On Parameter Tuning in Search Based Software Engineering

Andrea Arcuri[1] and Gordon Fraser[2]

[1] Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no
[2] Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Abstract. When applying search-based software engineering (SBSE) techniques one is confronted with a multitude of different parameters that need to be chosen: Which population size for a genetic algorithm? Which selection mechanism to use? What settings to use for dozens of other parameters? This problem not only troubles users who want to apply SBSE tools in practice, but also researchers performing experimentation – how to compare algorithms that can have different parameter settings? To shed light on the problem of parameters, we performed the largest empirical analysis on parameter tuning in SBSE to date, collecting and statistically analysing data from more than a million experiments. As case study, we chose test data generation, one of the most popular problems in SBSE. Our data confirm that tuning does have a critical impact on algorithmic performance, and over-fitting of parameter tuning is a dire threat to external validity of empirical analyses in SBSE. Based on this large empirical evidence, we give guidelines on how to handle parameter tuning.

Key words: Search based software engineering, test data generation, object-oriented, unit testing

1 Introduction

Recent years have brought a large growth of interest in search based software engineering (SBSE) [1], especially in software testing [2]. The field has even matured to a stage where industrial applications have started to appear [3, 4]. 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 (roulette wheel, tournament, rank-based, etc.), type of crossover (single point, multi-point, etc.), 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 [5]: All algorithms perform on average equally on *all* possible problems. For any problem an algorithm is good at solving, you can always find another problem 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, as 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 [1]. In particular, we consider test data generation for object-oriented software using the EvoSuite tool [6], 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). We chose to consider five parameter settings (e.g., population size and crossover rate). To make the experiments finish in feasible time, we only considered 20 software classes as case study (previous empirical analyses of EvoSuite were based on thousands of different classes [6]). Still, this led to more than *one million* experiments that 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, and unfortunately *very* sensitive to the chosen case study. This brings to a compulsory use of *machine learning* techniques [7] if one wants to evaluate tuning in a sound scientific way. Furthermore, a 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 study and the empirical analysis. 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

Eiben *et al.* [8] 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? On the other hand, 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. In this paper we only deal with parameter tuning. Parameter control is a promising area of research, but mainly unexplored in SBSE.

Recently, Smit and Eiben [9] 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 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 [10]) and response surface methodology (e.g., used in [11]). The goal of this paper is to study the effects of parameter tuning, which includes also the cases of sub-optimal choices. Such type of analysis requires an exhaustive evaluation. 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.

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 would need to tune the algorithm by herself, as default settings are likely to bring to poor performance. To help practitioners in making such tuning, there exist frameworks

such as GUIDE [12]. 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 [3]. As an example, a 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 are in the conditions to claim 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. 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.

The test cases may have variable length [13], and so earlier approaches to testing object-oriented software made use of method sequences [14, 15] or strongly typed genetic programming [16, 17]. In our experiments, we used the EVOSUITE [6] 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.

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 α is chosen from $[0,1]$, and the first offspring O_1 contains the first $\alpha|P_1|$ test cases from the first parent, followed by the last $(1-\alpha)|P_2|$ test cases from the second parent. The second offspring O_2 contains the first $\alpha|P_2|$ test cases from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent.

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.

If a test case is mutated, then three types of operations are applied 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 $\mathcal{T}$. The initial population of test cases is generated randomly, by repeatedly performing the insertion operator also used to mutate test cases.

The search objective we chose is branch coverage, which requires that a test suite exercises a program in such a way that every condition (if, while, etc.) evaluates to true and to false. The fitness function is based on the well-established branch distance [18], 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 distancesof 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 [6]. Such an approach is particularly useful to reduce the negative effects of infeasible targets for the search.

We applied several bloat control techniques [19] to avoid that the size of individuals becomes bloated during the search.

In the experiments presented in this paper, we investigated five parameters of the search, which are not specific to this application 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.

In addition to these parameters, another important decision in a genetic algorithm is when to stop the search, as it cannot be assumed that an optimal solution is always found. 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.

4 Experiments

In this paper, we use as case study a subset of 20 Java classes out of those previously used to evaluate EvoSuite [6]. 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 on which EvoSuite may still achieve high coverage 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.

We investigated 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. Furthermore, because the goal of this paper is to study the effects of tuning, we analyzed all the possible combinations of the selected parameters. On the other hand, 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.

Another important factor is the *search budget*. A search algorithm can be run for any 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. On the other hand, a large population can 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. In fact, the search budget is perhaps the only parameter a practitioner should set. 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. On the other hand, a search could then be left running overnight, and results collected the morning after. In these two situations, the parameter settings (e.g. population size) should be different. In this paper, we consider 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 consider the cases of a budget that is a tenth ($10{,}000$) and ten times bigger ($1{,}000{,}000$).

For each class in the case study, we run 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 reasons of space, in this paper we only consider coverage, in particular branch coverage.

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 [20]. 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.

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 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 collected from this large empirical study could be used to address *several* research questions. Unfortunately, for reasons of space we only focus on the four that we believe are most important.

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 tuning is not really important. But, in most cases, there is a very large difference between the worst and best configuration (e.g., BellmanFordIterator). 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.

