algorithms, we treat the steady state genetic algorithm as a special parameter setting of the elitism rate.

The fourth parameter is the *selection mechanism*, which describes the algorithm used to select individuals from the current population for reproduction. In roulette wheel selection, each individual is selected with a probability that is proportionate to its fitness (hence it is also known as fitness proportionate selection). In tournament selection, a number of individuals are uniformly selected out of the current population, and the one with the best fitness value is chosen as one parent for reproduction. The *tournament size* denotes how many individuals are considered for the "tournament". Finally, rank selection is similar to roulette wheel selection, except that the probability of an individual being selected is not proportionate to its fitness but to its rank when ranking individuals according to their fitness. The advantage of this approach over roulette wheel selection is that the selection is not easily dominated by individuals that are fitter than others, which would lead to premature convergence. The probability of a ranking position can be weighted using the *rank bias* parameter.

Finally, the fifth parameter we consider is whether or not to apply a *parent replacement check*. When two offspring have been evolved through crossover and mutation, checking against the parents means that the offspring survive only if at least one of the two offspring has a better fitness than their parents. If this is not the case, the parents are used in the next generation instead of the offspring.

We investigated the following values for the chosen five parameters:

- Crossover rate: $\{0\ ,\ .2\ ,\ .5\ ,\ .8\ ,\ 1\}$.
- Population size: $\{4\ ,\ 10,\ 50\ ,\ 100\ ,\ 200\}$.
- Elitism rate: $\{0\ ,\ 1,\ 10\%\ ,\ 50\%\}$ or steady state.
- Selection: roulette wheel, tournament with size either 2 or 7, and rank selection with bias either 1.2 or 1.7.
- Parent replacement check (activated or not).

Notice that the search algorithm used in EvoSuite has many other parameters to tune. Because the possible number of parameter combinations is exponential in the number of parameters, only a limited number of parameters and values could be used. For the evaluation we chose parameters that are common to most genetic algorithms, and avoided parameters that are specific in EvoSuite to handle object-oriented software. Regarding the actual values chosen for experimentation, we considered common values in the literature and, for each parameter, we tried to have both "low" and "high" values. In general there is an infinite number of possible values, and any choice done for experimentation is bound to be somewhat arbitrary.

Because the goal of this paper is to study the effects of tuning, we analyzed all the possible combinations of the selected parameter values. In contrast, if one is only interested in finding the "best" tuning for the case study at hand, techniques such as the response surface methodology could be used to reduce the number of configurations to evaluate. We will discuss such methodology in more details later on in Section 4.2 when we will present the second case study (it was not used for this first case study [4]).

4.1.3 Search Budget

An important factor in an SBSE tool is the *search budget*, i.e., when to stop the search, as it cannot be assumed that an optimal solution is always found. A search algorithm can be run for an arbitrary amount of time – for example, a practitioner could run a search algorithm for one second only, or for one hour. However, the search budget has a strong effect on parameter tuning, and it is directly connected to the concept of *exploration* and *exploitation* of the search landscape. For example, the choice of a large population size puts more emphasis on the exploration of the search landscape, which could lead to a better escape from local optima. However, a large population may also slow down the convergence to global optima when not so many local optima are present. If one has a small search budget, it would be advisable to use a small population size because with a large population only few generations would be possible. Therefore, parameter tuning is strongly correlated to the search budget.

We believe that the search budget is perhaps the most important (and maybe only) parameter a practitioner should set. Although most practitioners might not know the internal details of how a search algorithm works and the role of each of its different parameters, they would know for how long they are willing to wait before getting results. A realistic scenario might be the following: During working hours and development, a software engineer would have a small budget (in the order of seconds/minutes) for search, as coding and debugging would take place at the same time. However, a search might also be left running overnight, and results collected the morning after. In these two situations, the parameter settings (e.g., population size) should be different.

The search budget can be expressed in many different formats, for example, in terms of the time that the search may execute. A common format, often used in the literature to allow better and less biased comparisons, is to limit the number of fitness evaluations. In our setting, the variable size of individuals means that comparing fitness evaluations can be meaningless, as one individual can be very short and another one can be very long. Therefore, in this setting (i.e., test data generation for object-oriented software) we rather count the number of statements executed. In the experiments described in this section, we considered a budget of $100{,}000$ function call executions (considering the number of fitness function evaluations would not be fair due to the variable length of the evolved solutions). We also considered the cases of a budget that is a tenth ($10{,}000$) and ten times bigger ($1{,}000{,}000$).

4.1.4 Experiment Procedure

Experiments were performed on a cluster of computers, which has roughly 80 nodes, each with 8 computing cores and 8 gigabytes of memory running a Linux operating system. During the experiments, the cluster was shared with other researchers in the same research institute. To run EvoSuite on the cluster, we simply used its "command line" version (rather than its Eclipse plug-in). For each experiment (or group of), we had shell-scripts calling EvoSuite with the chosen parameters (given as command line inputs to EvoSuite). Outputs were re-directed to local files, that were gathered and analyzed at the end when all experiments were finished.

For each class in the case study, we ran each combination of parameter settings and search budget. All experiments were repeated 15 times to take the random nature of these algorithms into account. Therefore, in total we had $20 \times 5^4 \times 2 \times 3 \times 15 = 1{,}125{,}000$ experiments. Parameter settings were compared based on the achieved coverage. Notice that, in testing object-oriented software, it is also very important to take the size of the generated test suites into account. However, for simplicity, in this paper we only consider coverage, in particular branch coverage at the byte-code level.

Using the raw coverage values for parameter setting comparisons would be too noisy. Most branches are always covered regardless of the chosen parameter setting, while many others are simply infeasible. Given b the number of covered branches in a run for a class c, we used the following normalization to define a *relative coverage* r:

$$r(b,c) = \frac{b - min_c}{max_c - min_c} \ ,$$

where min_c is the worst coverage obtain in *all* the $56{,}250$ experiments for that class c, and max_c is the maximum obtained coverage. If $min_c == max_c$, then $r = 1$.

To analyze all these data in a sound manner, we followed the guidelines in [3]. Statistical difference is measured with the Mann-Whitney U-test, whereas effect sizes are measured with the Vargha-Delaney $\hat{A}_{12}$ statistics. The $\hat{A}_{12}$ statistics measures the probability that a run with a particular parameter setting yields better coverage than a run of the other compared setting. If there is no difference between two parameter setting performances, then $\hat{A}_{12} = 0.5$. For reasons of space it is not possible to show all the details of the data and analyses. For example, instead of reporting all the p-values, we only state when those are lower than 0.05.

4.1.5 Results

In the analyses in this paper, we focus on four specific settings: worst (W), best (B), default (D) and tuned (T). The worst combination W is the one that gives the worst coverage out of the $5^4 \times 2 = 1{,}250$ combinations, and can be different depending on the class under test and chosen search budget. Similarly, B represents the best configuration out of $1{,}250$. The "default" combination D is arbitrarily set to population size 100, crossover rate 0.8, rank selection with 1.7 bias, 10% of elitism rate and no parent replacement check. These values are in line with common suggestions in the literature, and those that we used in previous work. In particular, this default setting was chosen *before* running any of the experiments. Finally, given a set of classes, the tuned configuration T represents the configuration that has the highest average relative coverage on all that set of classes. When we write for example $\hat{A}_{DW} = 0.8$, this means that, for the addressed class and search budget, a run of the default configuration D has 0.8 probability of yielding a coverage that is higher than the one obtained by a run of the worst configuration W.

