A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering[1]

Andrea Arcuri[1] and Lionel Briand[2]

(1) Simula Research Laboratory, P.O. Box 134, Lysaker, Norway.
Email: arcuri@simula.no
(2) SnT Centre, University of Luxembourg, 6 rue Richard Coudenhove-Kalergi, L-1359, Luxembourg
Email: lionel.briand@uni.lu

Abstract

Randomized algorithms are widely used to address many types of software engineering problems, especially in the area of software verification and validation with a strong emphasis on test automation. However, randomized algorithms are affected by chance, and so require the use of appropriate statistical tests to be properly analyzed in a sound manner. This paper features a systematic review regarding recent publications in 2009 and 2010 showing that, overall, empirical analyses involving randomized algorithms in software engineering tend to not properly account for the random nature of these algorithms. Many of the novel techniques presented clearly appear promising, but the lack of soundness in their empirical evaluations casts unfortunate doubts on their actual usefulness. In software engineering, though there are guidelines on how to carry out empirical analyses involving human subjects, those guidelines are not directly and fully applicable to randomized algorithms. Furthermore, many of the text books on statistical analysis are written from the viewpoints of social and natural sciences, which present different challenges from randomized algorithms. To address the questionable overall quality of the empirical analyses reported in the systematic review, this paper provides guidelines on how to carry out and properly analyze randomized algorithms applied to solve software engineering tasks, with a particular focus on software testing which is by far the most frequent application area of randomized algorithms within software engineering.

Keyword: Statistical difference, effect size, parametric test, non-parametric test, confidence interval, Bonferroni adjustment, systematic review, survey.

1 Introduction

Many problems in software engineering can be alleviated through automated support. For example, automated techniques exist to generate test cases that satisfy some desired coverage criteria on the system under test, such as for example branch [58] and path coverage [51]. Because often these problems are undecidable, deterministic algorithms that are able to provide optimal solutions in reasonable time do not exist. The use of heuristics, implemented as randomized algorithms [86], is hence necessary to address this type of problems.

At a high level, a randomized algorithm is an algorithm that has one or more of its components based on randomness. Therefore, running twice the same randomized algorithm on the same problem instance may yield different results. The most well-known example of randomized algorithm in software engineering is perhaps *random testing* [31, 13]. Techniques that use random testing are of course randomized, as for example DART [51] (which combines random testing with symbolic execution). Furthermore, there is a large body of work on the application of *search algorithms* in software engineering [57], as for example Genetic Algorithms. Since search algorithms are typically randomized and numerous software engineering problems can be

[1]This paper is an extension of a conference paper [10] published in the International Conference on Software Engineering (ICSE), 2011.

addressed with search algorithms, randomized algorithms therefore play an increasingly important role. Applications of search algorithms include software testing [81], requirement engineering [18], project planning and cost estimation [2], bug fixing [14], automated maintenance [84], service-oriented software engineering [22], compiler optimisation [26] and quality assessment [67].

A randomized algorithm may be strongly affected by chance. It may find an optimal solution in a very short time or may never converge towards an acceptable solution. Running a randomized algorithm twice on the same instance of a software engineering problem usually produces different results. Hence, researchers in software engineering that develop novel techniques based on randomized algorithms face the problem of how to properly evaluate the effectiveness of these techniques.

To analyze the cost and effectiveness of a randomized algorithm, it is important to study the *probability distribution* of its output and various performance metrics [86]. Though a practitioner might want to know what is the execution time of those algorithms *on average*, this might be misleading as randomized algorithms can yield very complex and high variance probability distributions.

The probability distribution of a randomized algorithm can be analyzed by running such an algorithm several times in an independent way, and then collecting appropriate data about its results and performance. For example, consider the case in which one wants to trigger failures by applying random testing (assuming that an automated oracle is provided) on a specific software system. As a way to assess its cost and effectiveness, test cases can be sampled at random until the first failure is detected. For example, in the first experiment, a failure might be detected after sampling 24 test cases. Assume the second run of the experiment (if a pseudo-random generator is employed, there would be the need to use a different seed for it) triggers the first failure when executing the second random test case. If in a third experiment the first failure is obtained after generating 274 test cases, the *mean* value of these three experiments would be 100. Using such a mean to characterize the performance of random testing on a set of programs would clearly be misleading given the extent of its variation.

Since randomness might affect the reliability of conclusions when performing the empirical analysis of randomized algorithms, researchers hence face two problems: (1) how many experiments should be run to obtain reliable results, and (2) how to assess in a rigorous way whether such results are indeed reliable. The answer to these questions lies in the use of *statistical tests*, and there are many books on their various aspects (e.g., [99, 25, 71, 55, 119]). Notice that though statistical testing is used in most if not all scientific domains (e.g., medicine and behavioral sciences), each field has its own set of constraints to work with. Even within a field like software engineering the application context of statistical testing can vary significantly. When human resources and factors introduce randomness (e.g., [33, 63]) in the phenomena under study, the use of statistical tests is also required. But the constraints a researcher would work with are quite different from those of randomized algorithms, such as for example the size of data samples and the types of distributions.

Because of the widely varying situations across domains and the overwhelming number of statistical tests, each one with its own characteristics and assumptions, many practical guidelines have been provided targeting different scientific domains, such as biology [89] and medicine [64]. There are also guidelines for running experiment with human subjects in software engineering [120]. In this paper, the intent is to do the same for randomized algorithms in software engineering, with a particular focus on verification and validation, as they entail specific issues regarding the application of statistical testing.

To assess whether the results obtained with randomized algorithms are properly analyzed in software engineering research, and therefore whether precise guidelines are required, a systematic review was carried out. The analyses were limited to the years 2009 and 2010, as the goal was not to perform an exhaustive review of all research that was ever published but rather to obtain a recent, representative sample on which to draw conclusions about current practices. The focus was on research venues that deal with all aspects of software engineering, such as IEEE Transactions of Software Engineering (TSE), IEEE/ACM International Conference on Software Engineering (ICSE) and International Symposium on Search Based Software Engineering (SSBSE). The former two are meant to get an estimate of the extent to which randomized algorithms are used in software engineering. The latter, more specialized venue provides additional insight into the way randomized algorithms are assessed in software engineering. Furthermore, because randomized algorithms are more commonly used in software testing, the journal Software Testing, Verification and Reliability (STVR) was also taken into account. The review shows that, in many cases, statistical analyses are either missing, inadequate, or incomplete. For example, though journal guidelines in medicine require a mandatory use of standardized *effect size* measurements [55] to quantify the effect of treatments, only one case was found in which a standardized effect size was used to measure the relative effectiveness of a randomized algorithm [96]. Even more surprising, in many of the surveyed empirical analyses, randomized algorithms were evaluated based on the results of only one run. Only few empirical studies reported the use of statistical analysis.

Given the results of this survey, it was necessary to devise *practical* guidelines for the use of statistical testing in assessing randomized algorithms in software engineering applications. Note that, though guidelines have been provided for other scientific domains [89, 64] and for other types of empirical analyses in software engineering [33, 63], they are not directly applicable and complete in the context of randomized algorithms. The objective of this paper is therefore to account for the specific properties of randomized algorithms in software engineering applications.

Notice that Ali *et al.* [3] have recently carried out a systematic review of search-based software testing which includes some limited guidelines on the use of statistical testing. This paper builds upon that work by: (1) analyzing software engineering as whole and not just software testing, (2) considering all types of randomized algorithms and not just search algorithms, and (3) giving precise, practical, and complete suggestions on many aspects related to statistical testing that were either not discussed or just briefly mentioned in the work of Ali *et al.* [3].

The main contributions of this paper can be summarized as follows:

- A systematic review is performed on the current state of practice of the use of statistical testing to analyze randomized algorithms in software engineering. The review shows that randomness is not properly taken into account in the research literature.
- A set of practical guidelines is provided on the use of statistical testing that are tailored to randomized algorithms in software engineering applications, with a particular focus on verification and validation (including testing), and the specific properties and constraints they entail.

The paper is organized as follows. Section 2 discusses a motivating example. The systematic review follows in Section 3. Section 4 presents the concept of statistical difference in the context of randomized algorithms. Section 5 compares two kinds of statistical tests and discusses their implications on randomized algorithms. The problem of censored data and how it applies to randomized algorithms is discussed in Section 6. How to measure effect sizes and therefore the practical impact of randomized algorithms is presented in Section 7. Section 8 investigates the question of how many times randomized algorithms should be run. The problems associated with multiple tests are discussed in Section 9, whereas Section 10 deals with the choice of artifacts, which has usually a significant impact on results. Practical guidelines on how to use statistical tests are summarized in Section 11. The threats to validity associated with the work presented in this paper are discussed in Section 12. Finally, Section 13 concludes the paper.

2 Motivating Example

In this section, a motivating example is provided to show why the use of statistical tests is a necessity in the analyses of randomized algorithms in software engineering. Assume that two techniques $\mathcal{A}$ and $\mathcal{B}$ are used in a type of experiment in which the output is binary: either *pass* or *fail*. For example, in the context of software testing, $\mathcal{A}$ and $\mathcal{B}$ could be testing techniques (e.g., random testing [31, 13]), and the experiment would determine whether they trigger or not any failure given a limited testing budget. The technique with highest *success rate*, that is failure rate in the testing example, would be considered to be superior. Further assume that both techniques are run n times, and a represents the times $\mathcal{A}$ was successful, wheres b is the number of successes for $\mathcal{B}$. The *estimated* success rates of these two techniques are defined as a/n and b/n, respectively. A related example in software testing (in which success rates are compared) that currently seems very common in industry (especially for online companies such as Google and Amazon) is “A/B testing”[2].

Now, consider that such experiment is repeated $n = 10$ times, and the results show that $\mathcal{A}$ has a 70% estimated success rate, whereas $\mathcal{B}$ has a 50% estimated success rate. Would it be safe to conclude that $\mathcal{A}$ is better than $\mathcal{B}$? Even if $n = 10$ and the difference in estimated success rates is quite large (i.e., 20%), it would

[2] en.wikipedia.org/wiki/A/B_testing, accessed October 2012.

actually be unsound to draw any conclusion about the respective performance of the two techniques. Because this might not be intuitive, the exact mathematical reasoning is provided below to explain the above statement.

A series of repeated n experiments with binary outcome can be described as a *binomial distribution* [36], where each experiment has probability p of success, and the mean value of the distribution (i.e., number of successes) is pn. In the case of $\mathcal{A}$, one would have an estimated success rate $p = a/n$ and an estimated number of successes $pn = a$. The probability mass function of a binomial distribution $B(n,p)$ with parameters n and p is:

$$P(B(n,p) = k) = \binom{n}{k} p^k (1-p)^{n-k} \ .$$

$P(B(n,p) = k)$ represents the probability that a binomial distribution $B(n,p)$ would result in k successes. Exactly k runs would be successful (probability p^k) while the others $n-k$ would fail (probability $(1-p)^{n-k}$). Since the order of successful experiments is not important, there are $\binom{n}{k}$ possible orders. Using this probability function, what is the probability that a equals the expected number of successes? Considering the example provided in this section, having a technique with an *actual* 70% success rate, what is the probability of having exactly 7 successes out of 10 experiments? This can be calculated with:

$$P(B(10,0.7) = 7) = \binom{10}{7} 0.7^7 (0.3)^3 = 0.26 \ .$$

This example shows that there is only a 26% chance to have exactly $a = 7$ successes if the actual success rate is 70%! This shows a potential misconception: expected values (e.g., successes) often have a relatively low probability of occurrence. Similarly, the probability that both techniques have a number of successes equal to their expected value would be even lower:

$$P(B(10,0.7) = 7) \times P(B(10,0.5) = 5) = 0.06 \ .$$

Reversely, even if one obtains $a = 7$ and $b = 5$, what would be the probability that both techniques have an equal actual success rate of 60%? We would have:

$$P(B(10,0.6) = 7) \times P(B(10,0.6) = 5) = 0.04 \ .$$

Though 0.04 seems a rather "low" probability, it is not much lower than 0.06, the probability of the observed number of successes to be actually equal to their expected values. Therefore, one cannot really say that the hypothesis of equal actual success rates (60%) is much more implausible than the one with 70% and 50% actual success rates. But what about the case where the two techniques have exactly the same actual success rate equal to 0.2? Or what about the cases in which $\mathcal{B}$ would actually have a better actual success rate than $\mathcal{A}$? What would be the probability for these situations to be true? Figure 1 shows all these probabilities, when $a = 0.7n$ and $b = 0.5n$, for two different numbers of runs: $n = 10$ and $n = 100$. For $n = 10$, there is a great deal of variance in the probability distribution of success rates. In particular, the cases in which $\mathcal{B}$ has a higher actual success rate do not have a negligible probability. On the other hand, in the case of $n = 100$, the variance has decreased significantly. This clearly shows the importance of using sufficiently large samples, an issue that will be covered in more detail later in the paper.

In this example, with $n = 100$, the use of statistical tests (e.g., Fisher Exact test) would yield strong evidence to conclude that $\mathcal{A}$ is better than $\mathcal{B}$. At an intuitive level, a statistical test would estimate the probability of mistakenly drawing the conclusion that $\mathcal{A}$ is better than $\mathcal{B}$, under the form of a so-called *p*-value, as further discussed later in the paper. The resulting *p*-value would be quite small for $n = 100$ (i.e., 0.005), whereas for $n = 10$ it would far much larger (i.e. 0.649), thus confirming and quantifying what is graphically visible in Figure 1. So even for what might appear to be large values of n, the capability to draw reliable conclusions could still be weak. Though some readers might find the above example rather basic, the fact of the matter is that many papers reporting on randomized algorithms overlook the principles and issues illustrated above.