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	0.49	0.33	0.86	**0.93**	**0.00**
DateParse	1.00	1.00	1.00	0.50	0.50
Triangle	1.00	0.60	1.00	**0.70**	0.50
XMLElement	0.90	0.43	0.97	**1.00**	**0.10**
ZipOutputStream	1.00	0.47	1.00	**0.77**	0.50
CommandLine	0.41	0.11	0.59	**0.98**	0.34
Remainder	0.82	0.30	0.98	**0.98**	**0.13**
Industry1	0.95	0.53	0.98	**1.00**	**0.18**
Industry2	0.90	0.42	0.95	**1.00**	**0.11**
Attribute	0.47	0.21	0.90	**1.00**	**0.00**
DoubleMetaphone	0.63	0.22	0.96	**1.00**	**0.00**
Chronology	0.77	0.43	0.94	**1.00**	**0.00**
ArrayList	1.00	0.67	1.00	**0.67**	0.50
DateTime	0.60	0.21	0.95	**1.00**	**0.00**
TreeMap	0.65	0.00	0.78	**0.93**	**0.27**
Bessj	0.65	0.42	0.95	**1.00**	**0.00**
BellmanFordIterator	0.13	0.00	1.00	0.57	**0.07**
TTestImpl	0.55	0.21	1.00	**0.88**	**0.00**
LinkedListMultimap	0.81	0.18	1.00	**1.00**	**0.03**
FastFourierTransformer	1.00	0.29	1.00	**0.98**	0.47

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: If we tune a search algorithm based on a set of classes, how will its performance be on other new 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.

Of particular interest are the $\hat{A}_{TD}$ values. In three cases they are equal to 0.5 (so no difference between tuned and default settings), in seven cases they are higher than 0.5 (so tuning helps), but then in 10 cases they are lower than 0.5 (but only in four cases there is statistically significant difference). This means that, on the case study used in this paper, doing tuning is even *worse* than just using some arbitrary settings coming from the literature! This might be explained with the concept of *over-fitting* in machine learning [7]. A too intensive tuning on a set of problem instances can result in parameter settings that are too specific for that set. Even the case of 19 problem instances, as done in this paper, is too small to avoid such type of over-fitting.

Tuning should be done on a very large sample of problem instances. Otherwise, the obtained parameter settings are likely to be worse than arbitrary default values.

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 LinkedListMultimap) 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. For example, for DateParse, already a budget of $100{,}000$ is enough to get no difference between best, worst and default configuration. On the other hand, with a search budget of $1{,}000{,}000$, for example for CommandLine there is still a statistically strong difference.

As said, 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 again the case of CommandLine: At budget $10{,}000$, the $\hat{A}_{DW}$ is not statistically significant (i.e., close to 0.5 and Mann-Whitney U-test has

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	0.78	**0.98**	**0.27**
DateParse	1.00	0.50	0.50
Triangle	1.00	0.50	0.50
XMLElement	0.81	0.40	**0.11**
ZipOutputStream	0.93	0.47	0.47
CommandLine	0.38	0.32	**0.22**
Remainder	0.62	**0.23**	**0.05**
Industry1	0.90	**0.24**	**0.08**
Industry2	0.84	0.30	**0.17**
Attribute	0.52	**0.75**	**0.00**
DoubleMetaphone	0.57	**0.08**	**0.00**
Chronology	0.87	**0.76**	**0.28**
ArrayList	1.00	0.50	0.50
DateTime	0.93	**1.00**	0.30
TreeMap	0.32	0.33	**0.26**
Bessj	0.81	**0.92**	**0.18**
BellmanFordIterator	0.00	0.43	**0.00**
TTestImpl	0.68	**0.93**	**0.03**
LinkedListMultimap	0.98	**0.96**	**0.33**
FastFourierTransformer	0.97	**0.28**	**0.25**

p-value greater than 0.05), whereas it gets higher (close to 1) for $100{,}000$ and then for $1{,}000{,}000$. For $\hat{A}_{DB}$, it is statistically significant when budget is $10{,}000$, but not when we increase the budget to $100{,}000$. Interestingly, it comes back to be statistically significant at $1{,}000{,}000$, with an effect size that is even stronger than in the case of budget $10{,}000$. How come? The reason is that the testing targets have different difficulty to be covered. Even with an 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. But such an information is usually not known before doing tuning.