The data[2] collected from this large empirical study could be used to address *several* research questions. In this paper, we only focus on the five research questions that we believe are most important.

[2] The data reported in Table 1, Table 2 and Table 3 are different from what we originally reported in [4]. In the revision of this paper, we found a mistake in the calculation of the relative coverage formula, which has now been corrected.

Table 1 Relative coverage averaged out of 15 runs for default, worst and best configuration. Effect sizes for default compared to worst ($\hat{A}_{DW}$) and and compared to best configuration ($\hat{A}_{DB}$). Statistically significant effect sizes are in bold.

Class	Default	Worst	Best	$\hat{A}_{DW}$	$\hat{A}_{DB}$
Cookie	0.52	0.25	0.66	**1.00**	**0.00**
DateParse	1.00	1.00	1.00	0.50	0.50
Triangle	1.00	0.93	1.00	0.53	0.50
XMLElement	0.79	0.40	0.82	**1.00**	**0.12**
ZipOutputStream	1.00	0.88	1.00	**0.77**	0.50
CommandLine	0.45	0.25	0.59	**1.00**	0.35
Remainder	0.90	0.55	0.99	**0.99**	**0.12**
Industry1	0.63	0.30	0.88	**1.00**	**0.00**
Industry2	0.93	0.51	0.98	**1.00**	**0.01**
Attribute	0.51	0.22	0.68	**1.00**	**0.01**
DoubleMetaphone	0.70	0.51	0.75	**1.00**	**0.09**
Chronology	0.79	0.47	0.88	**1.00**	**0.04**
ArrayList	1.00	0.67	1.00	**0.67**	0.50
DateTime	0.64	0.22	0.93	**1.00**	**0.00**
TreeMap	0.62	0.06	0.73	**0.93**	**0.25**
Bessj	0.75	0.48	0.92	**1.00**	**0.00**
BellmanFordIterator	0.88	0.85	0.93	0.60	**0.30**
TTestImpl	0.56	0.37	0.69	**0.94**	**0.21**
LinkedListMultimap	0.83	0.21	1.00	**1.00**	**0.03**
FastFourierTransformer	0.73	0.62	0.76	**1.00**	**0.36**

RQ1: How large is the potential impact of a wrong choice of parameter settings?

In Table 1, for each class in the case study and test budget 100,000, we report the relative coverage (averaged out of 15 runs) of the worst and best configurations. There are cases in which the class under test is trivial for EvoSuite (e.g., DateParse), in which case tuning is not really important. But, in most cases, there is a very large difference between the worst and best configuration (e.g., Industry1). A wrong parameter tuning can make it hard (on average) to solve problems that could be easy otherwise.

Different parameter settings cause
very large variance in the performance.

RQ2: How does a "default" setting compare to the best and worst achievable performance?

Table 1 also reports the relative coverage for the default setting, with effect sizes of the comparisons with the worst and best configuration. As one would expect, a default configuration has to be better than the worst, and worse/equal to the best configuration. However, for most problems, although the default setting is *much better* than the worst setting (i.e., $\hat{A}_{DW}$ values close to 1), it is unfortunately *much worse* than the best setting (i.e., $\hat{A}_{DB}$ values are close to 0). When one uses randomized algorithms, it is reasonable to expect variance in the performance when they are run twice with a different seed. However, consider the example of Bessj in Table 1, where $\hat{A}_{DW} = 1$ and $\hat{A}_{DB} = 0$. In that case, the coverage values achieved by the default setting in 15 runs are always better than any of the 15 coverage values obtained with the worst configuration, but also always worse than any of the 15 runs obtained with best configuration. These data suggest that, if one does not have the possibility of tuning, then the use of a default setting is not particularly inefficient. However, there is large space for performance improvement if tuning is done.

Default parameter settings perform relatively well, but are far from optimal on individual problem instances.

RQ3: How is the performance of a search algorithm on a class when it has been tuned on different classes?

To answer this research question, for each class we tuned the algorithm on the *other* 19 classes, and then compared this tuned version with the default and best configuration for the class under test. Table 2 reports the data of this analysis. If one makes tuning on a sample of problem instances, then we would expect a relatively good performance on new instances. But the $\hat{A}_{TB}$ values in Table 2 are in most of the cases low and statistically significant. This means that parameter settings that should work well on average can be particularly inefficient on new instances compared to the best tuning for those instances. In other words, there is a very high variance in the performance of parameter settings.

Regarding the comparisons with default settings, $\hat{A}_{TD}$ values in Table 2 shows that tuning often results in statistically better results, whereas there is no case in which it gives statistically worse results (i.e., $\hat{A}_{TD} < 0.5$).

Tuned parameters can improve upon default values on average, but they are far from optimal on individual problem instances.

RQ4: What are the effects of the search budget on parameter tuning?

For each class and the three search budgets, we compared the performance of the default setting against the worst and the best; Table 3 shows the data of this analysis. For a very large search budget one would expect not much difference between parameter settings, as all achievable coverage would be reached with high probability. Recall that it is not possible to stop the search before because, apart from trivial cases, there are always infeasible testing targets (e.g., branches) whose number is unknown. The data in Table 3 show that trend for many of the used programs (e.g., see LinkedList-Multimap) regarding the default and best settings, but the worst setting is still much worse than the others (i.e., $\hat{A}_{DW}$ close to 1) even with a search budget of one million function calls. What is a "large" search budget depends of course on the case study.

Table 2 Relative coverage averaged out of 15 runs for tuned configuration. Effect sizes for tuned compared to default ($\hat{A}_{TD}$) and and compared to best configuration ($\hat{A}_{TB}$). Statistically significant effect sizes are in bold.

Class	Tuned	$\hat{A}_{TD}$	$\hat{A}_{TB}$
Cookie	0.63	**1.00**	**0.12**
DateParse	1.00	0.50	0.50
Triangle	1.00	0.50	0.50
XMLElement	0.80	0.64	**0.21**
ZipOutputStream	1.00	0.50	0.50
CommandLine	0.50	0.68	0.48
Remainder	0.94	0.66	**0.26**
Industry1	0.86	**1.00**	**0.24**
Industry2	0.97	**0.92**	0.36
Attribute	0.63	**0.98**	**0.24**
DoubleMetaphone	0.73	**0.91**	0.32
Chronology	0.84	**0.86**	**0.29**
ArrayList	0.80	0.40	0.40
DateTime	0.90	**1.00**	**0.20**
TreeMap	0.63	0.64	0.51
Bessj	0.89	**1.00**	**0.21**
BellmanFordIterator	0.87	0.47	**0.27**
TTestImpl	0.63	0.64	0.37
LinkedListMultimap	1.00	**0.97**	0.50
FastFourierTransformer	0.73	0.53	**0.38**

For example, for DateParse, already a budget of $100{,}000$ is enough to get no difference between best, worst and default configuration. In contrast, with a search budget of $1{,}000{,}000$, for example for Attribute there is still a statistically strong difference.