3 Systematic Review

Systematic reviews are used to gather, in an unbiased and comprehensive way, published research on a specific subject and analyze it [65]. Systematic reviews are a useful tool to assess general trends in published research,

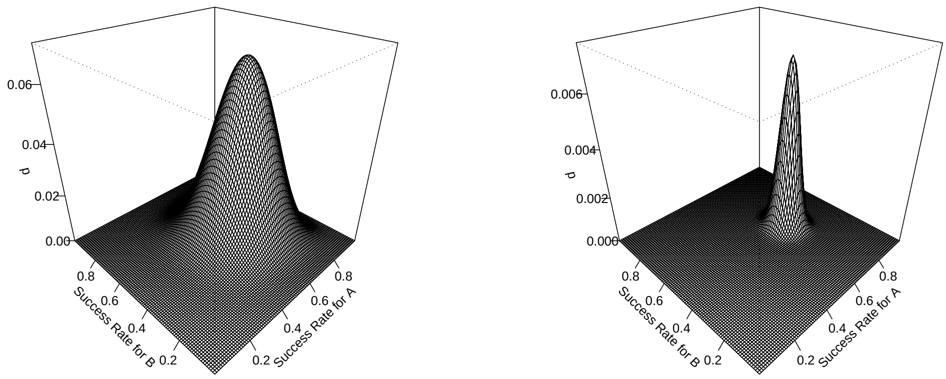

Figure 1: Probabilities to obtain $a = 0.7n$ and $b = 0.5n$ when $n = 10$ (left) and $n = 100$ (right) for different success rates of the algorithms $\mathcal{A}$ and $\mathcal{B}$.

and they are becoming increasingly common in software engineering [70, 33, 63, 3].

The systematic review reported in this paper aims at analyzing: (RQ1) how often randomized algorithms are used in software engineering, (RQ2) how many runs were used to collect data, and (RQ3) which types of statistical analyses were used for data analysis.

To answer RQ1, two of the main venues that deal with all aspects of software engineering were selected: IEEE Transactions of Software Engineering (TSE) and IEEE/ACM International Conference on Software Engineering (ICSE). The International Symposium on Search-Based Software Engineering (SSBSE) was also considered, which is a specialized venue devoted to the application of search algorithms in software engineering. Furthermore, because many of the applications of randomized algorithms are in software testing, the journal Software Testing, Verification and Reliability (STVR) was included as well. Because the goal of this paper is not to perform an exhaustive survey of published works, but rather to get an up-to-date snapshot of current practices regarding the application of randomized algorithms in software engineering research, only 2009 and 2010 publications were included.

Only full length research papers were retained and, as a result, 77 papers at ICSE and 11 at SSBSE were excluded. A total of 246 papers were considered: 96 in TSE, 104 in ICSE, 23 in SSBSE and 23 in STVR. These papers were manually checked to verify whether they made use of randomized algorithms, thus leading to a total of 54 papers. The number of analyzed papers is in line with other systematic reviews (e.g., in the work of Ali *et al.* [3] a total of 68 papers were analyzed). For example, in their systematic review on systematic reviews in software engineering, Kitchenham *et al.* [70] show that 11 out 20 systematic reviews involved less than 54 publications. Table 1 summarizes the details of the systematic review divided by venue and year.

Notice that papers were excluded if it was not clear whether randomized algorithms were used. For example, the techniques described in the work of Hsu and Orso [60] and the work of Thum *et al.* [112] use external SAT solvers, and those might be based on randomized algorithms, though it was not possible to tell with certainty. Furthermore, papers that involve *machine learning* algorithms that are randomized were not considered since they require different types of analysis [85]. On the other hand, if a paper focused on presenting a deterministic, novel technique, then it was included when randomized algorithms were used for comparison purposes (e.g., fuzz testing [43]). Table 2 (for the year 2009) and Table 3 (for the year 2010) summarize the results of this systematic review for the final selection of 54 papers. The first clearly visible result is that randomized algorithms are widely used in software engineering (RQ1): they were found in 15% of the regular articles in TSE and ICSE, which are general-purpose and representative software engineering venues. More specifically, 72% of all the papers (i.e., 39 out of 54) are on verification and validation (V&V).

To answer RQ2, the data in Table 2 and Table 3 shows the number of times a technique was run to collect data regarding its performance on each artifact in the case study. Only 27 cases out of 54 show at least 10 runs.

Table 1: Number of publications grouped by venue, year and type.

Venue	Year	All	Regular	Randomized Algorithms
TSE	2009	48	48	3
	2010	48	48	12
ICSE	2009	70	50	4
	2010	111	54	10
SSBSE	2009	17	9	9
	2010	17	14	11
STVR	2009	12	12	4
	2010	11	11	1
Total		334	246	54

In many cases, data are collected from only one run of the randomized algorithms. Furthermore, notice that the case in which randomized algorithms are evaluated based on *only one run per case study artifact* is quite common in the literature. Even very influential papers, such as DART [51], feature this problem which poses serious threats to the validity of their reported empirical analyses.

In the literature, there are empirical analyses in which randomized algorithms are run only once per case study artifact, but a large number of artifacts were generated at random (e.g., [90, 118]). The validity of such empirical analyses depends on the representativeness of instances created with the random generator. At any rate, the choice of a case study that is statistically appropriate, and its relations to the required number of runs for evaluating a randomized algorithm, needs careful consideration and will be discussed in more detail in Section 10.

Regarding RQ3, only 19 out of 54 articles include empirical analyses supported by some kind of statistical testing. More specifically, those are *t*-tests, Welch and U-tests when algorithms are compared in a pairwise fashion, whereas ANOVA and Kruskal-Wallis are used for multiple comparisons. Furthermore, in some cases linear regression is employed to build prediction models from a set of algorithm runs. However, in only one article [96] standardized *effect size* measures (see Section 7) are reported to quantify the relative effectiveness of algorithms.

Results in Table 2 and 3 clearly show that, when randomized algorithms are employed, empirical analyses in software engineering do not properly account for their random nature. Many of the novel proposed techniques may indeed be useful, but the results in Table 2 and 3 cast serious doubts on the validity of most existing results.

Notice that some of empirical analyses in Table 2 and 3 do not use statistical tests since they do not perform any comparison of the technique they propose with alternatives. For example, in the award winning paper at ICSE 2009, a search algorithm (i.e., Genetic Programming) was used and was run 100 times on each artifact in the case study [117]. However this algorithm was not compared against simpler alternatives or even random search (e.g., successful applications of automated bug fixing on real-world software can be traced back at least down to the work of Griesmayer *et al.* [54]). When looking more closely at the reported results in order to assess the implications of such lack of comparison, one would see that the total number of fitness evaluations was 400 (a population size of 40 individuals that is evolved for 10 generations). This sounds like a very low number (for example, for test data generation in branch coverage, it is common to see 100,000 fitness evaluations for *each* branch [58]) and one can therefore conclude that there is very limited search taking place. This implies that a random search might have yielded similar results, and this would have warranted a comparison with random search. This is directly confirmed in the reported results in the work of Weimer *et al.* [117], in which in half of the subject artifacts in the case study, the average number of fitness evaluations per run is at most 41, thus implying that, on average, appropriate patches are found in the random initialization of the first population before the actual evolutionary search even starts.

As the search operators were tailored to specific types of bugs, then the choice of the case study and its representativeness play a major role in assessing the validity of an empirical study (more details in Section 10). Therefore, as discussed by Ali *et al.* [3], a search algorithm should always be compared against at least random search in order to check that success is not due to the search problem (or case study) being easy. Notice, however, that the previous work on automated bug fixing does not seem to feature comparisons neither (e.g.,

Table 2: Results of systematic review for the year 2009.

Reference	Venue	V&V	Repetitions	Statistical Tests
[1]	TSE	yes	1/5	U-test
[80]	TSE	yes	1	None
[90]	TSE	no	1	None
[83]	ICSE	no	100	*t*-test, U-test
[117]	ICSE	yes	100	None
[43]	ICSE	yes	1	None
[68]	ICSE	yes	1	None
[7]	SSBSE	yes	1000	Linear regression
[48]	SSBSE	yes	30/500	None
[32]	SSBSE	no	100	U-test
[46]	SSBSE	yes	50	None
[72]	SSBSE	yes	10	Linear regression
[66]	SSBSE	yes	10	None
[79]	SSBSE	yes	1	None
[69]	SSBSE	no	1	None
[106]	SSBSE	no	1	None
[21]	STVR	yes	1/100	None
[95]	STVR	yes	1	None
[104]	STVR	yes	1	None
[61]	STVR	yes	Undefined	None

see [111, 110, 54, 14]). The work of Weimer *et al.* [117] was discussed only because it was among the sampled papers in the systematic review, and it is a good example to point out the importance of comparisons.

Since comparisons with simpler alternatives (at a very minimum random search) is a necessity when one proposes a novel randomized algorithm or addresses a new software engineering problem [3], statistical testing should be part of all publications reporting such empirical studies. In this paper, specific guidelines are provided on how to use statistical tests to support comparisons among randomized algorithms. One might argue that, depending on the addressed problem and the aimed contribution, there might be cases when comparisons with alternatives are either not possible or unnecessary, thus removing the need for statistical testing. However, such cases should be rare and in any case not nearly as common as what can be observed in the systematic review.

4 Statistical Difference

When a novel randomized algorithm $\mathcal{A}$ is developed to address a software engineering problem, it is common practice to compare it against existing techniques, in particular simpler alternatives. For simplicity, consider the case in which just one alternative randomized algorithm (called $\mathcal{B}$) is used in the comparisons. For example, $\mathcal{B}$ can be random testing, and $\mathcal{A}$ can be a search algorithm such as Genetic Algorithms or an hybrid technique that combines symbolic execution with random testing (e.g., DART [51]).

To compare $\mathcal{A}$ versus $\mathcal{B}$, one first needs to decide which criteria are used in the comparisons. Many different measures (M), either attempting to capture the effectiveness or the cost of algorithms, can be selected depending on the problem at hand and contextual assumptions, e.g., source code coverage, execution time. Depending on the selected choice, one may want to either minimize or maximize M, for example maximize coverage and minimize execution time.

To enable statistical analysis, one should run both $\mathcal{A}$ and $\mathcal{B}$ a large enough number (n) of times, in an independent way, to collect information on the probability distribution of M for each algorithm. A *statistical test* should then be run to assess whether there is enough empirical evidence to claim, with a high level of confidence, that there is a difference between the two algorithms (e.g., $\mathcal{A}$ is better than $\mathcal{B}$). A *null hypothesis* H_0 is typically defined to state that there is no difference between $\mathcal{A}$ and $\mathcal{B}$. In such a case, a statistical test aims

Table 3: Results of systematic review for the year 2010.

Reference	Venue	V&V	Repetitions	Statistical Tests
[45]	TSE	yes	1000	None
[125]	TSE	yes	100	*t*-test
[58]	TSE	yes	60	U-test
[96]	TSE	yes	32	U-test, $\hat{A}_{12}$
[30]	TSE	yes	30	Kruskal-Wallis, undefined pairwise
[109]	TSE	no	20	None
[20]	TSE	no	10	U-test, *t*-test, ANOVA
[34]	TSE	no	3	U-test
[6]	TSE	yes	1	None
[16]	TSE	yes	1	None
[19]	TSE	yes	1	None
[118]	TSE	no	1	None
[74]	ICSE	yes	100	None
[126]	ICSE	yes	50	None
[50]	ICSE	yes	5	None
[87]	ICSE	yes	5	None
[42]	ICSE	yes	1	None
[56]	ICSE	yes	1	None
[62]	ICSE	no	1	None
[123]	ICSE	yes	1	None
[92]	ICSE	yes	1	None
[103]	ICSE	no	1	None
[28]	SSBSE	yes	100	*t*-test
[29]	SSBSE	no	100	None
[78]	SSBSE	no	50	*t*-test
[82]	SSBSE	yes	50	U-test
[122]	SSBSE	yes	30	U-test
[124]	SSBSE	yes	30	*t*-test
[75]	SSBSE	yes	30	Welch
[115]	SSBSE	no	30	ANOVA
[17]	SSBSE	yes	3/5	None
[77]	SSBSE	yes	3	None
[127]	SSBSE	no	1	None
[128]	STVR	yes	24/480	Linear regression

to verify whether one should reject the null hypothesis H_0. However, what aspect of the probability distribution of M is being compared depends on the used statistical test. For example, a *t*-test compares the mean values of two distributions whereas others tests focus on the median or proportions, as discussed in Section 5.

There are two possible types of error when performing statistical testing: (I) one rejects the null hypothesis when it is true (i.e., claiming that there is a difference between two algorithms when actually there is none), and (II) H_0 is accepted when it is false (there is a difference but the researcher claims the two algorithms to be equivalent). The *p*-value of a statistical test denotes the probability of a Type I error. The *significant level* α of a test is the highest *p*-value one accepts for rejecting H_0. A typical value, inherited from widespread practice in natural and social sciences, is $\alpha = 0.05$.