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

5 Guidelines

The empirical analysis carried out in this paper clearly shows that tuning has a strong impact on search algorithm performance, and if it is not done properly, there are dire

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	**0.77**	**0.07**	**0.93**	**0.00**	**0.82**	**0.11**
DateParse	**0.63**	0.50	0.50	0.50	0.50	0.50
Triangle	**0.67**	0.50	**0.70**	0.50	**0.69**	0.50
XMLElement	**0.81**	**0.07**	**1.00**	**0.10**	**1.00**	0.50
ZipOutputStream	**0.87**	0.43	**0.77**	0.50	**0.71**	0.50
CommandLine	0.54	**0.23**	**0.98**	0.34	**1.00**	**0.00**
Remainder	**0.72**	**0.21**	**0.98**	**0.13**	**1.00**	0.46
Industry1	0.63	**0.00**	**1.00**	**0.18**	-	-
Industry2	**0.82**	**0.06**	**1.00**	**0.11**	**1.00**	0.42
Attribute	**0.80**	**0.06**	**1.00**	**0.00**	**1.00**	**0.15**
DoubleMetaphone	**0.87**	**0.06**	**1.00**	**0.00**	**0.92**	**0.14**
Chronology	**0.90**	**0.08**	**1.00**	**0.00**	**1.00**	**0.17**
ArrayList	**0.70**	0.43	**0.67**	0.50	**1.00**	0.50
DateTime	0.69	**0.06**	**1.00**	**0.00**	**0.88**	0.45
TreeMap	0.60	**0.24**	**0.93**	**0.27**	**1.00**	**0.27**
Bessj	**0.83**	**0.10**	**1.00**	**0.00**	**1.00**	0.33
BellmanFordIterator	0.50	**0.00**	0.57	**0.07**	-	-
TTestImpl	**0.88**	**0.21**	**0.88**	**0.00**	**0.95**	**0.31**
LinkedListMultimap	0.60	**0.05**	**1.00**	**0.03**	**0.96**	0.50
FastFourierTransformer	**0.83**	**0.00**	**0.98**	0.47	-	-

risks in ending up with tuned configurations that are worse than suggested values in the literature. To avoid these problems, it would hence be important to use machine learning techniques [7] 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 [7].

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 [11]). 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 [7]. 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. On the other hand, 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 15 times with different random seeds, and we followed rigorous statistical procedures to evaluate their results.

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 as 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. However, to the best of our knowledge, there is no standard benchmark in test data generation for object-oriented software that we could have rather used.

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 has 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 focus on test data generation for object-oriented software using the EVOSUITE tool [6].

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 arbitrary parameter settings can lead to sub-optimal search performance. Even if one does a first phase of parameter tuning on some problem instances, the results on new problem instances can be very poor, even worse than arbitrary settings. Hence, tuning should be done on (very) large samples of problem instances. The main contribution of this paper is that it provides compelling empirical evidence to support these claims based on rigorous statistical methods.

To entail technology transfer to industrial practice, parameter tuning is a task of responsibility of who develops and releases search based tools. It is hence important to have *large* tuning phases on which *several* problem instances are employed. Unfortunately, parameter tuning phases can result in over-fitting issues. To validate whether a search based tool can be effective in practice once delivered to software engineers that will use it on their problem instances, it is important to use machine learning techniques [7] to achieve sound scientific conclusions. For example, tuning can be done on a subset of the case study (i.e., the so called *training set*), whereas performance evaluation would be done a on a separate and independent set (i.e., the so called *test set*). This would reduce the dire threats to external validity coming from over-fitting the parameter tuning. To the best of our knowledge, in the literature of search based software engineering, in most of the cases parameter tuning is either not done, done on the *entire* case study at hand, or its details are simply omitted.

Another 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.

Acknowledgements

Andrea Arcuri is supported by the Norwegian Research Council. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany.

References

1. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, King's College (2009)

2. 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 **36**(6) (2010) 742–762
3. 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). (2010) 175–184
4. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box system testing of real-time embedded systems using random and search-based testing. In: ICTSS'10: Proceedings of the IFIP International Conference on Testing Software and Systems, Springer (2010) 95–110
5. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1) (1997) 67–82
6. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: International Conference On Quality Software (QSIC). (2011)
7. Mitchell, T.: Machine Learning. McGraw Hill (1997)
8. Eiben, A., Michalewicz, Z., Schoenauer, M., Smith, J.: Parameter control in evolutionary algorithms. Parameter Setting in Evolutionary Algorithms (2007) 19–46
9. Smit, S., Eiben, A.: Parameter tuning of evolutionary algorithms: Generalist vs. specialist. Applications of Evolutionary Computation (2010) 542–551
10. Bartz-Beielstein, T., Markon, S.: Tuning search algorithms for real-world applications: A regression tree based approach. In: IEEE Congress on Evolutionary Computation (CEC). (2004) 1111–1118
11. 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)
12. Da Costa, L., Schoenauer, M.: Bringing evolutionary computation to industrial applications with GUIDE. In: Genetic and Evolutionary Computation Conference (GECCO). (2009) 1467–1474
13. 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 (2011) http://doi.ieeecomputersociety.org/10.1109/TSE.2011.44.
14. Tonella, P.: Evolutionary testing of classes. In: ISSTA'04: Proceedings of the ACM International Symposium on Software Testing and Analysis, ACM (2004) 119–128
15. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. In: ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis, ACM (2010) 147–158
16. Wappler, S., Lammermann, F.: Using evolutionary algorithms for the unit testing of object-oriented software. In: GECCO'05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, ACM (2005) 1053–1060
17. Ribeiro, J.C.B.: Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In: GECCO'08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation, ACM (2008) 1819–1822
18. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2) (2004) 105–156
19. 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). (2011)
20. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: IEEE International Conference on Software Engineering (ICSE). (2011)