These results confirm that a very large search budget might reduce the importance of tuning. However, when we increase the search budget, that does not always mean that tuning becomes less important. Consider the case of CommandLine: At budget $10{,}000$, the $\hat{A}_{DW}$ is not statistically significant (i.e., the effect size is 0.59 and Mann-Whitney U-test has p-value greater than 0.05), whereas it gets higher (i.e., 1) for $100{,}000$ and for $1{,}000{,}000$. For $\hat{A}_{DB}$, it is statistically significant when budget is $10{,}000$, but not when we increase the budget. For example, for budget $1{,}000{,}000$ there is no difference, i.e., $\hat{A}_{DB} = 0.5$. How come? The reason is that the testing targets have different difficulty to be covered. Even with appropriate tuning, for some targets we would still need a minimum amount of search budget. If the search budget is lower than that threshold, then we would not cover (with high probability) those targets even with the best tuning. Therefore, tuning might not be so important if either the search budget is too "large", or if it is too "small", where "large" and "small" depend on the case study. Unfortunately, this information is usually not known before performing tuning.

The available search budget has strong impact on the parameter settings that should be used.

Table 3 For each test budget, effect sizes of default configuration compared to the worst ($\hat{A}_{DW}$) and best configuration ($\hat{A}_{DB}$). Statistically significant effect sizes are in bold. Some data are missing (-) due to the testing tool running out of memory.

Class	Test Budget					
	10,000		100,000		1,000,000	
	$\hat{A}_{DW}$	$\hat{A}_{DB}$	$\hat{A}_{DW}$	$\hat{A}_{DB}$	$\hat{A}_{DW}$	$\hat{A}_{DB}$
Cookie	**0.78**	**0.00**	**1.00**	**0.00**	**1.00**	**0.06**
DateParse	**0.63**	0.50	0.50	0.50	0.50	0.50
Triangle	**0.70**	0.50	0.53	0.50	0.62	0.50
XMLElement	**0.89**	**0.04**	**1.00**	**0.12**	1.00	0.00
ZipOutputStream	**0.81**	0.43	**0.77**	0.50	**0.72**	0.50
CommandLine	0.59	**0.22**	**1.00**	0.35	**1.00**	0.50
Remainder	**0.89**	**0.25**	**0.99**	**0.12**	**1.00**	0.46
Industry1	**0.92**	**0.00**	**1.00**	**0.00**	-	-
Industry2	**0.92**	**0.00**	**1.00**	**0.01**	**1.00**	0.42
Attribute	**0.76**	**0.00**	**1.00**	**0.01**	**1.00**	**0.00**
DoubleMetaphone	**0.86**	**0.00**	**1.00**	**0.09**	**1.00**	**0.04**
Chronology	0.67	**0.00**	**1.00**	**0.04**	**1.00**	0.33
ArrayList	**0.70**	0.43	**0.67**	0.50	**1.00**	0.50
DateTime	**0.93**	**0.00**	**1.00**	**0.00**	**1.00**	0.40
TreeMap	**0.63**	**0.24**	**0.93**	**0.25**	**1.00**	**0.14**
Bessj	**0.80**	**0.00**	**1.00**	**0.00**	1.00	**0.00**
BellmanFordIterator	0.57	0.43	0.60	**0.30**	-	-
TTestImpl	**0.87**	**0.11**	**0.94**	**0.21**	**0.97**	**0.27**
LinkedListMultimap	0.70	**0.00**	**1.00**	**0.03**	**1.00**	0.50
FastFourierTransformer	0.63	**0.00**	**1.00**	**0.36**	-	-

RQ5: How much can generalization improve with a larger number of classes used for tuning?

In machine learning [25], the larger the training set is, the better the results will likely be. In the context of parameter tuning for SBSE, how large should a training set be? Having a larger case study will result in better tuning, but it would be more expensive and time consuming to carry out the tuning process. If for example we double the size of the case study, how much will tuning improve? To answer these research questions, we carried out a set of simulations based on the data of the previous experiments.

Given q classes chosen at random from the 20 classes used in the case study, we tune EvoSuite on those q classes (i.e., we choose the configuration with highest average coverage from the previous experiments). We then evaluate the average relative coverage z_i of EvoSuite with those parameters on 10 classes chosen at random from the remaining $20 - q$ classes. Then, we repeat this process 100 times, and calculate the average of the average relative coverage, i.e., $z^q = \frac{1}{100}\sum_{i=1}^{100} z_i$. In this way, z^q gives us an estimation of what would be the average relative coverage if we apply tuning on a training set of q classes. We apply this simulation for different increasing values of q, from 2 to 10, with a search budget of 100,000 executed statements. Figure 1 shows the average relative coverages z^q, whereas Figure 2 shows the average $\hat{A}_{12}$ values of comparing $q > 2$ with $q = 2$. In other words, given average performance with tuning on $q = 2$ classes, we calculate the average effect size of the improvement of the settings that were tuned with a higher number q of classes.

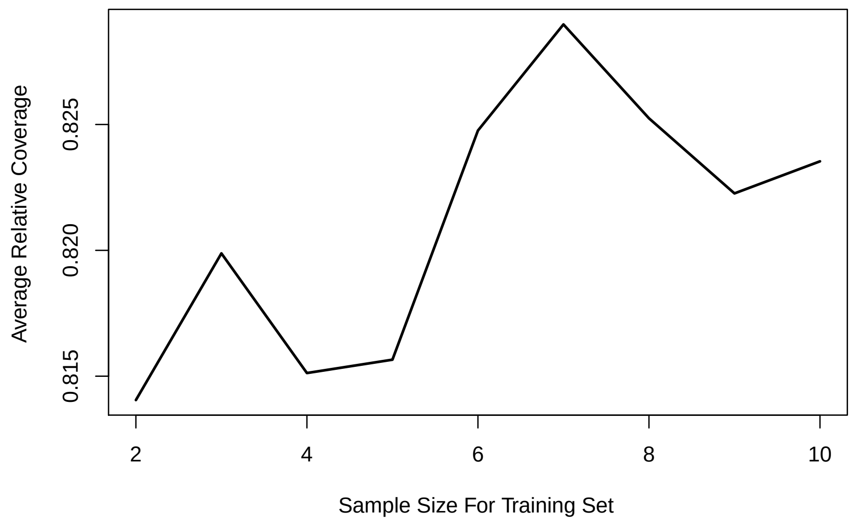

Fig. 1 Average relative coverage for increasing training sizes.

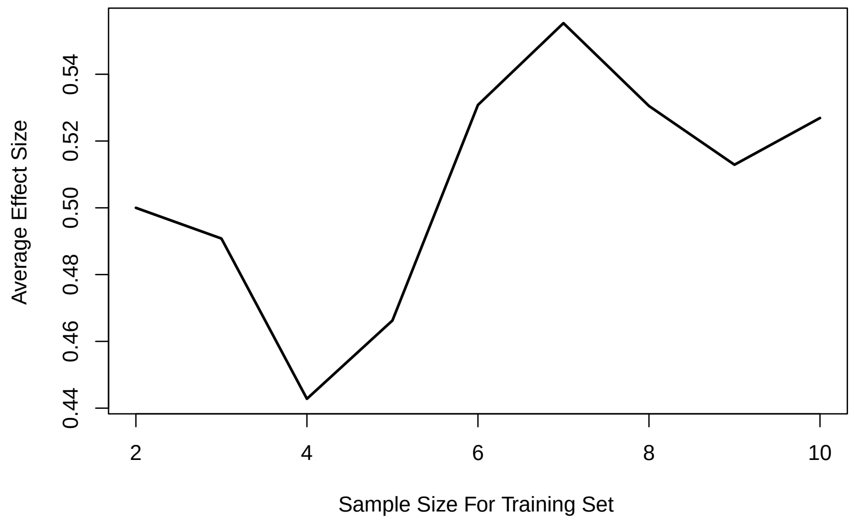

Fig. 2 Average $\hat{A}_{12}$ of increasing training sizes $q > 2$ compared to $q = 2$.