Notice that the two types of error are conflicting; minimizing the probability of one of them necessarily tends to increase the probability of the other. But traditionally there is more emphasis on not committing a Type I error, a practice inherited from natural sciences where the goal is often to establish the existence of a natural phenomenon in a conservative manner. In this context, one would only conclude that an algorithm $\mathcal{A}$ is better than $\mathcal{B}$ when the probability of a Type I error is below α. The price to pay for a small α value is that, when the data sample is small, the probability of a Type II error can be high. The concept of statistical *power* [25] refers to the probability of rejecting H_0 when it is false (i.e., the probability of claiming statistical difference when there is actually a difference).

Getting back to the comparison of techniques $\mathcal{A}$ and $\mathcal{B}$, assume one obtains a *p*-value equal to 0.06. Even if one technique seems significantly better than the other in terms of effect size (Section 7), the researcher would then conclude that there is no difference when using the traditional $\alpha = 0.05$ threshold. In software engineering, or in the context of *decision-making* in general, this type of reasoning can be counter-productive. The tradition of using $\alpha = 0.05$, discussed by Cowles [27], has been established in the early part of the last century, in the context of natural sciences, and is still applied by many across scientific fields. It has, however, an increasing number of detractors [52, 53] who believe that such thresholds are arbitrary, and that researchers should simply report *p*-values and let the readers decide in context what risks they are willing to take in their decision-making process.

When there is the need to make a choice between techniques $\mathcal{A}$ and $\mathcal{B}$, an engineer would like to use the technique that is more likely to outperform the other. If one is currently using $\mathcal{B}$, and a new technique $\mathcal{A}$ seems to show better results, then a high level of confidence (i.e., a low *p*-value) might be required before opting for the "cost" (e.g., buying licenses and training) of switching from $\mathcal{B}$ to $\mathcal{A}$. On the other hand, if the "cost" of applying the two techniques is similar, then whether one gets a *p*-value lower than α bears little consequence from a practical standpoint, as in the end an alternative *must* be selected, for example to test a system. However, as it will be shown in Section 8, obtaining *p*-values lower than $\alpha = 0.05$ should not be a problem when experimenting with randomized algorithms. The focus of such experiments should rather be on whether a given technique brings any practically significant advantage, usually measured in terms of an estimated effect size and its confidence interval, an important concept addressed in Section 7.

In practice, the selection of an algorithm would depend on the *p*-value of effectiveness comparisons, the effectiveness effect size, and the cost difference among algorithms (e.g., in terms of user-provided inputs or execution time). Given a context-specific decision model, the reader, using such information, could then decide which technique is more likely to maximize benefits and minimize risk. In the simplest case where compared techniques would have comparable costs, one would simply select the technique with the highest effectiveness regardless of the *p*-values of comparisons, even if as a result there is a non-negligible probability that it will bring no particular advantage.

When one has to carry out a statistical test, one must choose between *one-tailed* and a *two-tailed* test. Briefly, in a two-tailed test, the researcher would reject H_0 if the performance of $\mathcal{A}$ and $\mathcal{B}$ are different regardless of which one is the best. On the other hand, in a one-tailed test, the researcher is making assumptions about the relative performance of the algorithms. For example, one could expect that a new sophisticated algorithm $\mathcal{A}$ is better than a naive algorithm $\mathcal{B}$ used in the literature. In such a case, one would detect a statistically significant difference when $\mathcal{A}$ is indeed better than $\mathcal{B}$, but ignoring the "unlikely" case of $\mathcal{B}$ being better than $\mathcal{A}$. An historical example in the literature of statistics is the test to check whether there is the right percent of gold (carats) in coins. One could expect that a dishonest coiner might produce coins with lower percent of gold than declared, and so a one-tailed test would be used rather than a two-tailed. Such a test could be used if one wants to verify whether the coiner is actually dishonest, whereas giving more gold than declared would be very unlikely. Using a one-tailed test has the advantage, compared to a two-tailed test, that the resulting *p*-value is lower (so it is easier to detect statistically significant differences).

Are there cases in which a one-tailed test could be advisable in the analysis of randomized algorithms in software engineering? As a rule of thumb, the authors of this paper believe this is not the case: two-tailed tests should be used. One should use a one-tailed test only if he has strong arguments to support such a decision. In contrast to empirical analyses in software engineering involving human subjects, most of the time one cannot make any assumption on the relative performance of randomized algorithms. Even naive testing techniques such as random testing can fare better than more sophisticated techniques on some classes of problems (e.g., [105, 9]). The reason is that sophisticated novel techniques might incur extra computational overhead compared to simpler alternatives, and the magnitude of this overhead might not only be very high but also difficult to determine before running the experiments. Furthermore, search algorithms do exhibit complex behavior, which is dependent on the properties of the search landscape of the addressed problem. It is not uncommon for a novel testing technique to be better on certain types of software and worse on others. For example, an empirical analysis in software testing in which this phenomenon is visible with statistical confidence can be found in the work of Fraser and Arcuri [37]. In that paper, a novel technique for test data generation of object-oriented software was compared against the state of the art. Out of a total of 727 Java classes, the novel technique gave better results in 357 cases, but worse on 81 (on the remaining 289 classes there was no difference). In summary, if one wants to lower the *p*-values, it is recommended to have a large number of runs (see Section 8) when possible rather than using an arguable one-tailed test.

Assume that a researcher runs n experiments and does not obtain significant results. It might be then tempting to run an additional k experiments, and base the statistical analyses on those $n + k$ runs, in the hope of getting significant results as a result of increased statistical power. However, in this case, the k runs are not independent, as the choice of running them depended on the outcome of the first n runs. As a result, the real *p*-value ends up being higher than what is estimated by statistical testing. This problem and related solutions are referred to in the literature as "sequence statistical testing" or "sequential analysis", and have been applied in numerous fields such as repeated clinical trials [108]. In any case, if one wants to run k more experiments after analyzing the first n, it is important to always state it explicitly, as otherwise the reader would be misled when interpreting the obtained results.

5 Parametric vs Non-Parametric Tests

In the research context of this paper, the two most used statistical tests are the *t*-test and the Mann-Whitney U-test. These tests are in general used to compare two independent data samples (e.g., the results of running n times algorithm $\mathcal{A}$ compared to $\mathcal{B}$). The *t*-test is *parametric*, whereas the U-test is *non-parametric*.

A parametric test makes assumptions on the underlying distribution of the data. For example, the *t*-test assumes normality and equal variance of the two data samples. A non-parametric test makes no assumption about the distribution of the data. *Why* is there the need for two different types of statistical tests? A simple answer is that, in general, non-parametric tests are less powerful than parametric ones when the latter's assumptions are fulfilled. When, due to cost or time constraints, only small data samples can be collected, one would like to use the most powerful test available if its assumptions are satisfied.

There is a large body of work regarding which of the two types of tests should be used [35]. The assumptions of the *t*-test are in general not met. Considering that the variance of the two data samples is most of the time different, a Welch test should be used instead of a *t*-test. But the problem of the normality assumption remains.

An approach would be to use a statistical test to assess whether the data is normal, and, if the test is successful, then use a Welch test. This approach increases the probability of Type I error and is often not necessary. In fact, the Central Limit theorem tells that, for large samples, the *t*-test and Welch test are robust even when there is strong departure from a normal distribution [99, 102]. But in general one cannot know how many data points (n) he needs to reach reliable results. A rule of thumb is to have at least $n = 30$ for each data sample [99].

There are three main problems with such an approach: (1) if one needs to have a large n for handling departures from normality, then it might be advisable to use a non-parametric test since, for a large n, it is likely to be powerful enough; (2) the rule of thumb $n = 30$ stems from analyses in behavioral science and there is no supporting evidence of its efficacy for randomized algorithms in software engineering; (3) the Central

Limit theorem has its own set of assumptions, which are too often ignored. Points (2) and (3) will be now discussed in more details by accounting for the specific properties of the application of randomized algorithms in software engineering, with an emphasis on software testing.

5.1 Violation of Assumptions

Parametric tests make assumptions on the probability distributions of the analyzed data sets, but "The assumptions of most mathematical models are always false to a greater or lesser extent" [49]. Consider the following software testing example. A technique is used to find a test case for a specific testing target (e.g., a test case that triggers a failure or covers a particular branch/path), and then a researcher evaluates how many test cases X_i the technique requires to sample and evaluate before covering that target. This experiment can be repeated n times, yielding n observations $\{X_1, \ldots, X_n\}$ to study the probability distribution of the random variable X. Ideally, one would like a testing technique that minimizes X.

Since using the t-test assumes normality in the distribution X, are there cases for which it can be used to compare distributions of X resulting from different test techniques? The answer to this question is *never*. First, a normal distribution is continuous, whereas the number of sampled test cases X would be discrete. Second, the density function of the normal distribution is always positive for any value, whereas X would have zero probability for negative values. At any rate, asking whether a data set follows a normal distribution is not the right question [49]. A more significant question is what are the effects of departures from the assumptions on the validity of the tests. For example, a t-test returns a *p*-value that quantifies the probability of Type I error. The more the data departs from normality and equal variance, the more the resulting *p*-value will deviate from the true probability of Type I error.

Glass *et al.* [49] showed that in many cases the departures from the assumptions do not have serious consequences, particularly for data sets with not too high kurtosis (roughly, the kurtosis is a measure of infrequent extreme deviations). However, such empirical analyses reported and surveyed by Glass *et al.* [49] are based on social and natural sciences. For example, Glass *et al.* [49] wrote:

> "Empirical estimates of skewness and kurtosis are scattered across the statistical literature. Kendall and Stuart (1963, p. 57) reported the frequency distribution of age at marriage for over 300,000 Australians; the skewness and kurtosis were 1.96 and 8.33, respectively. The distribution of heights of 8,585 English males (see Glass & Stanley, 1970, p. 103) had skewness and kurtosis of -0.08 and 3.15, respectively".

Data sets for age at marriage and heights have known bounds (e.g., according to Wikipedia, the tallest man in world was 2.72 meters, whereas the oldest was 122 years old). As a result, extreme deviations are not possible. This is not true for software testing, where testing effort can drastically vary across software systems. For example, one can safely state that testing an industrial system is vastly more complex than testing a method implementing the triangle classification problem. None of the papers surveyed in Section 3 report skewness or kurtosis values. Although meta-analyses of the literature are hence not possible, the following arguments cast even further doubts about the applicability of parametric tests to analyze randomized algorithms in software testing.

Random testing is perhaps the easiest and most known automated software testing technique. It is often recommended as a comparison baseline to assess whether novel testing techniques are indeed useful [57]. When random testing is used to find a test case for a specific testing target (e.g., a test case that triggers a failure or covers a particular branch/path), it follows a geometric distribution. When there is more than one testing target, e.g., full structural coverage, it follows a coupon's collector problem distribution [13]. Given θ the probability of sampling a test case that covers the desired testing target, then the expectation (i.e., the average number of required test cases to sample) of random testing is $\mu = 1/\theta$ and its variance is $\delta^2 = (1-\theta)/\theta^2$ [36].

Figure 2 plots the mass function of a geometric distribution with $\theta = 0.01$ and a normal distribution with same μ and δ^2. In this context, the mass function represents the probability that, for a given number of sampled test cases l, the target is covered after sampling exactly l test cases. For random testing, the most likely outcome is $l = 1$, whereas for a normal distribution it is $l = \mu$. As it is easily visible from Figure 2, the geometric distribution has a very strong departure from normality! Comparisons of novel techniques versus random testing (as this is common practice when search algorithms are evaluated [57]) using *t*-tests can be questionable if the number of repeated experiments is "low". Furthermore, the probability distributions for

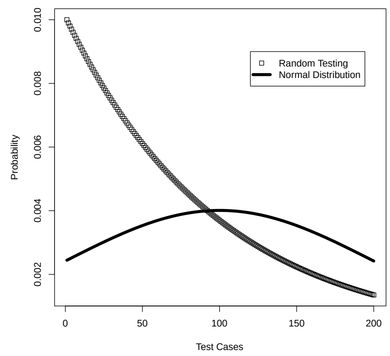

Figure 2: Mass and density functions of random testing and normal distribution given same mean $\mu = 1/\theta$ and variance $\sigma^2 = (1-\theta)/\theta^2$, where $\theta = 0.01$.

performance M (recall Section 4) for search algorithms may also strongly depart from normality. A common example is when the search landscape of the addressed problem has trap-like regions [91].

Violations of the assumptions of a statistical test such as t-test can be tolerated as long as they are not too "large" (where "large" can be somehow quantified with the kurtosis value [49]). Empirical evidence suggests that to be the case for natural and social sciences, and therefore probably so for empirical studies in software engineering involving human subjects. On the other end, there is no evidence at all in the literature that confirms it should be the case for randomized algorithms, used for example in the context of software testing. The arguments presented in this section actually cast doubts on such possibility. As long as no evidence is provided in the randomized algorithm literature to disprove the above concerns, in software testing or other fields of applications, one should not blindly follow guidelines provided for experiments with human subjects in software engineering or other experimental fields.