The results in Figure 1 shows that there is not much difference between $q = 2$ and $q = 10$. A Kruskal-Wallis test on these data, to check if the parameter q has any impact on performance, resulted in $\chi^2 = 12.72$ and a p-value equal to 0.12. Although one would expect a monotonically increase of coverage z^q with higher q, this does not happen. The reason is simple, because the average z^q is just an estimation based on 100 observations, and those estimations are affected by variance. The differences in coverage values with different q are so small that such variance hides them. However, it is important to note that, already with $q = 2$ we obtain an average relative coverage of 0.81, whereas without tuning it is 0.76. Furthermore, we only consider sizes only up to $q = 10$. Even if improvement is not much from $q = 2$ to $q = 10$, at this point we cannot state what would happen for higher values such as $q = 1{,}000$.

Although increasing the sample size improves performance, the improvement is low.

4.2 Second Case Study - Real-World Impact of Tuning

In the second case study, we focus on the real-world impact of parameter tuning.

4.2.1 Experimental Setup

We chose 10 projects at random from the SF100 corpus of classes [16], which is a random sample of 100 Java projects taken from the SourceForge repository, consisting of 8784 classes in total. The 10 randomly selected projects resulted in a total of 609 classes. Note that we used only 10 out of the 100 projects in SF100 just because running these experiments is very time consuming, even with the help of a cluster of computers to run the experiments. Second, instead of considering the number of executed statements as stopping criterion, we use a two minute timeout. The reason for using a timeout is to make the experiments more realistic, as practitioners in general would choose time as stopping criterion. However, the use of time increases the threats to both internal and external validity, as all the low level implementation details of EvoSuite can affect the final results.

In the previous case study, we chose a set of parameters and a set of values for them. For example, for the crossover rate we chose values in $\{0 , .2 , .5 , .8 , 1\}$. But what if the best value is not among the ones used for the experiments? Choosing and evaluating all possible combinations of values for all parameters is simply not possible. There are, however, techniques that, given a set of experiments with different parameter/value combinations, can try to estimate which is the best combination. The commonly used technique is the so called *response surface methodology* [26]. In this section, we will describe how we applied the response surface methodology to tune the parameters of EvoSuite. However, note that the response surface methodology is a very complex subject. The interested reader is thus referred to the excellent book written by Meyers *et al.* [26] for more detailed explanations on the different aspects of the response surface methodology.

4.2.2 Response Surface Methodology

At a high level, the response surface methodology can be summarized as follows: First, given n parameters where each w_i represents their value, a parameter configuration would be defined by the vector $W = \{w_1, \, \dots \, , w_n\}$. Each parameter w_i can assume several values. We can have k experiments with different configurations W_ρ. If the parameters w_i are all binary, then there would be 2^n configurations, which can be already a very high number. Therefore, in general, k represents only a small subset of all configurations. For each configuration W_ρ among the k chosen, we can calculate a *response* $r(W_\rho) = y$, i.e., a measure that quantifies how good the configuration W_ρ is at solving the problem at hand.

In our case, the response y is the average raw coverage obtained on all the classes in the case study. In other words, to calculate y, we need to run EvoSuite with configuration W_ρ on the entire case study (609 classes). Recall from Section 2 that the final goal of this tuning process is not to find good parameters for the employed case study, but for any new problem instance. From a practical standpoint, that might be considered irrelevant as the real final goal is to find good parameters that will work well on the *instances of the practitioners* when they use EvoSuite in their daily software engineering jobs. Using a random sample of open source programs is, at the moment, a reasonable approach to try to obtain such general results. However, such context is different from the normal applications of the response surface methodology, e.g., the optimization of chemical and physical processes, where the conditions of the "case study" are the same (or very similar) as the ones of the "practitioners". For example, once the right parameters for a chemical process are found, such a chemical process can be repeated millions of times with the optimized parameters by everyone that needs to use it. In contrast, once we have optimized EvoSuite parameters on a set of classes, after we generate test cases for them, in general we do not need to generate test cases for those classes many times again.

Once we have k configurations, and we calculate the response $r(W_\rho) = y$ for each of them, we can build a model f to capture the relations between the parameters w_i and response y. In particular, we can consider the following two-way interaction model:

$$f(W_\rho) = \beta_0 + \sum_{i=1}^{n} \beta_i w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \beta_{ij} w_i w_j \; .$$

At this point, we can use a regression algorithm to find the values of the constants in the model (i.e., β_0, and all the β_i and β_{ij}) such that the error on the predicted value y if minimized. In other words, we want a model that is able to properly predict the response y in each of the k experiments with different configurations W_ρ, i.e., $f(W_\rho) = r(W_\rho) = y$.

Once the model f is built, we can *optimize* its variables (but keeping the βs as constant) to find the combination that obtains a response y as high as possible. In other words, we first build a model that we use to predict how well a configuration W_ρ will perform (i.e., average coverage y) without the need of running any experiment (apart from the ones done for building the model in the first place). Then, we use the model to find the configuration that gives highest response y (again, without the need to run any further experiment on the case study). Obviously, the model f can only *predict* what the performance would be. As any prediction, it can be more or less wrong. Once we obtain a configuration that optimizes f, we still need to run it on the case study to actually verify how good it is. How well this configuration will actually perform will depend not only on the number k of data points W_ρ used to build the model, but also on how well a two-way interaction model is at capturing the dynamics/relations among the tuned parameters (i.e., whether higher level interactions can have strong influence on the response y). From a practical point of view it does not really matter whether the model is really correct or not. The only thing that really matters in the end is whether this methodology can yield a significantly better set of tuned parameters.

4.2.3 Parameters

There are several ways to choose a fractional factorial design, i.e., the choice of the k configurations. A proper choice depends on the number of variables n and on which kind of model (e.g., linear or two-way interactions) we want to build. For the analyses in this paper, we used the freely available statistical tool R [28] and its package *rsm* [23]. In particular, we used a central composite design given by the *ccd* function, where the model was built with the function *rsm*. (Note: *rsm* stands for "response surface methodology"). The response of the model is then optimized with the function *steepest*. Notice that, with a central composite design, we have five levels for each parameter w_i. In other words, in the k generated configurations, each of the w_i parameters is experimented with five different values.

An advanced tool like EVOSUITE has many parameters that can be tuned. One approach to cope with this problem could be to divide the tuning in two phases: a first lighter "screening" phase (with a small factorial design) in which we can rule out all the parameters that do not seem to effect much performance; then, we can concentrate and further experiment and tune the most important parameters from the first screening phase. This is for example what has been done by Feldt and Nordin [12] for Genetic Programming. In this paper, to simplify the analyses, we chose a single phase with eight parameters. We choose parameters that are general for SBSE (e.g., population size), mutually independent (e.g., if we consider rank selection, then we ignore the tournament size parameter) and we only consider parameters that have a numerical range of values (i.e., not binary, no enumerations). On one hand, this means we just needed to build a single model for the response surface methodology. On the other hand, we could not consider all the parameters used in the first case study. In addition to the parameters used in the first case study, here we also consider the probability P_s of using the pool of constants, the probability σ of inserting new test cases, and number m of mutations (see Section 3 for more details).

The eight employed parameters and their chosen ranges are:

- Population Size: $[5, 99]$
- Chromosome Length: $[5, 99]$
- Rank Bias: $[1.01, 1.99]$
- Number of Mutations: $[1, 10]$
- Max Initial Number of Tests: $[1, 10]$

Table 4 Actual values used for the parameter mapping, where $\alpha = 3.13$.

Parameter	$-\alpha$	-1	0	+1	$+\alpha$
Population Size	5	37	52	67	99
Chromosome Length	5	37	52	67	99
Rank Bias	1.01	1.34	1.5	1.65	1.99
Number of Mutations	1	4	5	7	10
Max Initial Number of Tests	1	4	5	7	10
Crossover Rate	0.01	0.34	0.5	0.65	0.99
Probability of Using the Pool of Constants	0.01	0.34	0.5	0.65	0.99
Probability of Inserting New Test Case	0.01	0.34	0.5	0.65	0.99

– Crossover Rate: $[0.01, 0.99]$
– Probability of Using the Pool of Constants: $[0.01, 0.99]$
– Probability of Inserting New Test Case: $[0.01, 0.99]$

When we create a central composite design with the function *ccd*, for each parameter we obtain five levels, i.e., five different values. In particular, we obtain $\{0, 1, -1, \alpha, -\alpha\}$, where α depends on several factors [26]. We obtain k configurations divided in two distinct blocks. On one hand, in the "cube" block we have several combinations of ± 1 centered at the origin $(0, 0, \dots ,0)$. On the other hand, the "star" block is composed of $2n$ configurations centered in the origin in which each variable is considered with $\pm\alpha$ only once (while the other $n-1$ variables are kept to 0).

However, these five values need to be "mapped" to the actual ranges of the parameters. For some parameters there are known bounds (e.g., $[0,1]$ for probability values), whereas for others (e.g., population size) in theory there might be no (upper) bound (e.g., very large population sizes). It is responsibility of the researcher to define which is the *region of interest* [26] for the experiments. On one hand, a too small region might not present much of the difference among the response values of its design points. On the other hand, choosing a too large region could reduce the precision of the built models if the response surface has complex shapes. The chosen regions for the different analyzed parameters where arbitrarily set to "reasonable" values before running any of the experiments.

The values are mapped such that $-\alpha$ represents the lowest value of the region of interest, α the highest value, and the center 0 being the middle of the range. The mapping of ± 1 depends on the actual value of α. For example, if $\alpha = 2$, then the coordinate -1 will be mapped to 25% of the range, whereas the coordinate $+1$ will represent the 75% of the range. To be more precise, given the range $[min,max]$, then a coordinate w (e.g., $w = \pm 1$) would be mapped into to the value $v(w) = min + \frac{w+\alpha}{2\alpha}(max - min)$. For example, the center 0 will have value $v(0) = min + \frac{1}{2}(max - min) = \frac{min+max}{2}$ independently of α. Table 4 shows the mapping used in this case study.

4.2.4 Experimental Procedure

Given the eight parameters under study, a central composite design resulted in 280 configurations (using *ccd(8)*). For every configuration, we ran EvoSuite on each of the 609 classes in the case study for two minutes each. The choice of using a two minute timeout is based on possible realistic usages of EvoSuite (recall Section 4.1) while, at the same time, making it possible to complete the experiments in a reasonable amount of time.

In total, these experiments took $280 \times 609 \times 2 = 341{,}040$ minutes, i.e., more than 236 days (so we needed to use a cluster of computers). After collecting data for the 280 configurations, we built and optimized a two-way interaction model. Among the different configurations given as output by the *steepest* function, we chose the one with 3.5 distance, as it had highest response, and with the parameters still inside the original ranges.

We then ran the optimized configuration 100 times with different seeds on each class in the case study. In a similar way, we ran EvoSuite with no tuning (i.e., we kept the default values for those eight parameters) 100 times on each class. In total, these experiments added another 169 days of computational time. Note that, for this second case study, we had a total of $(280 + 100 + 100) \times 609 = 292{,}320$ runs of EvoSuite.

4.2.5 Results

RQ6: What is the impact of parameter tuning in a real-world context?

Results of these experiments were unexpected, considering the many successful real-world results on the application of the response surface methodology [26]. On average, the tuned configuration obtained a raw coverage of 43.55%, whereas the default configuration obtained 44.19%. The effect size of tuned compared to default is 0.497, i.e., the difference between the two configurations is minimal and likely of little practical value. In other words, *tuning did not improve the performance in this case*. When we look at the statistical difference, even considering the high number of data points (i.e., $2 \times 100 \times 609 = 121{,}800$), a Mann-Whitney U-test resulted in a 0.09 p-value. In other words, the difference is so small that, even with a large sample, it is not possible to reject the null hypothesis with high statistical confidence.

In order to determine the reasons for this finding, we applied a so called *lack of fit* test on the model in order to evaluate its quality. The *rsm* function returns a p-value of a lack of fit test, where the null hypothesis is that there is no lack of fit. The built model had a 0.02 p-value, which suggests that model is not particularly fit, and so the results provided by the *steepest* function are not reliable. Following the suggestions in the literature, we tried to fit a higher order model, in particular a second order model (the highest provided in the *rsm* function). Still, we obtained a non-particularly fit model (i.e., p-value equal to 0.03).

In these cases, the literature of the response surface methodology suggests to run further experiments, but with a smaller "region of interest" [26]. For example, instead of considering probability values (e.g., crossover rate) from 0.01 to 0.99 (i.e., ± 0.49

on the center 0.5), we could choose a smaller delta, e.g., ± 0.2 (and so range 0.3 to 0.7), or even smaller if the new built models are still not fit. But, depending on the problem, a too small region could have the difference between response values small enough to be masked by their variance (e.g., due to the randomized nature of the used algorithms). Once a fit model is built, and the optimized parameters (i.e., highest response) in that region of interest are found, then a new series of experiments can be run centered in that optimum (for more details, please refer to [26]). As such experiments are very time consuming, and because the current data are already sufficient to (at least partially) answer our research question, we did not carry out those further experiments.

To improve the fit of the model without running new experiments, we also tried the following approach: Because parameters with little impact on the response might reduce the fit of the model, we removed them from the model, and built a new one without them. In particular, regression functions such as lm in [28] (and rsm as well) also carry out statistical tests on each parameter β of the model to see if it is statistically different from 0. Considering a 0.05 significant level, in our case, for the crossover rate and the probability P_s of choosing from the pool of constants, it was not possible to reject the hypothesis that they have zero coefficient (and so no impact) on the response of the model. We removed those two parameters, and built a new second-order model with the remaining six parameters. Unfortunately, the new model still presented a lack of fit.