5.2 Central Limit Theorem

The Central Limit theorem states that the *sum* of n random variables converges to a normal distribution [36] as n increases. For example, consider the result of throwing a die. There are only six possible outcomes, each one with probability $1/6$ (assuming a fair die). If one considers the *sum* of two dice (i.e., $n = 2$), there would be 11 possible outcomes, from value 2 to 12. Figure 3 shows that with $n = 2$, in the case of dice, a distribution that resembles the normal one is already obtained, even though with $n = 1$ it is very far from normality. In the research context of this paper, these random variables are the results of the n runs of the analyzed algorithm. This theorem makes four assumptions: the n variables should be independent, coming from the same distribution and their mean μ and variance δ^2 should exist (i.e., they should be different from infinity). When using randomized algorithms, having n independent runs coming from the same distribution (e.g., the same algorithm) is usually trivial to achieve (one just needs to use different seeds for the pseudo-random generators). But the existence of the mean and variance requires more scrutiny. As shown before, those values μ and δ^2 exist for random testing. A well known "paradox" in statistics in which mean and variance do not exist is the Petersburg Game [36]. Similarly, the existence of mean and variance in search algorithms is not always guaranteed, as discussed next.

To put this discussion on a more solid ground, the Petersburg Game is here briefly described. Assume a player tosses an unbiased coin until a head is obtained. The player first gives an amount of money to the opponent which needs to be negotiated, and then she receives from the opponent an amount of money (Kroner) equal to $k = 2^t$, where t is the number of times the coin was tossed. For example, if the player obtains two tails and then a head, then she would receive from the opponent $k = 2^3 = 8$ Kroner. *On average*, how many Kroner k will she receive from the opponent in a single match? The probability of having $k = 2^x$ is equivalent to get first $x-1$ tails and then one head, so $p(2^x) = 2^{-(x-1)} \times 2^{-1} = 2^{-x}$. Therefore, the average reward is $\mu = E[k] = \sum_k kp(k) = \sum_t 2^t p(2^t) = \sum_t 2^t \times 2^{-t} = \sum_t 1 = \infty$. Unless the player gives an *infinite* amount of money to the opponent before starting tossing the coin, then the game would not be fair *on average* for the

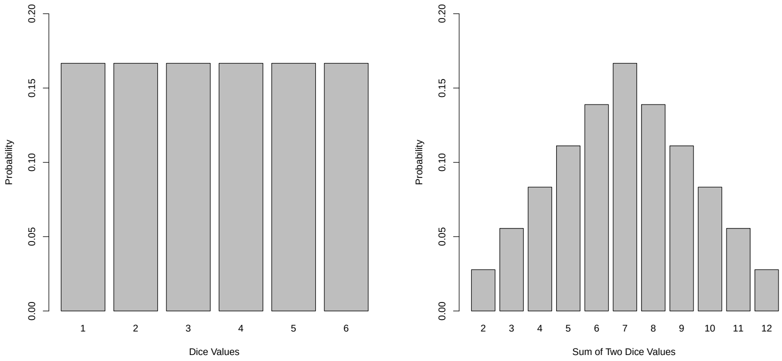

Figure 3: Density functions of the outputs of one dice and the sum of two dice.

opponent! This a classical example of a random variable where it is not intuitive to see that it has no finite mean value. For example, obtaining $t > 10$ is very unlikely, and if one tries to repeat the game n times, the average value for k would be quite low and would be a very wrong estimate of the actual, theoretical average (infinity).

Putting the issue illustrated by the Petersburg Game principle in the research context of this paper, if the performance of a randomized algorithm is bounded within a predefined range, then the mean and variance always exist. For example, if an algorithm is run for a predefined amount of time to achieve structural test coverage, and if there are z structural targets, then the performance of the algorithm would be measured with a value between 0 and z. Therefore, one would have $\mu \leq z$ and $\delta^2 \leq z^2$, thus making the use of a *t*-test valid.

The problems arise if no bound is given on how the performance is measured. A randomized algorithm could be run until it finds an optimal solution to the addressed problem. For example, random testing could be run until the first failure is triggered (assuming an automated oracle is provided). In this case, the performance of the algorithm would be measured in the number of test cases that are sampled before triggering the failure and there would be no upper limit for a run. If a researcher runs a search algorithm on the same problem n times, and he has n variables X_i representing the number of test cases sampled in each run before triggering the first failure, the mean would be estimated as $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i$, and one would hence conclude that the mean exists. As the Petersburg Game shows, this can be wrong, because $\hat{\mu}$ is only an *estimation* of μ, which might not exist.

For most search algorithms convergence in finite time is proven under some conditions (e.g., [100]), and hence mean and variance exist. But in software engineering, when new problems are addressed, standard search algorithms with standard search operators may not be usable. For example, when testing for object-oriented software using search algorithms (e.g., [114]), complex non-standard search operators are required. Without formal proofs (e.g., as done by Fraser and Arcuri [40]), it is not safe to speak about the existence of the mean in those cases.

However, the non-existence of the mean is usually not a problem from a practical standpoint. In practice, there usually are upper limits to the amount of computational resources a randomized algorithm can use. For example, a search algorithm can be prematurely stopped when reaching a time limit. Random testing could be stopped after $100{,}000$ sampled test cases if it has found no failure so far. But, in these cases, one is actually dealing with *censored* data [71] (in particular, right-censorship) and this requires proper care in terms of statistical testing and the interpretation of results, as it will be discussed in Section 6.

5.3 Differences in the Compared Properties

Even under proper conditions for using a parametric test, one aspect that is often ignored is that the *t*-test and U-test analyze two different properties. Consider a random testing example in which one counts the number of test cases run before triggering a failure. Considering a failure rate θ, the mean value of test cases sampled by random testing is hence $\mu = 1/\theta$. Assume that a novel testing technique $\mathcal{A}$ yields a normal distribution of the required number of test cases to trigger a failure. If one further considers the same variance as random testing and a mean that is 85% of that of random testing, which one is better? Random testing with mean μ or $\mathcal{A}$ with mean 0.85μ? Assuming a large number of runs (e.g., n is equal to one million), a *t*-test would state that $\mathcal{A}$ is better, whereas a Mann-Whitney U-test would state exactly the opposite. How come? This is not an error as the two tests are measuring different things: The *t*-test measures the difference in mean values whereas the Mann-Whitney U-test deals with their stochastic ranking, i.e., whether observations in one data sample are more likely to be larger than observations in the other sample. Notice that this latter concept is technically different from detecting difference in *median* values (which can be stated only if the two distributions have same shape). In a normal distribution, the median value is equal to the mean, whereas in a geometric distribution the median is roughly 70% of the mean [36]. On one hand, half of the data points for random testing would be lower than 0.7μ. On the other hand, with $\mathcal{A}$ half of the data points would be above 0.85μ, and a significant proportion between 0.7μ and 0.85μ. This explains the apparent contradiction in results: though the average is higher for random testing, its median is lower than that of $\mathcal{A}$.

From a practical point of view, which statistical test should be used? Based on the discussions in this section, and in line with Leech and Onwuegbuzie [76], it is recommendable to use Mann-Whitney U-test (to assess difference in stochastic order) rather than the *t*-test and Welch test (to assess difference in mean values). However, the full motivation will become clearer once censored data, effect size, and the choice of n will be discussed in the next sections.

5.4 Rank Transformation

There is an important aspect that needs to considered: data can be "transformed" before being given as input to a statistical test. As discussed by Ruxton [101], a Welch test can be used instead of a U-test if the raw values in the data are replaced by their rank. For example, consider the data set $\{24, 2, 274\}$ discussed in the introduction regarding random testing. Those values could be substituted with their ranks $\{2, 1, 3\}$ before being given as input to a statistical test. What would be the motivation of doing so? The U-test might be negatively affected if the two compared distributions have "significantly" different variance, and in such case a Welch test on ranked data might be better (in the sense that it would have lower probability of Type I and II errors). However, the Welch test would still be negatively affected by violations of the normality assumption (ranked data might not be normal). Ruxton [101] reports on some cases in which a Welch test on ranked data is better than a U-test, but the results of those *empirical* analyses might not generalize to the context of randomized algorithms applied to software engineering problems.

For simplicity and because it has widespread applications, the authors of this paper recommend to use a U-test rather than a Welch test on ranked data. There might be cases in which this latter test could be preferable, but it might be difficult, for a non-expert in statistics, to clearly identify those cases. Nevertheless, it is important to clarify that a Welch test on ranked data does not assess any more whether there is a statistical difference among the mean values of the two compared distributions. Rather, it assesses differences in mean values of the ranks and therefore determine whether there is any difference in stochastic ordering between the two distributions. For example, assume the two data sets $X = \{1, 2, 3, 4, 5, 6, 49\}$ and $Y = \{7, 8, 9, 10, 11, 12, 13\}$. If it were not for the "outlier" 49 in X, then all the values in Y would be greater than the values in X. Both data sets have a mean value equal to 10. A Welch test on raw values would result in a *p*-value equal to 1, which is not surprising considering that the two data sets have the same mean. However, if one does a rank transformation, then the outlier 49 would be replaced by the value 14 (all the other values in X and Y remain the same). In this case, the resulting *p*-value of the Welch test would be 0.02, which suggests a strong difference in the stochastic ordering (i.e., ranks) between the two distributions.

5.5 Test for Randomized vs Deterministic Algorithm

In the discussions above, it was assumed that both algorithms $\mathcal{A}$ and $\mathcal{B}$ are randomized. If one of them is deterministic (e.g., $\mathcal{B}$), it is still important to use statistical testing. Consistent with the above recommendation, the non-parametric *One-Sample Wilcoxon* test should be used. Given $m_{\mathcal{B}}$ the performance measure of the deterministic algorithm, a one-sample Wilcoxon test would verify whether the performance of $\mathcal{A}$ is symmetric about $m_{\mathcal{B}}$, i.e., whether by using $\mathcal{A}$ one is as likely to obtain a value lower than $m_{\mathcal{B}}$ as otherwise.

6 Censored Data

Assume that the result of an experiment is dichotomous: either one finds a solution to solve the software engineering problem at hand (*success*), or he does not (*failure*). For example, in software testing, if the goal is to cover a particular target (e.g., a specific branch), one can run a randomized algorithm with a time limit L, chosen based on available computing resources. The algorithm will be stopped as soon as a solution is found, otherwise the search stops after time L. Another example is bug fixing [117] where one finds a patch within time L, or does not.

The above types of experiments are dealing with *right-censored* data, and their properties are equivalent to survival/failure time analysis [71, 41]. Let X be the random variable representing the time a randomized algorithm takes to solve a software engineering problem, and consider n experiments in which a researcher collects X_i values. This is a case of right-censorship since, assuming a time limit L, one will not have observations X_i for the cases $X > L$. Although there are several ways to deal with this problem [71], in this paper the discussions are limited to simple solutions.

One interesting special case is when one cannot say for sure whether the chosen target has been achieved, e.g., generation of test cases that achieve code branch coverage. Putting aside trivial cases, there are usually infeasible targets (e.g., unreachable code) and their number is unknown. As a result, such experiments are not dichotomous because one cannot know whether all feasible targets have been covered. Even when using a time limit L, these cases would still not be considered as involving censored data. However, if in the experiments the comparisons are made reusing artifacts from published studies in the literature, and if one wants to know whether or not, within a given time, he can obtain better coverage than these reported studies, then such experiments can be considered dichotomous despite infeasible targets.

Consider the case in which one needs to compare two randomized algorithms $\mathcal{A}$ and $\mathcal{B}$ on a software engineering problem with dichotomous outcome. Let X be the random variable representing the time $\mathcal{A}$ takes to find a valid solution, and let Y be the same type of variable for $\mathcal{B}$. Assume that a researcher runs $\mathcal{A}$ and $\mathcal{B}$ n times, collecting observations X_i and Y_i, respectively. Using a time limit L, to evaluate which of the two algorithms is better, one can consider their *success rate* $\gamma = k/n$, i.e., the proportion of number of times k, out of the n runs, for which a valid solution is found. To evaluate whether there is statistical difference between the success rates of $\mathcal{A}$ and $\mathcal{B}$, a test for differences in proportions is then appropriate, such as the Fisher exact test [71].

The Fisher exact test is a parametric test, which assumes that the analyzed data follows a binomial distribution. In contrast to other parametric tests (e.g., the *t*-test), its assumptions are always valid: if the experiments are independent, then the success rate of a series of randomized experiments would always follow a binomial distribution, where γ represents the estimated probability of success. Furthermore, for values of n until roughly 100, the test is "exact". This means that the resulting *p*-values are precise, and not estimates based on how close the data are from satisfying the conditions of a test (e.g., normality and equal variance in a *t*-test). However, for larger values of n, the computational cost of the test would start to be too prohibitive, and approximations are then used to calculate the *p*-values (this is often done automatically in many statistical tools).

Assume that out of $n = 100$ runs the success rate of $\mathcal{A}$ is $\gamma_{\mathcal{A}} = 1\%$, whereas for $\mathcal{B}$ it is $\gamma_{\mathcal{B}} = 5\%$. A Fisher exact test has a resulting *p*-value equal to 0.21, which might be considered high, i.e., there is a 21% probability that the success rates of the two algorithms are actually equal. In such cases, one can run more experiments (i.e., increase n) to obtain higher statistical power (i.e., decrease the *p*-value). Alternatively, if there is no statistically or practically significant difference between the success rates of $\mathcal{A}$ and $\mathcal{B}$, a practical question is then to determine which technique uses *less* time. This is particularly relevant if the success rates of both techniques are high. There can be different ways to analyze such cases, such as considering artificial censorships at different times before L. For example, one can consider censorship at $L/2$, i.e., the success rate with half the time, and determine which technique still fares better and whether its success rate is acceptable. Note that such analysis does not require to run any further experiments as success rates can be computed at $L/2$ from existing runs. Another alternative to compare execution times is to apply a Mann-Whitney U-test, recommended above, using only the times of successful runs, which have X_i and Y_i values lower or equal to L.