For building the response model, by using a central composite design, we carried out experiments at 280 different configurations. Although the derived model is not fit, we still have those 280 observations. We hence chose the one among them with highest response (i.e., average raw coverage), which was 44.28%. This is higher than what we obtained with default values (i.e., 44.19%), but with very low effect size (i.e., 0.501), where a Mann-Whitney U-test resulted in a very high 0.89 p-value. So, there is no statistical evidence to claim that this tuned configuration is better than default values.

In our experiments, the response surface methodology did not lead to tuned values that improved over the default parameter setting.

To further understand why the response surface methodology did not provide good results, we also investigated the *quality* of the used data. In the experiments, we had 280 configurations, whose responses were derived by calculating the mean value from one run of EvoSuite on the 609 classes. This is an expensive process, as it takes $609 \times 2/60 = 20.3$ hours per configuration. Among the configurations, the worse had average 41.7% coverage and, as already discussed above, the best had average 44.28% coverage. But those average coverage values are just estimates calculated on 609 data points. Are those estimates accurate? To check this potential threat to validity, for each average value, we used the bootstrapping technique to calculate 95% confidence intervals [7]. We used the R package *boot*, and each bootstrapping calculation was based on $10{,}000$ runs. Given an average value m, a 95% confidence interval $[a,b]$ means that the real average value has a 95% probability of being between a and b, where m is the most likely estimate. Above the 280 configurations, the highest value for a was 41.0, whereas the lowest value for b was 44.9. In other words, each of the 280 average coverage values lies inside the 95% confidence intervals of all the others.

The bootstrapping analysis shows that the obtained coverage value estimates are not tight enough. Increasing the sample size would help, but is unfeasible in our experimental setting because of the resulting computational cost. Such variance in the data can be explained by the fact that many classes are either trivial (e.g., classes with only set/get methods) or very difficult (e.g., they require the creation/manipulation of files). Consequently, in the SF100 corpus there is such a large variance among the achievable coverage values among different classes [16], that one single run on 609 classes is not enough to obtain precise estimates.

In summary, unless we carry out further experiments that demonstrate otherwise, the response surface methodology cannot be considered to provide any improvement to the tuning of EvoSuite compared to default parameter values.

When addressing real-world SBSE problems, parameter tuning becomes computational expensive and it does not necessarily bring significant improvements.

4.3 Third Case Study – Variability in SBSE Problems

In the previous section, the application of the response surface methodology to the problem of tuning EvoSuite on the SF100 corpus of classes was not successful. We did not manage to build a fit model that could be used to find better tuned configurations. Without formal mathematical proofs, it is not possible to state why the response surface methodology did not work. We can only provide plausible explanations, backed up by further empirical data (e.g., the application of bootstrapping confidence intervals [7]). As discussed in the previous section, one possible culprit for the bad performance of the response surface methodology could be the "variability" of the responses on the design points (e.g., we used 609 different classes, and EvoSuite is also affected by randomness). Indeed, variability is a major issue in the response surface methodology. For example, in their book [26], Myers *et al.* at page 114 wrote:

> "If the magnitude of the variability seems reasonable, continue; on the other hand, if larger than anticipated (or reasonable!) variability is observed, stop. Often it will be very profitable to study why the variability is so large before proceeding with the rest of the experiment."

Unfortunately, Myers *et al.* did not formally quantify what "reasonable" and "large" mean (in numbers). On one hand, it could be that the variability in the previous experiments was large enough to preclude a successful use of the response surface methodology. On the other hand, maybe the variability was not really so large, and other factors played a role in the achieved negative results. To shed light on this issue, we carried out further experiments to answer the following research question:

RQ7: Is the variance in the SF100 corpus the main obstacle for a successful use of the response surface methodology?

To answer this research question, we applied the same central composite design we used in the experiments in the previous section, with the same eight parameters to tune, and the same ranges. This time, however, instead of using a single run of EVOSUITE on 609 classes from the SF100 corpus, we used one single class, TreeMap, and 100 runs per design point. The reason of using a single class is to avoid high variance in the responses due to putting together different classes with different difficulty, as in the SF100 corpus. We chose to use only TreeMap as it is one the most commonly used artifacts in the literature of testing object-oriented software [32]. Each run lasted two minutes, as in previous experiments. In total, we had $280 \times 100 \times 2 = 56{,}000$ minutes, i.e., 38 days of computational effort.

As in the previous experiments, the inferred second order model exhibited a lack of fit (p-value was equal to 0.0039). Using only one class does not completely remove the variability in the responses, as EVOSUITE is based on a randomized algorithm. Maybe 100 runs are not enough. But having $1{,}000$ runs per design point, for example, would have required another 342 days of computational effort.

Besides the problem of variability, another possible cause for the lack of fit can be the complexity of the response surface. To shed light on this possibility, we ran another set of experiments. We still used TreeMap with a two minute timeout, but this time we only considered two parameters to tune (i.e., population size and crossover rate). A central composite design resulted in 16 configuration points. This time, however, to try to reduce to minimum the effects of variability, we used $1{,}000$ runs. In total, we had $16 \times 1{,}000 \times 2 = 32{,}000$ minutes, i.e., 22 days of computational effort. To our greatest disbelief, a second order model again resulted in lack of fit! In particular, we obtained a p-value equal to 1.7×10^{-5}.

To avoid jumping to conclusions based on a single class, we repeated the same experiments on a different class. In particular, we used the class called "Option" from the Apache Commons libraries. We chose this class because it is complex, and because we have long experience in using EVOSUITE on it (e.g., during demos), which would help manual debugging. This set of experiments added another 22 days of computational effort. Yet again, the model was not fit, giving a 1.6×10^{-4} p-value.

All these negative results raise doubt on whether we applied the response surface methodology correctly. To reduce this threat to validity, we applied the same type of analysis on common data sets from the literature (e.g., data sets included in the R library *rsm*). Those worked fine, in the sense that it was possible to infer fit models.

Unless there were implementation errors in how we collected and analyzed the data, it does not seem that variability alone is the culprit here. If $1{,}000$ runs per design point on one single class and just two parameters are not enough to cope with variability issues, then one could reasonably pose questions on the practical usefulness of the response surface methodology. At this point we cannot explain with high confidence *why* it does not work in our empirical study. This would require further experiments and in depth analyses that would go beyond the scope of this paper.

Variability is a major issues for the response surface methodology, but our empirical study shows that there are other still unknown factors that play an even greater role.

4.4 Consequences of the Negative Result

In our empirical analyses, the response surface methodology failed to provide models that could be used to infer better parameter configurations. We investigated why that was the case. In particular, we focused on the possible issue of variability in the responses. Variability is a major problem in the response surface methodology, but our empirical study shows with high confidence that such variability is not the main culprit in our context. The negative results presented in this section, considering the current status of published research in software engineering, could be considered as "inconclusive".

The role of negative results has been long discussed in many academic fields, and their importance has long been advocated (e.g., [22, 31, 33]). Negative results play their role in the advancement of scientific development, and are not necessarily less useful and valid than positive results. In fact, positive results often simply represent the experimenter's bias, which has led Ioannidis to show that *most published research findings are false* [21]. There are several reasons why most positive results in the literature are likely to be false. But, for a detailed discussion on the argument, we refer the interested reader to Ioannidis's excellent essay [21].

A simple and effective example to understand the importance of negative results can be found in the correspondence section of *Nature* by Gupta and Stopfer [18]:

> "Say a study finds no statistically favourable evidence for a hypothesis at the predetermined significance level ($P = 0.05$, for example) and, like most with negative results, it is never published. If 19 other similar studies are conducted, then 20 independent attempts at the 0.05 significance level are, by definition, expected to give at least one hit. A positive result obtained in one of the 19 studies, viewed independently, would then be statistically valid and so support the hypothesis, and would probably be published."

In this paper, we analyzed an important problem in software engineering, i.e., parameter tuning. Many software engineering tasks can be addressed with search algorithms [19], and most work in SBSE can be affected by tuning. We chose test data generation using EvoSuite as working example, and our results can generalize outside our case study, as we address a statistically valid sample of open source projects (by using the SF100 corpus [16]). We applied the world's most used technique for optimizing parameters, i.e., the response surface methodology [26], and not a new previously unknown technique that we invented and turned out to be not working. Therefore, we believe the negative results presented in this paper can be useful to direct future research.

Future work will be needed to fully understand why the response surface methodology failed, e.g., by studying different sizes for the regions of interest and trying different factorial designs rather than the central composite one. Such work would be important to design other tuning techniques (or variants of the response surface methodology) that are effective for test data generation.

It will also be important to verify whether other SBSE applications (e.g., regression testing) present similar tuning challenges, or if test data generation at unit level for object-oriented software is just an exceptional case. For other problems in SBSE, it might be that the response surface methodology is effective, but this cannot be stated before investigating such matter empirically. A practitioner or researcher who has the resources for a tuning phase should still use such methodology, even if later it turns out that the models are not fit. The reason is that she/he would still get the responses of the investigated design points, and those might result in better performance compared to the default parameter settings (although this was not the case in our empirical study).

5 Discussion and Guidelines

The empirical analysis carried out in this paper clearly shows that tuning can have a strong impact on search algorithm performance and, if it is not done properly, there are dire risks of ending up with tuned configurations that are worse than suggested values in the literature.

The main conclusion of our analysis is that using default values coming from the literature is a viable option. Applying parameter tuning is a time consuming activity, which is not always going to bring significant improvements. On one hand, even if the improvements are not large, it makes sense to apply parameter tuning before releasing an SBSE tool prototype to practitioners. On the other hand, if parameter tuning is indeed expensive, and a researcher is investigating new techniques, it could make sense to rather use the available time for larger case studies and more technique comparisons instead of parameter tuning.

Although there are “default” values in the literature for many common parameters, and even though there has been work on studying their effects, addressing real-world SBSE problems would likely bring to new parameters that are specific for the addressed problem. For example, the probability P_s of using a pool of constants derived from the source/byte-code of the SUT is something very specific to test data generation. For these types of parameters, there are no “default” and widely studied/evaluated values in the literature. In such cases, parameter tuning can become necessary. So, when new techniques are proposed that require new parameters to be set, it is advisable to study different values for them.

When we tune an algorithm on some problem instances, it might end up that the found parameters are too specific for the chosen case study. This could lead to problems of *overfitting*. It would hence be important to use machine learning techniques [25] when tuning parameters. Which ones to use is context dependent, and a detailed discussion is beyond the scope of this paper. Instead, we discuss some basic scenarios here, aiming at developers who want to tune parameters before releasing SBSE tool prototypes, or researchers who want to tune tools for scientific experiments. Further details can be found for example in [25].

Given a case study composed of a number of problem instances, randomly partition it in two non-overlapping subsets: the *training* and the *test* set. A common rule of thumb is to use 90% of instances for the training set, and the remaining 10% for the test set. Do the tuning using only the problem instances in the training set. Instead of considering all possible parameter combinations (which is not feasible), use techniques such as the response surface methodology (e.g., used in [27]). Given a parameter setting that performs best on this training set, then evaluate its performance on the test set. Draw conclusions on the algorithm performance only based on the results on this test set.

If the case study is "small" (e.g., because composed of industrial systems and not open-source software that can be downloaded in large quantities), and/or if the cost of running the experiment is relatively low, use k-fold cross validation [25]. In other words, randomly partition the case study in k non-overlapping subsets (a common value is $k = 10$). Use one of these as test set, and merge the other $k-1$ subsets to use them as training set. Do the tuning on the training set, and evaluate the performance on the test set. Repeat this process k times, every time with a different subset for the test set, and remaining $k - 1$ for the training set. Average the performance on all the results obtained from all the k test sets, which will give some value v describing the performance of the algorithm. Finally, apply tuning on *all* the case study (do not use any test set), and keep the resulting parameter setting as the final one to use. The validity of this parameter setting would be estimated by the value v calculated during the cross validation.

Comparisons among algorithms should never be done on their performance on the training set — only use the results on validation sets. As a rule of thumb, if one compares different "tools" (e.g., prototypes released in the public domain), then no tuning should be done on released tools, because parameter settings are an essential component that *define* a tool. However, if the focus is on evaluating algorithms at a high level (e.g., on a specific class of problems, is it better to use population based search algorithms such as genetic algorithms or single individual algorithms such as simulated annealing?), then each compared algorithm should receive the same amount of tuning.

6 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our experiment 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 repeated each experiment several times with different random seeds, and we followed rigorous statistical procedures to evaluate their results.

There is an overwhelming amount of successful applications of the response surface methodology in many engineering disciplines. However, in our empirical study, it was not successful. This poses a reasonable doubt on whether we applied the response surface methodology correctly. Further analyses (e.g., with smaller regions for the parameter values) could shed light on this issue, but they are very time consuming. In any case, our study shows that proper parameter tuning is not a trivial task.

Threats to *construct validity* come from the fact that we evaluated parameter settings only based on structural coverage of the resulting test suites generated by EVOSUITE. Other factors that are important for practitioners and that should be considered as well are the size of the test suites and their readability (e.g., important in case of no formal specifications when assert statements need to be manually added). Whether these factors are negatively correlated with structural coverage is a matter of further investigation.

Threats to *external validity* come from the fact that, due to the very large number of experiments, we only used 20 classes in the first case study, which still took weeks even when using a computer cluster. Furthermore, we manually selected those 20 classes, in which we tried to have a balance of different kinds of software. A different selection for the case study might result in different conclusions. To cope with this problem, in the second case study we used the SF100 corpus [16], which is composed of projects randomly selected from the open source repository SourceForge. In total, 609 classes were used (10 projects selected randomly from the 100 available).

Note that there are two reasons for which SF100 was not used for the first case study as well. First, at the time the empirical analysis on the first case study was carried out [4], not only the SF100 corpus [16] was not available yet, but also EVOSUITE still missed several important features to be able to handle programs randomly selected from internet (e.g., sandbox execution to avoid corruption of the host environment, as for example deletion of files). Second, there is very large variation (with high kurtosis) in the SF100 corpus, and so a small sample of only 20 classes could be too skewed.

The results presented in this paper might not be valid on all software engineering problems that are commonly addressed in the literature of SBSE. Based on the fact that parameter tuning might have large impact on search algorithm performances, we hence strongly encourage the repetition of such empirical analysis on other SBSE problems.