A more complex situation is when one algorithm shows a significantly higher success rate, but takes more time to produce valid solutions than the other. This is a typical situation, that is not so uncommon, where a choice needs to be made. For example, on one hand, a *local search* [81] might be very fast in generating appropriate testing data if it starts from the right area of the search landscape. But, at the same time, it could yield a low success rate if most of the search landscape has gradient toward local optima, and if the number of such optima is low. (Notice that this is just an example: it is not in the scope of the paper to give lengthy explanations of why that would be a problem for local search; see the work of Arcuri [8] for further details on this topic.) On the other hand, a population-based search algorithm, such as Genetic Algorithms, could avoid the problem of local optima, which in turn would result in higher success rate than a local search. However, because an entire population is evolved at the same time, depending on the selection pressure of the algorithm (e.g., the value of the tournament size in tournament selection) and the population size, a Genetic Algorithm might take much longer than a local search to converge towards a solution in its successful runs.

7 Effect Size

When comparing a randomized algorithm $\mathcal{A}$ against another $\mathcal{B}$, given a large enough number of runs n, it is most of the time possible to obtain statistically significant results with a *t*-test or U-test. Indeed, two different algorithms are extremely unlikely to have exactly the same probability distribution. In other words, with a large enough n one can obtain statistically significant differences even if they are so small as to be of no practical value.

Though it is important to assess whether an algorithm fares statistically better than another, it is in addition crucial to assess the magnitude of the improvement. To analyze such a property, *effect size* measures are needed [55, 63, 89]. Effect sizes can be divided in two groups: standardized and unstandardized. Unstandardized effect sizes are dependent on the unit of measurement used in the experiments. Consider the difference in means between two algorithms $\Delta = \mu^{\mathcal{A}} - \mu^{\mathcal{B}}$. This value Δ has a measurement unit, that of $\mathcal{A}$ and $\mathcal{B}$. For example, in software testing, μ can be the expected number of test executions to find the first failure. On one testing artifact it could be that $\Delta_1 = \mu^{\mathcal{A}} - \mu^{\mathcal{B}} = 100 - 1 = 99$, whereas on another testing artifact it can be $\Delta_2 = \mu^{\mathcal{A}} - \mu^{\mathcal{B}} = 100{,}000 - 200{,}000 = -100{,}000$. Deciding based on Δ_1 and Δ_2 which algorithm is better is difficult to determine since the two scales of measurement are different. Δ_1 is very low compared to Δ_2, but in that case $\mathcal{A}$ is 100 times worse than $\mathcal{B}$, whereas it is only twice as fast in the case Δ_2.

Empirical analyses of randomized algorithms, if they are to be reliable and generalizable, require the use of large numbers of artifacts (e.g., programs). The complexity of these artifacts is likely to widely vary, such as the number of test cases required to fulfill a coverage criterion on various programs. The use of standardized effect sizes, that are independent from the evaluation criteria measurement unit, is therefore necessary to be able to compare results across artifacts and experiments. In their systematic review of empirical analyses in software engineering involving controlled experiments with human subjects, Kampenes *et al.* [63] found that standardized effect sizes were reported in only 29% of the cases. In the systematic review performed in this paper, only one paper [96] was found, which uses the Vargha and Delaney's $\hat{A}_{12}$ statistics (described later in this section).

In this section, the most known standardized effect size measure is described first followed by an explanation of why it should *not* be used when analyzing randomized algorithms applied in software engineering. Then, two other standardized effect sizes are described, and instructions are given on how to apply them in practice.

The most known effect size is the so called d family which, in the general form, is $d = (\mu^{\mathcal{A}} - \mu^{\mathcal{B}})/\sigma$. In other words, the difference in mean is scaled over the standard deviation (several corrections exists to this formula, but for more details please see the book of Grissom and Kim [55]). Though one obtains a measure that has no measurement unit, the problem is that it assumes normality of the data, and strong departures can make it meaningless [55]. For example, in a normal distribution, roughly 64% of the points lie within $\mu \pm \sigma$ [36], i.e., they are at most σ away from the mean μ. But for distributions with high skewness (as in the geometric distribution and as it is often the case for search algorithms), the results of scaling the mean difference by the standard deviation "would not be valid", because "standard deviations can be very sensitive to a distribution's shape" [55]. In this case, a non-parametric effect size should be preferred. Existing guidelines [63, 89] only briefly discuss the use of non-parametric effect sizes.

The Vargha and Delaney's $\hat{A}_{12}$ statistic is a non-parametric effect size measure [116, 55]. Its use has been advocated by Leech and Onwuegbuzie [76], and one example of its use in software engineering in which randomized algorithms are involved can be found in the work of Poulding and Clark [96]. In the research context of this paper, given a performance measure M, $\hat{A}_{12}$ measures the probability that running algorithm $\mathcal{A}$

yields higher M values than running another algorithm $\mathcal{B}$. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. This effect size is easier to interpret compared to the d family. For example, $\hat{A}_{12} = 0.7$ entails one would obtain better results 70% of the time with $\mathcal{A}$. Though this type of non-parametric effect size is not common in statistical tools, it can be very easily computed [76, 55]. The following formula is reported in the work of Vargha and Delaney [116]:

$$\hat{A}_{12} = (R_1/m - (m+1)/2)/n \tag{1}$$

where R_1 is the rank sum of the first data group under comparison. For example, assume the data $X = \{42, 11, 7\}$ and $Y = \{1, 20, 5\}$. The data set X would have ranks $\{6, 4, 3\}$, whose sum is 13, whereas Y would have ranks $\{1, 5, 2\}$. The rank sum is a basic component in the Mann-Whitney U-test, and most statistical tools provide it. In Equation 1, m is the number of observations in the first data sample, whereas n is the number of observations in the second data sample. In most experiments, one would run two randomized algorithms the same number of times: $m = n$.

When dealing with dichotomous results (as discussed in Section 6), several types of effect size measures [55] can be considered. The *odds ratio* is the most used and "is a measure of how many times greater the odds are that a member of a certain population will fall into a certain category than the odds are that a member of another population will fall into that category" [55]. Given a the number of times algorithm $\mathcal{A}$ finds an optimal solution, and b for algorithm $\mathcal{B}$, the odds ratio is calculated as

$$\psi = \frac{a+\rho}{n+\rho-a} / \frac{b+\rho}{n+\rho-b} \, , \tag{2}$$

where ρ is any arbitrary positive constant (e.g., $\rho = 0.5$) used to avoid problems with zero occurrences [55]. There is no difference between the two algorithms when $\psi = 1$. The cases in which $\psi > 1$ imply that algorithm $\mathcal{A}$ has higher chances of success.

Both $\hat{A}_{12}$ and ψ are standardized effect size measures. But because their calculation is based on a finite number of observations (e.g., n for each algorithm, so $2n$ when two algorithms are compared), they are only estimates of the real $\hat{A}^*_{12}$ and ψ^*. If n is low, these estimations might be very inaccurate. One way to deal with this problem is to calculate *confidence intervals* (CI) for them [55]. A $(1-\alpha)$ CI is a set of values for which there is $(1-\alpha)$ probability that the value of the effect size lies in that range. For example, if one has $\hat{A}_{12} = 0.54$ and a $(1-\alpha)$ CI with range $[0.49{,}59]$, then with probability $(1-\alpha)$ the real value $\hat{A}^*_{12}$ lies in $[0.49{,}59]$ (where $\hat{A}_{12} = 0.54$ is its most likely estimation). Such effect size confidence intervals can facilitate decision making as they enable the comparison of the costs of alternative algorithms while accounting for uncertainty in their estimates. To see how confidence intervals are calculated for $\hat{A}_{12}$, please see the book of Grissom and Kim [55] or the work of Vargha and Delaney [116].

Furthermore, general techniques such as *bootstrapping* [24] can be employed to create confidence intervals for $\hat{A}_{12}$ or any other statistics of interest (e.g., mean and median). At a high level, bootstrapping works as follows. Assume n experiments with results x_i. The arithmetic average would be calculated as $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$. Because n is finite, μ is only an estimate of the real average (e.g., recall the Petersburg Game discussed in Section 5.2). By defining X as the set of n results x_i, bootstrapping works by resampling n values with replacement from X and by calculating the statistics of interest (e.g., the mean) on this new set (e.g., μ_j). This process is repeated k times (e.g., $k =$ 1,000), which provides k values for the statistics of interest (e.g., μ_1,μ_2, $\ldots$, μ_k). Then, several different techniques can be used to create a confidence interval at level α from these k estimates. For more details on the properties of bootstrapping, the interested reader is referred to Chernick's book [24].

Notice that a confidence interval can replace a test of statistical difference (e.g., *t*-test and U-test). If the null hypothesis H_0 lies within the confidence interval, then there is insufficient evidence to claim there is a statistically significant difference. In the previous example, because 0.5 is inside the $(1-\alpha)$ CI $[0.49{,}59]$, then there is no statistical difference at the selected significance level α. For a dichotomous result, H_0 would be $\psi = 1$.

8 Number of Runs

How many runs does a researcher need when analyzing and comparing randomized algorithms? A general answer is: As many as necessary to show with high confidence that the obtained results are statistically significant and to obtain a small enough confidence interval for effect size estimates. In many fields of science (e.g., medicine and behavioral science), a common rule of thumb is to use at least $n = 30$ observations. In the many fields where experiments are very expensive and time consuming, it is in general not feasible to work with high values for n. Several new statistical tests have been proposed and discussed to cope with the problem of lack of power and violation of assumptions (e.g., normality of data) when smaller numbers of observations are available [119].

Empirical studies of randomized algorithms usually do not involve human subjects and the number of *runs* (i.e., n) is only limited by computational resources. When there is access to clusters of computers as this is the case for many research institutes and universities, and when there is no need for expensive, specialized hardware (e.g., hardware-in-the-loop testing), then large numbers of runs can be carried out to properly analyze the behavior of randomized algorithms. Many software engineering problems are furthermore not highly computationally expensive, as for example code coverage at the unit testing level, and can therefore involve very large numbers of executions. There are however exceptions, such as the system testing of embedded systems (e.g., [12]) where each test case can be very expensive to run.

Whenever possible, in most cases, it is therefore recommended to use a very high number of runs. For most problems in software engineering, thousands of randomized algorithm runs should be feasible and would solve most of the problems related to the power and accuracy of statistical tests. For example, as illustrated in references [83, 32] in Table 2, even with 100 runs, the U-test might not be powerful enough to confirm a statistical difference at a 0.05 significance level, even when the data seems to suggest such a difference.

Most discussions in the literature about statistical tests focus on situations with small numbers of observations (e.g., [101]). However, with thousands of runs, one would detect statistically significant differences on practically any experiment (Section 4). It is hence essential to complement such analyses with a study of the effect size as discussed in Section 7. Even when having large numbers of runs is not necessary, for a set α level (e.g., 0.05), to obtain differences that are large enough to show *p*-values less than α, additional runs would help tighten the confidence intervals for effect size estimates and would be of practical value to support decision making.

In Section 4, it was suggested to use U-test instead of *t*-test. For very large samples, such as $n = 1{,}000$, there would be no practical difference between them regarding power and accuracy. However, the choice of a non-parametric test would be driven by its corresponding effect size measure. In Section 7 it was argued that effect size measures based on the mean (i.e., the d family) were not appropriate for randomized algorithms in software engineering due to violations in distribution assumptions. It would then be inconsistent to investigate the statistical difference of mean values with a *t*-test if one cannot use a reliable measure for its effect size. In other words, it is advisable to use size measures that are consistent with the differences being tested by the selected statistical test.

9 Multiple Tests

In most situations, researchers need to compare several alternative algorithms. Furthermore, if one is comparing different algorithm settings (e.g., population size in a Genetic Algorithm), then each setting technically defines a different algorithm [11]. This often leads to a large number of statistical comparisons. It is possible to use statistical tests that deal with multiple techniques (treatments, experiments) at the same time (e.g., Factorial ANOVA), and effect sizes have been defined for those cases [55]. There are several types of statistical tests addressing multiple comparisons, and the choice depends on which research question one is addressing. This paper only deals with the two most common research questions, since several books are dedicated to this topic, and an exhaustive analysis would not be possible:

- Does the choice of a particular parameter affect the performance of a randomized algorithm?
- Among a set of randomized algorithms, which one is the best in solving the addressed problem?

Given a parameter that can take several different values $j \in J$, assume a researcher has carried out a series of experiments for a set of parameter values $\{j_1, j_2, \ldots, j_k\} \subseteq J$. For example, in a Genetic Algorithm, one might want to study whether applying different cross-over rates has any effect on the effectiveness of the algorithm. One could consider the values $\{0, 0.25, 0.5\ 0.75, 1\}$, and have $n = 1{,}000$ independent experiments for each of these five rate values. If the goal is to evaluate whether the choice of this rate has any effect on the effectiveness of a Genetic Algorithm, then an *omnibus* test such as ANOVA can be employed. The null hypothesis is that the choice of the parameter value has no effect on the mean effectiveness of the algorithm. However, ANOVA suffers of the same problems as the t-test, i.e., assumption about normality of the data and equal variance. A non-parametric equivalent is the so called Kruskal-Wallis test [73].

Assume that the result of a Kruskal-Wallis test suggests that the choice of that crossover rate has a statistically significant effect (i.e., the resulting *p*-value is low, so one can reject the null hypothesis). A relevant question might then be which crossover rate should be used (i.e., which one gives the best performance?). An omnibus test is not able to answer such a research question. This situation is exactly equivalent to the case of identifying the best algorithm among $K = 5$ algorithms/variants. In this case, one would like to individually compare the performance of each algorithm against all other alternatives. Given a set of algorithms, a researcher would not be interested in simply determining whether all of them have the same mean values. Rather, given K algorithms, one wants to perform $Z = K(K-1)/2$ pairwise tests and measure effect size in each case.

However, using several statistical tests inflates the probability of Type I error. If one has only one comparison, the probability of Type I error is equal to the obtained *p*-value. On the other hand, if one has many comparisons, even when all the *p*-values are low, there is usually a high probability that at least in one of the comparisons the null hypothesis is true as all these probabilities somehow add up. In other words, if in all the comparisons the *p*-values are lower than α, then a researcher would normally reject all the null hypotheses. But the probability that at least one null hypothesis is true could be as high as $1 - (1-\alpha)^Z$ for Z comparisons, which converges to 1 as Z increases.

One way to address this problem is to use the so called *Bonferroni adjustment* [94, 88]. Instead of applying each test assuming a significance level α, a researcher would use an adjusted level α/Z. For example, if the probability of Type I error is selected to be 0.05 and two comparisons are performed, two statistical tests are run with $\alpha = 0.025$ to check whether both differences are significant (i.e., if both *p*-values are lower than 0.025). However, the Bonferroni adjustment has been repeatedly criticized in the literature [94, 88], and the authors of this paper largely agree with those critiques. For example, assume that for both those tests the researcher obtains *p*-values equal to 0.04. If a Bonferroni adjustment is used, then both tests will not be statistically significant with $\alpha = 0.05$. It would then be tempting to publish the results of only one of them and claiming statistical significance because $0.04 < 0.05$. Such a practice can therefore hinder scientific progress by reducing the number of published results [94, 88]. This would be particularly true when many randomized algorithms can be compared to address the same software engineering problem: it would be very tempting to leave out the results of some of the poorly performing algorithms. Notice that there are other adjustment techniques that are equivalent to Bonferroni but that are less conservative [44]. However, the statistical significance of a single comparison would still depend on the number of performed and reported comparisons. Though in general it is not recommend to use the Bonferroni adjustment, it is important to always report the obtained *p*-values, not just whether a difference is significant or not at an arbitrarily chosen α level. If for some reasons the readers want to evaluate the results using a Bonferroni adjustment or any of its (less conservative) variants, then it is possible to do so. For a full list of other problems related to the Bonferroni adjustment, the reader is referred to the work of Perneger [94] and Nakagawa [88].

Instead of pairwise tests using Bonferroni-like corrections, another (less popular) approach is to use the so called *post-hoc* methods, such as the Tukey's range test. This test is applied on each of the Z pairs, and it is very similar to a *t*-test. Similar to the Bonferroni method, it employs a *p*-value correction to handle possible inflation of probability of Type I error.

At any rate, alpha level adjustments can be very important when assessing the validity of behavioral or natural phenomena with high confidence. For example, the leading international journal *Nature* has the following *requirement*[3] for published research papers regarding multiple tests:

> Multiple comparisons: When making multiple statistical comparisons on a single data set, authors should

[3] http://www.nature.com/nature/authors/gta/index.html#a5.6, accessed November 2011.

explain how they adjusted the alpha level to avoid an inflated Type I error rate, or they should select statistical tests appropriate for multiple groups (such as ANOVA rather than a series of t-tests).

However, in Section 4 it was stated that in software engineering in general, and for randomized algorithms in particular, one mostly deals with decision-making problems. For example, if one must test software, then one must choose one alternative among K different techniques. In this case, even if the *p*-values are higher than α, the software needs to be tested anyhow and a choice must be made. In this context, Bonferroni-like adjustments make even less sense. Just keep using the current technique because there is no statistically significant difference at a prefixed arbitrary α level is not optimal as it ignores available information.

Assume that a researcher has analyzed the performance of K algorithms using pairwise tests and effect sizes. How to visualize the results of such analyses to grasp how their performance relate? There can be different ways (e.g., see the recent work of Carrano *et al.* [23]), and the description of a simple but practical technique is here provided, which was used for example by Fraser and Arcuri [38].

In their work [38], the effects of six parameters of a search algorithm were investigated in the context of automated unit testing of object-oriented software. Five parameters are binary (**Bo**, **Xo**, **Ra**, **Pa** and **Be**) and one ternary (**W**), for a total of $2^5 \times 3 = 96$ configurations. Each configuration was compared against all the other 95 (i.e., a total of 96×95 comparisons, which can be divided by two due to the symmetric property of the comparisons). Pairwise comparisons were made using a U-test, where the α level was arbitrarily set to 0.05. Initially, a score of zero is assigned to each configuration. For each comparison in which a configuration is statistically better, its score is increased by one, whereas it is reduced by one in case it is statistically worse. Therefore, in the end each configuration obtains a score between -95 and 95, where the higher the score, the better the configuration. After this first phase, these scores are ranked such that the highest score has the best rank, where better ranks have lower values. In case of ties, the ranks are averaged. For example, if one has five configurations with scores $\{10,\, 0,\, 0,\, 20,\, -30\}$, then their ranks will be $\{2,\, 3.5,\, 3.5,\, 1,\, 5\}$. In the work of Fraser and Arcuri [38], this procedure was repeated for each artifact in the case study (i.e., for all the 100 branches used in that empirical study), and the average of these ranks over all artifacts were calculated for each configuration, for a total of $100 \times 96 \times 95/2 = 456{,}000$ statistical comparisons. After collecting all of these data, a table (reported in Table 4) was made in which the configurations were ordered based on their average rank from top (best) to bottom (worst). From this table, not only it is clear which are the best configurations, but it also possible to visualize some trends in the data (e.g., configurations with **Ra** are always better and **Xo** does not seem particularly useful). However, the above ranking mechanism has limitations, as it ignores the effect sizes and the actual *p*-values (e.g., a 0.051 value would be treated in the same way as a 1).

10 Experimenting With Several Artifacts

10.1 Choice of the Artifacts

When assessing randomized algorithms, the choice of artifacts to which these algorithms are applied (e.g., source code or executable programs) is of paramount importance as it usually has a strong bearing on the evaluation results. When analyzing empirical analyses in the software engineering literature evaluating randomized algorithms, many of the studies are carried out on artificial and small artifacts. Empirical analyses on real industrial systems are rare, thus raising questions about the credibility of results and the usefulness of the proposed algorithms. However, achieving realism by using representative industrial systems is particularly challenging. One usually cannot precisely characterize the population of artifacts he is targeting in his studies. Even if a researcher could, he usually does not have access to large collections of industrial artifacts that are readily available to be sampled. And even if that were the case, studies are necessarily limited in terms of resources and time, and the number of artifacts studied is typically much more restricted than one would like. As a result, studies about randomized algorithms in software engineering typically present threats to external validity, making it difficult to generalize the results to other systems than the ones under study. In this paper, because the focus is on how to apply statistical tests, the details of how one should choose artifacts from a general standpoint are not emphasized. The following discussions in the paper rather concentrate on how this choice affects the statistical tests procedures and the number of runs required.

The first question one faces is whether the selected artifacts are *representative* of the type of problem that is being addressed. For example, assume one wants to evaluate a new tool for automatically generating unit

Table 4: Results of empirical analysis performed in the work of Fraser and Arcuri [38]. The table shows the performance of the the 96 configurations, ordered from top (best performance) to bottom (worst performance). Symbols are used to indicate whether a particular boolean parameter is activated.

Bo	Xo	Ra	Pa	Be		W		Av. Rank	Av. Success Rate
					20	50	80		
△		⊕	▽	⊞		$\mathcal{W}$		31.475	0.464
△		⊕	▽			$\mathcal{W}$		31.840	0.456
△		⊕		⊞		$\mathcal{W}$		32.595	0.482
		⊕	▽	⊞		$\mathcal{W}$		32.670	0.456
		⊕	▽			$\mathcal{W}$		34.725	0.447
△		⊕				$\mathcal{W}$		35.415	0.448
		⊕		⊞		$\mathcal{W}$		36.070	0.442
△		⊕		⊞	$\mathcal{W}$			37.335	0.423
△	⊠	⊕	▽	⊞		$\mathcal{W}$		37.430	0.430
△		⊕		⊞			$\mathcal{W}$	37.605	0.459
	⊠	⊕		⊞	$\mathcal{W}$			37.615	0.418
△	⊠	⊕		⊞		$\mathcal{W}$		38.080	0.422
	⊠	⊕	▽	⊞		$\mathcal{W}$		39.325	0.419
	⊠	⊕		⊞		$\mathcal{W}$		39.455	0.423
	⊠	⊕	▽			$\mathcal{W}$		39.580	0.413
△		⊕			$\mathcal{W}$			39.790	0.431
		⊕		⊞	$\mathcal{W}$			39.815	0.431
	⊠	⊕			$\mathcal{W}$			40.050	0.414
△		⊕	▽		$\mathcal{W}$			40.140	0.420
△	⊠	⊕	▽			$\mathcal{W}$		40.330	0.425
△		⊕	▽	⊞	$\mathcal{W}$			40.670	0.413
△		⊕	▽	⊞			$\mathcal{W}$	40.700	0.432
△	⊠	⊕		⊞	$\mathcal{W}$			40.835	0.405
		⊕		⊞			$\mathcal{W}$	40.940	0.438
△		⊕	▽				$\mathcal{W}$	41.200	0.455
△	⊠	⊕			$\mathcal{W}$			41.350	0.410
		⊕	▽	⊞			$\mathcal{W}$	41.695	0.423
		⊕	▽	⊞	$\mathcal{W}$			41.890	0.405
		⊕	▽		$\mathcal{W}$			41.925	0.413
	⊠	⊕	▽		$\mathcal{W}$			42.150	0.399
	⊠	⊕	▽	⊞			$\mathcal{W}$	42.195	0.401
	⊠	⊕	▽	⊞	$\mathcal{W}$			42.470	0.388
△	⊠	⊕	▽		$\mathcal{W}$			42.500	0.395
	⊠	⊕		⊞			$\mathcal{W}$	42.800	0.422
		⊕			$\mathcal{W}$			43.075	0.407
	⊠	⊕				$\mathcal{W}$		43.095	0.421
△	⊠	⊕				$\mathcal{W}$		43.255	0.420
△	⊠	⊕	▽	⊞	$\mathcal{W}$			43.635	0.377
		⊕				$\mathcal{W}$		45.160	0.398
	⊠	⊕	▽				$\mathcal{W}$	45.205	0.393
		⊕	▽				$\mathcal{W}$	45.285	0.412
△	⊠	⊕	▽				$\mathcal{W}$	45.450	0.392
△		⊕					$\mathcal{W}$	45.850	0.418
		⊕					$\mathcal{W}$	46.460	0.401
△	⊠	⊕					$\mathcal{W}$	46.625	0.388
△	⊠	⊕		⊞			$\mathcal{W}$	46.700	0.409
△	⊠	⊕	▽	⊞			$\mathcal{W}$	47.760	0.379
	⊠	⊕					$\mathcal{W}$	47.850	0.384
△			▽	⊞		$\mathcal{W}$		48.985	0.342
			▽			$\mathcal{W}$		49.585	0.329
			▽	⊞		$\mathcal{W}$		49.705	0.334
△			▽	⊞	$\mathcal{W}$			49.995	0.369
△	⊠		▽	⊞		$\mathcal{W}$		50.290	0.313
△			▽		$\mathcal{W}$			50.740	0.356
△	⊠		▽			$\mathcal{W}$		51.295	0.313
△			▽			$\mathcal{W}$		51.350	0.340
△				⊞		$\mathcal{W}$		51.570	0.327
△			▽	⊞			$\mathcal{W}$	52.215	0.326
△				⊞	$\mathcal{W}$			52.800	0.330
			▽	⊞			$\mathcal{W}$	53.260	0.330
	⊠		▽	⊞		$\mathcal{W}$		53.610	0.309
△			▽				$\mathcal{W}$	53.845	0.321
	⊠		▽	⊞	$\mathcal{W}$			54.040	0.310
	⊠		▽				$\mathcal{W}$	54.475	0.312
			▽	⊞	$\mathcal{W}$			54.835	0.296
			▽		$\mathcal{W}$			55.080	0.306
				⊞		$\mathcal{W}$		55.290	0.317
	⊠		▽			$\mathcal{W}$		55.390	0.313
	⊠		▽	⊞			$\mathcal{W}$	55.605	0.304
△					$\mathcal{W}$			55.635	0.305
			▽				$\mathcal{W}$	55.695	0.324
△	⊠		▽		$\mathcal{W}$			56.065	0.310
△						$\mathcal{W}$		56.160	0.309
	⊠			⊞		$\mathcal{W}$		56.200	0.304
△	⊠		▽	⊞			$\mathcal{W}$	56.255	0.301
	⊠		▽		$\mathcal{W}$			56.295	0.312
△	⊠		▽	⊞	$\mathcal{W}$			56.655	0.312
△	⊠		▽				$\mathcal{W}$	56.835	0.291
△	⊠					$\mathcal{W}$		57.095	0.279
△	⊠			⊞		$\mathcal{W}$		57.135	0.291
△				⊞			$\mathcal{W}$	57.180	0.319
				⊞			$\mathcal{W}$	57.390	0.306
							$\mathcal{W}$	58.955	0.285
△	⊠			⊞			$\mathcal{W}$	59.085	0.297
				⊞	$\mathcal{W}$			59.190	0.297
△	⊠			⊞	$\mathcal{W}$			59.270	0.285
	⊠					$\mathcal{W}$		59.595	0.279
△							$\mathcal{W}$	59.995	0.300
	⊠			⊞	$\mathcal{W}$			60.145	0.281
	⊠						$\mathcal{W}$	60.150	0.289
△	⊠				$\mathcal{W}$			60.675	0.278
	⊠			⊞			$\mathcal{W}$	60.705	0.289
△	⊠						$\mathcal{W}$	60.975	0.292
						$\mathcal{W}$		61.655	0.267
	⊠				$\mathcal{W}$			65.220	0.238
					$\mathcal{W}$			71.765	0.190