7 Conclusion

In this paper, we have reported the results of the largest empirical study in parameter tuning in search-based software engineering to date. In particular, we focused on test data generation for object-oriented software using the EVOSUITE tool [13].

It is well known that parameter tuning has effects on the performance of search algorithms. However, this paper is the first that quantifies these effects for a search-based software engineering problem. The results of this empirical analysis clearly show that tuning can improve performance, but default values coming from the literature can be already sufficient. Researchers need to take this into account when the cost of a tuning phase becomes so high to come into the way of larger case studies or more technique comparisons.

To entail technology transfer to industrial practice, parameter tuning is a task of responsibility of who develops and releases search-based tools. A practitioner, that wants to use such tools, should not be required to run large tuning phases before being able to apply those tools on his problems at hand. Before releasing a tool to the public, a tuning phase, even if costly, would be beneficial, but only as long as it is carried out properly. It is hence important to have appropriate tuning phases on which several problem instances are employed before releasing a search-based tool, and such instances should represent a statistically valid sample.

An issue that is often neglected is the relation between tuning and search budget. The final user (e.g., software engineers) in some cases would run the search for some seconds/minutes, in other cases they could afford to run it for hours/days (e.g., weekends and night hours). In these cases, to improve search performance, the parameter settings should be different. For example, the population size in a genetic algorithm could be set based on a linear function of the search budget. However, that is a little investigated topic, and further research is needed.

Among the several research questions addressed in this paper, we also studied the application of the response surface methodology, which is perhaps the most used technique to tune parameters. However, that study led to *negative results*, as such methodology did not work in our context. Negative results are rarely reported in software engineering, although other more mature research fields such as biology do have entire journals dedicated to them. We addressed the question of why the response surface methodology did not work, and such endeavor resulted in an empirical analysis totaling 465 days of computational effort (which required the use of a cluster of computers).

For future work, we think it will be important to repeat the same type of experiments on other search-based software engineering problems. The goal would be to study if those problems present the same type of challenges from the point of view of parameter tuning, or if there are large differences among the different problems. Furthermore, it will also be important to carry out further studies to fully understand why the response surface methodology did not work in our context. This would help in designing new variants (or completely different tuning techniques) that lead to find better parameter settings in the context of test data generation for unit testing. For example, techniques like the one described in [6] would be a first candidate to investigate.

For more information about EvoSuite, please visit our website at:

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

Acknowledgements

Andrea Arcuri is funded by the Norwegian Research Council. This project has been funded by a Google Focused Research Award on “Test Amplification”.

References

1. Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R.: A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Transactions on Software Engineering (TSE) **36**(6), 742–762 (2010)

2. Arcuri, A.: A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. IEEE Transactions on Software Engineering (TSE) **38**(3), 497–519 (2012)
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 1–10 (2011)
4. Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: International Symposium on Search Based Software Engineering (SSBSE), pp. 33–47 (2011)
5. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box system testing of real-time embedded systems using random and search-based testing. In: IFIP International Conference on Testing Software and Systems (ICTSS), pp. 95–110 (2010)
6. Bartz-Beielstein, T., Markon, S.: Tuning search algorithms for real-world applications: A regression tree based approach. In: IEEE Congress on Evolutionary Computation (CEC), pp. 1111–1118 (2004)
7. Chernick, M.: Bootstrap Methods: A Practitioner's Guide (Wiley Series in Probability and Statistics) (1999)
8. Conrad, A., Roos, R., Kapfhammer, G.: Empirically studying the role of selection operators during search-based test suite prioritization. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1373–1380. ACM (2010)
9. Da Costa, L., Schoenauer, M.: Bringing evolutionary computation to industrial applications with GUIDE. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1467–1474 (2009)
10. De Jong, K.: Parameter setting in EAs: a 30 year perspective. Parameter Setting in Evolutionary Algorithms pp. 1–18 (2007)
11. Eiben, A., Michalewicz, Z., Schoenauer, M., Smith, J.: Parameter control in evolutionary algorithms. Parameter Setting in Evolutionary Algorithms pp. 19–46 (2007)
12. Feldt, R., Nordin, P.: Using factorial experiments to evaluate the effect of genetic programming parameters. Genetic Programming pp. 271–282 (2000)
13. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: International Conference On Quality Software (QSIC), pp. 31–40. IEEE Computer Society (2011)
14. Fraser, G., Arcuri, A.: It is not the length that matters, it is how you control it. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 150 – 159 (2011)
15. Fraser, G., Arcuri, A.: 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 (2012)
16. Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 178–188 (2012)
17. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 147–158. ACM (2010)
18. Gupta, N., Stopfer, M.: Negative results need airing too. Nature **470**(39) (2011). Doi:10.1038/470039a
19. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Tech. Rep. TR-09-03, King's College (2009)
20. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**(3), 270–285 (1993)
21. Ioannidis, J.: Why most published research findings are false. PLoS medicine **2**(8), e124 (2005)
22. Knight, J.: Negative results: Null and void. Nature **422**(6932), 554–555 (2003)
23. Lenth, R.: Response-surface methods in R, using RSM. Journal of Statistical Software **32**(7), 1–17 (2009)
24. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2), 105–156 (2004)
25. Mitchell, T.: Machine Learning. McGraw Hill (1997)
26. Myers, R., Montgomery, D., Anderson-Cook, C.: Response surface methodology: process and product optimization using designed experiments, vol. 705. John Wiley & Sons Inc (2009)
27. Poulding, S., Clark, J., Waeselynck, H.: A principled evaluation of the effect of directed mutation on search-based statistical testing. In: International Workshop on Search-Based Software Testing (SBST) (2011)
28. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2008). URL http://www.R-project.org. ISBN 3-900051-07-0
29. Ribeiro, J., Zenha-Rela, M., de Vega, F.: Adaptive evolutionary testing: An adaptive approach to search-based test case generation for object-oriented software. Nature Inspired Cooperative Strategies for Optimization (NICSO 2010) pp. 185–197 (2010)

30. Ribeiro, J.C.B.: Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1819–1822. ACM (2008)
31. Schooler, J.: Unpublished results hide the decline effect. Nature **470**(437) (2011). Doi:10.1038/470437a
32. Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., Marinov, D.: Testing container classes: Random or systematic? In: Fundamental Approaches to Software Engineering (FASE) (2011)
33. Smart, R.: The importance of negative results in psychological research. Canadian Psychologist/Psychologie canadienne; Canadian Psychologist/Psychologie canadienne **5**(4), 225 (1964)
34. Smit, S., Eiben, A.: Parameter tuning of evolutionary algorithms: Generalist vs. specialist. Applications of Evolutionary Computation pp. 542–551 (2010)
35. Tonella, P.: Evolutionary testing of classes. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 119–128 (2004)
36. Vos, T., Baars, A., Lindlar, F., Kruse, P., Windisch, A., Wegener, J.: Industrial scaled automated structural testing with the evolutionary testing tool. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 175–184 (2010)
37. Wappler, S., Lammermann, F.: Using evolutionary algorithms for the unit testing of object-oriented software. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1053–1060. ACM (2005)
38. Whitley, D.: An overview of evolutionary algorithms: practical issues and common pitfalls. Information and Software Technology **43**(14), 817–831 (2001)
39. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997)