tests for object-oriented software (e.g., Pex [113], Randoop [93] or EvoSuite [40]). Which (types of) classes should be selected for experimenting? Following common practice in many empirical studies (e.g., [5, 98, 15]), is only using "container classes" acceptable? Arguably, it should depend on what is the target set of classes for the evaluation. If the proposed testing techniques are aimed *only* at container classes (e.g., [15]), then this would likely be acceptable. On the other hand, if the goal is to propose a *general* tool for generating unit tests, then using only container classes would lead to *serious* threats to external validity. But then the question is which classes should ideally be used? Again, one does not have well defined populations of classes that can be explicitly targeted and sampled. One possible simple heuristic is to try to maximize the diversity in terms of the type of classes, their size and complexity, and various other properties that are deemed relevant given the objective of the randomized algorithm, e.g., number of tasks accessing a lock when investigating deadlocks or data races [107].

As a practical alternative, one could use open source repositories such as SourceForge[4], and randomly select a subset of projects for experimenting among the $319{,}000$ that are currently hosted (as for example done by Fraser and Arcuri [39]). If one wants to evaluate the applicability of a general tool for unit testing, this would be better than using only container classes or arbitrarily choosing some programs in a non-systematic way (as it is often the case in the literature). However, even if one randomly samples projects from SourceForge, the empirical analyses would likely have some sort of bias. For example, open source projects in general may not be representative of programs developed in industry. Embedded systems and financial applications, for example, are unlikely to be well represented among these open source projects.

Regarding randomized algorithms (in particular search and optimization algorithms), there are specific and rigorous theoretical reasons for which the choice of artifacts is extremely important. The *No Free Lunch* theorem states that, on average across all possible problems (i.e., artifacts), all search algorithms have the same performance [121]. If one does not clearly define which is the *space* of artifacts being targeted, then any comparison among randomized algorithms is doomed to be arbitrary. For example, consider again the example of unit testing of object-oriented software. Assume that a case study involves 10 classes, and algorithm $\mathcal{A}$ is statistically better on seven of them, whereas algorithm $\mathcal{B}$ is statistically better on the other three. One could naively claim that algorithm $\mathcal{A}$ is *on average* better than $\mathcal{B}$. But, maybe, those seven classes for which $\mathcal{A}$ is better are all container classes, whereas the other three classes are related to string manipulations (e.g., [4]). If one had chosen for the case study more classes of this latter type, then the conclusions could be different (i.e., $\mathcal{B}$ would be considered *on average* better than $\mathcal{A}$). Though the problem of choosing *appropriate* artifacts is intrinsically difficult, it is important for researchers to define their target artifacts as well as possible and carefully attempt to provide plausible reasons for differences in results across artifacts, such as classes, based on a thorough analysis of their characteristics.

Ideally, when realistic artifacts for a certain type of problems are difficult to find, one would like to be able to generate large numbers of them automatically in a realistic fashion. However, this requires that the artifacts have a clear and predictable structure, that there exist heuristics to generate correct and meaningful instances of such artifacts. If this is possible, one strong advantage is that one can control and vary interesting properties of the artifacts (e.g., class size, number of test cases) to enable interesting sensitivity analyses and assess the performance of randomized algorithms as a function of these properties. For example, in the work of Hemmati *et al.* [59], the authors analyzed different test suite reduction techniques for model-based testing of large systems. Obtaining real models from industry is difficult, and UML models of real systems are not common in open source repositories. Although the case study was based on two real industrial systems (e.g., one provided by Cisco Systems), to cope with possible threats to external validity, the authors also used a large set of artificially generated test suites following some specific rules and a randomized construction algorithm. For example, the number of test cases in the test suites and the fault detection rate were varied in order to assess their impact on the effectiveness of the resulting selection technique. The aim was to do so while retaining as much as possible the realism of the test suites in the case studies. Such studies may be considered a type of simulation and may not generate fully realistic artifacts. But they may provide useful insights into the impact of some artifact properties on the effectiveness of a randomized algorithm.

For some types of software engineering problems, a large number of artifacts can be selected or generated (e.g., randomly selecting classes to investigate the unit testing of open source software). When evaluating randomized algorithms in this context one has to make the following decision: Assume a budget for experiments

[4]http://sourceforge.net/, accessed November 2011.

$b = n \times z$ for each algorithm, where n represents the times a randomized algorithm is run on each artifact, and z is the number of these artifacts. If one considers b to be fixed (e.g., depending on how long it takes to run b experiments), then a practical and important question is how to choose n and z? Two extreme cases would be $(n = 1, z = b)$ and $(n = b, z = 1)$, but they would clearly lead to problems in terms of statistical testing and external validity, respectively. Researchers have to strike a balance between two objectives: one wants to analyze as many artifacts as possible to improve external validity and wishes, at the same time, to retain enough runs (i.e., n) to check whether there is a statistically significant difference on any single artifact when applying and comparing two randomized algorithms. This would, for obvious reasons, not be possible if $n = 1$. Though in Section 8 it was suggested as a rule of thumb to use $n = 1{,}000$ when possible, in certain circumstances this may not be an option. If one has the possibility to analyze a large number z of artifacts but has practical constraints regarding the number of experiments to be run (e.g., having experiments running on a PC for a couple of years would not be very practical), then it may be more appropriate to execute less runs, perhaps as low as $n = 30$ or even $n = 10$. But going lower than such values would make the use of standard statistical tests very difficult and, very likely, depending on the actual effect size and variance, would bring statistical power to unacceptably low levels.

As discussed in Section 3, there are cases in the literature (e.g., [90, 118]) in which a random instance generator is used, but then the algorithms are run only once (i.e., $n = 1$) on each artifact. For all the reasons discussed in this section, in general one would prefer to have a higher number of runs even if that would lead to use less artifacts. It is possible that there might be cases in which having $n = 1$ could be preferable. At any rate, in such cases it is recommended to properly clarify why the choice of using $n = 1$ was made, and to inform the reader of the possible validity threats related to statistical power and representativeness of the case study.

10.2 Analysis of Multiple Artifacts

If for the addressed research question the considered artifacts can be considered representative of the target, it is meaningful to then use statistical tests for evaluating whether algorithm $\mathcal{A}$ is significantly better than $\mathcal{B}$ on all selected artifact instances. However, as it will be shown below, which type of test is used is of the highest importance. Using again the same example described before, assume six classes have been selected for investigating the unit testing of object-oriented software. Each algorithm is run on each of these six classes n times (e.g., $n = 30$), and average values out of these runs are collected for each class. This makes up a total of $2 \times 6 \times 30 = 360$ runs. Assume that the algorithms are evaluated based on how many test cases they generate before reaching full coverage. For the first algorithm, assume that a researcher obtains the following average values $X = \{10k, 20k, 30k, 40k, 50k, 60k\}$, whereas for the second algorithm she obtains $Y = \{12k, 21k, 34k, 41k, 53k, 68k\}$. The average values are ordered by problem instance where $k = 1000$, i.e., in X, out of $n = 30$ runs on the first artifact the average number of test cases run equals $10{,}000$. Further assume that the problem instances are ordered by difficulty (i.e., solving the first problem is much easier than solving the fifth, because on average it requires to generate/run less test cases). If one wants to evaluate whether there is any statistical difference between X and Y, an *unpaired test*, such as Mann-Whitney U-test, would yield a *p*-value equal to 0.699 (e.g., by using the R [97] command "wilcox.test(X,Y)"), thus suggesting the difference is not statistically significant. However, this would be technically incorrect since different artifacts present different levels of difficulty, and considering all data together at the same time would blur the relative performance of the compared algorithms. In other words, a run of an inefficient algorithm on an *easy* problem would likely result in a better value than a run of a more efficient algorithm that is run instead on a *difficult* problem. If the case study involves artifacts of different levels of difficulty (as it is usually the case, either by design or due to random sampling) then it might be challenging to detect any statistical difference with an unpaired test.

Alternatively, *paired tests* such as the Wilcoxon rank sum test can be used (e.g., "wilcox.test(X,Y, paired=TRUE)" in R [97]). In a paired rank sum test, what is evaluated is whether the differences $Z_i = Y_i - X_i$ are centered around 0, i.e., the null hypothesis is $Z = 0$. In that example, it would be $Z = \{2k, 1k, 4k, 1k, 3k, 8k\}$, i.e, on average the second algorithm is always better than the first. A Wilcoxon rank sum test here yields *p*-value $= 0.035$, which suggests a statistically significant difference among the performance of the two algorithms, a result in sharp contrast with the unpaired test results above. This highlights why it is extremely important to use paired tests when comparing randomized algorithms on a set of selected artifacts. Another similar approach would be to calculate the effect sizes and check whether they are symmetric around the null hypothesis. Assume for example that the resulting $\hat{A}_{XY}$ effect sizes are equal to $ES = \{0.4, 0.4, 0.4, 0.4, 0.4, 0.4\}$ (note, their actual values are not important as long as they are lower than 0.5). Then a test for symmetry in R would be "wilcox.test(ES, mu=0.5)", which would result in a *p*-value equal to 0.019.

In the above example, the first algorithm is better in six out of six cases, which is a clear case. But typically results are not that consistent, and several of the compared algorithms may perform best on different artifacts. For example, assume a case study involving 100 artifacts: if an algorithm fares better on 51 of these, then the difference among the two would not be statistically significant when using a paired test. Using the example where an algorithm $\mathcal{A}$ is better than another $\mathcal{B}$ on some artifacts and worse on other artifacts, a paired rank sum test evaluates whether one algorithm is statistically better on a higher number of artifacts.

The above discussion on the use of appropriate statistical tests is incomplete as it considers the evaluation of a randomized algorithm as ternary, i.e., it is either better, equivalent or worse than another one. Consider the following example: algorithm $\mathcal{A}$ is better on 60% of the case study, but only by a very limited amount (where such "better" is defined based on the effect size). On the other hand, on the other 40% of the case study, it is much worse than algorithm $\mathcal{B}$. In this case, blindly applying a paired Wilcoxon rank sum test would lead to the conclusion that $\mathcal{A}$ is preferable, whereas a practitioner might prefer to use $\mathcal{B}$. Another option could be to collect standardized effect sizes for each problem instance, and then average them over all problems instances. This would provide additional information, but it would not solve the problem of fully describing the relative performance of two randomized algorithms, and would still be strongly dependent on the choice of the case study. Consider a case with five artifacts and the following $\hat{A}_{12}$ measures $\{0.6, 0.6, 0.6, 0.6, 0.1\}$. One algorithm is better than the other on four artifacts ($\hat{A}_{12} = 0.6$), but worse on the last one ($\hat{A}_{12} = 0.1$). If one averages those values on the entire case study, he would obtain $\hat{A}_{12} = 0.5$, thus suggesting there is no difference among the two algorithms! This example illustrates the fact that aggregate statistics on a set of artifacts are useful to summarize the comparisons of two (or more) algorithms, but only as long as particular care is taken to handle cases where sharp differences can be observed among artifacts. In general, researchers should report the performance of the algorithms on each problem instance separately and attempt, as discussed above, to explain differences. One useful way to show the relative performance of randomized algorithms on a set of artifacts is to use box-plots of the effect sizes, especially when dealing with many artifacts

11 Practical Guidelines

Based on the above discussions, this section summarizes a set of practical guidelines for the use of statistical tests in experiments comparing randomized algorithms. Though one would expect exceptions, given the current state of practice (see Section 3 and the systematic reviews of Ali *et al.* [3] and Kampenes *et al.* [63]), the authors of this paper believe that it is important to provide practical guidance that will be valid in most cases and enable higher quality studies to be reported. It is recommendable that practitioners follow these guidelines and justify any necessary deviation.

There are many statistical tools that are available. In the following, all the examples will be provided based on R [97], because it is a powerful tool that is freely available and supported by many statisticians. But any other professional tool would provide similar capabilities.

Practical guidelines are summarized below. Notice that often, for reasons of space, it is not possible to report all the data of the statistical tests. Based on the circumstances, authors need to make careful choices on what to report.

- When randomized algorithms are analyzed, clearly specify the number of runs and employed statistical tests. For example, they can be summarized in a threats to validity section, in which how randomness has been taken into account should be discussed and justified.

- On each artifact in the case study, run each randomized algorithm at least $n = 1{,}000$ times. If this is not possible, explain the reasons and report the total amount of time it took to run the entire case study. If for example 30 runs were performed and the total execution time was just one hour, then it is rather difficult to justify why a higher number of runs was not used to gain statistical power, lower *p*-values, and narrow the confidence interval of effect size estimates (Section 8).

- When a large number of artifacts can be used in the case study (e.g., for unit testing of open source software) but there are constraints in terms of execution time, then it is advisable to execute less runs per artifact (though at least $n = 10$) and use more artifacts (rather than having $n = 1{,}000$ but only few artifacts, see Section 10.1). The objective is to strike a balance between generalization and statistical power.

- The choice of artifacts, to which randomized algorithms are applied, has a large impact on the validity and statistical interpretation of the final results (Section 10.1). Ideally, a large unbiased selection of artifacts that are representative of the addressed problem should be used as case study. Even if obtaining such artifacts is usually not possible, it is important to always clarify how they were chosen. The aim is to allow the reader to properly interpret the results of the statistical analyses when more than one artifact is used in a case study.

- For detecting statistical differences, use the two-tailed non-parametric Mann-Whitney U-test for interval-scale results and the Fisher exact test for dichotomous results (i.e., in the cases of censored data as discussed in Section 6). For the former case, in R you can use the function "w=wilcox.test(X,Y)" where X and Y are the data sets with the observations of the two compared randomized algorithms. If you are comparing a randomized algorithm against a deterministic one, use the one-sample version of the test with "w=wilcox.test(X,mu=D)", where D is the resulting performance measure for the deterministic algorithm. When there are a successes for the first algorithm and b successes for the second, one should use "f=fisher.test(m)", where m is a matrix derived in this way: "m =matrix(c(a,n-a,b,n-b),2,2)".

- Report all the obtained *p*-values, whether they are smaller than α or not, and not just whether differences are significant. The motivation is for the reader to choose the level of risk that is suitable in her application context. When reporting all *p*-values is not possible (e.g., due to space reasons), one could report the proportion of significant test results: "x out of y tests were significant at α level ...".

- Always report standardized effect size measures. For dichotomous results, the odds ratio ψ can be calculated using Equation 2, where for example $\rho = 0.5$ (used to address zero occurrence cases [55]). For interval-scale results and the $\hat{A}_{12}$ effect size, the rank sum R_1 used in Equation 1 can be calculated with "R1=sum(rank(c(X,Y))[seq_along(X)])". It is also strongly advised to report effect size confidence intervals, e.g., by using a bootstrapping technique. In R, there is library *boot* from which the function "boot" (to do the sampling) and "boot.ci" (to create a confidence interval) can be used. A confidence interval is much easier to use than *p*-values for decision making as potential benefits can be compared to costs while accounting for uncertainty.

- To help the meta-analyses of published results across studies, report means and standard deviations (in case readers for some reasons want to calculate effect sizes in the d family). For dichotomous experiments, always report the values a and b (so that other types of effect sizes can be computed [55]).

- If space permits, provide full statistics for the collected data, as for example mean, median, variance, min/max values, skewness, kurtosis and median absolute deviation. Box-plots are also useful to visualize them.

- When analyzing more than two randomized algorithms, use pairwise comparisons including pairwise statistical tests and effect size measures. If the case study can be considered as a statistically valid sample, then you can also use a test for symmetry on the null hypothesis for the effect sizes (Section 10.2). For example, if ES contains the $\hat{A}_{12}$ effect sizes for each artifact in the case study, then "w=wilcox.test(ES,mu=0.5)" will tell whether one algorithm is better on a *higher number* of artifacts (but this would not take into account the *magnitude* of the improvement).

- If space permits, state the employed statistical tool and how it was used (there can be subtle differences on how the tests are computed).

12 Threats to Validity

The systematic review in Section 3 is based on only four sources, from which only 54 out of 246 papers were selected. Although this systematic review is larger than the majority of systematic reviews in software engineering [70], accounting for more sources of information might lead to different results. One can, however, safely argues that TSE and ICSE are representative of research trends in software engineering. Furthermore, that review is only used as a motivation for providing practical guidelines, and its results are in line with other larger systematic reviews [3, 63]. Last, papers sometimes lack precision and interpretation errors are always possible.

As already discussed in Section 11, the practical guidelines provided in this paper may not be applicable to all contexts. Therefore, in every specific context, one should always carefully assess them. For some specific cases, other statistical procedures could be preferable, especially when only few runs of the randomized algorithms are possible.

13 Conclusion

Randomized algorithms (e.g., Genetic Algorithms) are widely used to address many software engineering problems, such as test case selection. In this paper, as a first contribution, a systematic review is performed to evaluate how the results of randomized algorithms in software engineering are analyzed.

Similar to previous systematic reviews on related topics [3, 63], this review shows that most of the published results regarding the use of randomized algorithms in software engineering are missing rigorous statistical analyses to support the validity of their conclusions.

To cope with this problem, this paper provides, discusses, and justifies a set of *practical* guidelines targeting researchers in software engineering. In contrast to other guidelines in the literature for experimental software engineering [120] and other scientific fields (e.g., [89, 64]), the guidelines in this paper are tailored to the specific properties of randomized algorithms when applied to software engineering problems, with a particular focus on software verification and validation. The use of these guidelines is important in order to develop a reliable body of empirical results over time, by enabling comparisons across studies so as to converge towards generalizable results of practical importance. Otherwise, as in many other aspects of software engineering, unreliable results will prevent effective technology transfer and will inevitably limit the impact of research on practice.

Note that there are advanced topics in statistics that have not been discussed in this paper, as for example Bayesian data analysis [47]. This paper is not meant to be a complete and ultimate reference for experimenters in software engineering, but rather be an essential guide to help them to use fundamental and common statistical methods in an appropriate manner.

Acknowledgments

The authors of this paper would like to thanks Lydie du Bousquet and Zohaib Iqbal for useful comments on an early draft of this paper. The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE. Lionel Briand was also supported by a FNR PEARL grant, Luxembourg.

References

[1] R. Abraham and M. Erwig. Mutation Operators for Spreadsheets. *IEEE Transactions on Software Engineering (TSE)*, 35(1), 2009.

[2] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43:875–882, 2001.

[3] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. 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.

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

[5] J. H. Andrews, T. Menzies, and F. C. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering (TSE)*, 37(1), 2011.

[6] J. Antunes, N. Neves, M. Correia, P. Verissimo, and R. Neves. Vulnerability discovery with attack injection. *IEEE Transactions on Software Engineering (TSE)*, 36(3):357–370, 2010.

[7] A. Arcuri. Full theoretical runtime analysis of alternating variable method on the triangle classification problem. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 113–121, 2009.

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

[9] A. Arcuri and L. Briand. Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering (TSE)*, 2011. doi:10.1109/TSE.2011.85.

[10] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.

[11] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *SSBSE*, pages 33–47, 2011.

[12] A. Arcuri, M. Z. Iqbal, and L. Briand. 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)*, pages 95–110, 2010.

[13] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277, 2012.

[14] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

[15] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[16] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering (TSE)*, 36(4):474–494, 2010.

[17] F. Asadi, G. Antoniol, and Y. Gueheneuc. Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 153–162, 2010.

[18] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, 2001.

[19] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering (TSE)*, 36(4):495–508, 2010.

[20] M. Bowman, L. C. Briand, and Y. Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering (TSE)*, 36(6):817–837, 2010.

[21] R. Bryce and C. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability (STVR)*, 19(1):37–53, 2009.

[22] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1069–1075, 2005.

[23] E. Carrano, E. Wanner, and R. Takahashi. A multicriteria statistical based comparison methodology for evaluating evolutionary algorithms. *IEEE Transactions on Evolutionary Computation (TEC)*, (99):1–23, 2011.

[24] M. Chernick. Bootstrap methods: A practitioner\'s guide (wiley series in probability and statistics). 1999.

[25] J. Cohen. Statistical power analysis for the behavioral sciences, 1988.

[26] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, 1999.

[27] M. Cowles and C. Davis. On the origins of the .05 level of statistical significance. *American Psychologist*, 37(5):553–558, 1982.

[28] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The Human Competitiveness of Search Based Software Engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 143–152, 2010.

[29] J. del Sagrado, I. M. del Aguila, and F. J. Orellana. Ant Colony Optimization for the Next Release Problem. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 67–76, 2010.

[30] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering (TSE)*, 36(5):593–617, 2010.

[31] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):438–444, 1984.

[32] J. Durillo, Y. Zhang, E. Alba, and A. Nebro. A Study of the Multi-objective Next Release Problem. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 49–58, 2009.

[33] T. Dybå, V. Kampenes, and D. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology (IST)*, 48(8):745–755, 2006.

[34] P. Emberson and I. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Transactions on Software Engineering (TSE)*, 36(5):704–718, 2010.

[35] M. Fay and M. Proschan. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics Surveys*, 4:1–39, 2010.

[36] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1*. Wiley, 3 edition, 1968.

[37] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, 2011.

[38] G. Fraser and A. Arcuri. 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.

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

[40] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 2012.

[41] G. Freitag, S. Lange, and A. Munk. Non-parametric assessment of non-inferiority with censored data. *Statistics in medicine*, 25(7):1201, 2006.

[42] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 15–24, 2010.

[43] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 474–484, 2009.

[44] L. García. Escaping the Bonferroni iron claw in ecological studies. *Oikos*, 105(3):657–663, 2004.

[45] V. Garousi. A genetic algorithm-based stress test requirements generator tool and its empirical evaluation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):778–797, 2010.

[46] B. Garvin, M. Cohen, and M. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 13–22, 2009.

[47] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian data analysis*. Chapman & Hall/CRC, 2003.

[48] K. Ghani, J. Clark, and Y. Heslington. Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 122–131, 2009.

[49] G. Glass, P. Peckham, and J. Sanders. Consequences of failure to meet assumptions underlying the fixed effects analyses of variance and covariance. *Review of educational research*, 42(3):237–288, 1972.

[50] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 225–234, 2010.

[51] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Conference on Programming language design and implementation (PLDI)*, pages 213–223, 2005.

[52] S. Goodman. P values, hypothesis tests, and likelihood: implications for epidemiology of a neglected historical debate. *American Journal of Epidemiology*, 137(5):485–496, 1993.

[53] S. Goodman. Toward evidence-based medical statistics. 1: The P value fallacy. *Annals of Internal Medicine*, 130(12):995–1004, 1999.

[54] A. Griesmayer, R. P. Bloem, and C. Byron. Repair of boolean programs with an application to C. In *Computer Aided Verification*, pages 358–371, 2006.

[55] R. Grissom and J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum, 2005.

[56] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 55–64, 2010.

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

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

[59] H. Hemmati, A. Arcuri, and L. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 327–336, 2011.

[60] H. Hsu and A. Orso. MINTS: A general framework and tool for supporting test-suite minimization. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 419–429, 2009.

[61] J. Huo and A. Petrenko. Transition covering tests for systems with queues. *Software Testing, Verification and Reliability (STVR)*, 19(1):55–83, 2009.

[62] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224, 2010.

[63] V. Kampenes, T. Dybå, J. Hannay, and D. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology (IST)*, 49(11-12):1073–1086, 2007.

[64] M. Katz. *Multivariable analysis: a practical guide for clinicians*. Cambridge Univ Pr, 2006.

[65] K. Khan, R. Kunz, J. Kleijnen, and G. Antes. *Systematic reviews to support evidence-based medicine: how to review and apply findings of healthcare research*. RSM Press, 2004.

[66] U. Khan and I. Bate. WCET analysis of modern processors using multi-criteria optimisation. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 103–112, 2009.

[67] T. Khoshgoftaar, L. Yi, and N. Seliya. A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation (TEC)*, 8(6):593–608, 2004.

[68] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 199–209, 2009.

[69] D. Kim and S. Park. Dynamic Architectural Selection: A Genetic Algorithm Based Approach. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 59–68, 2009.

[70] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering-A systematic literature review. *Information and Software Technology (IST)*, 51(1):7–15, 2009.

[71] J. Klein and M. Moeschberger. *Survival analysis: techniques for censored and truncated data*. Springer Verlag, 2003.

[72] S. Kpodjedo, F. Ricca, G. Antoniol, and P. Galinier. Evolution and Search Based Metrics to Improve Defects Prediction. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 23–32, 2009.

[73] W. Kruskal and W. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, pages 583–621, 1952.

[74] Z. Lai, S. Cheung, and W. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 235–244, 2010.

[75] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: A tool for Search Based Software Testing for the C Language and its Evaluation on Deployed Automotive Systems. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 101–110, 2010.

[76] N. Leech and A. Onwuegbuzie. A Call for Greater Use of Nonparametric Statistics. Technical report, US Dept. Education, 2002.

[77] F. Lindlar and A. Windisch. A Search-Based Approach to Functional Hardware-in-the-Loop Testing. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 111–119, 2010.

[78] G. Lu, R. Bahsoon, and X. Yao. Applying Elementary Landscape Analysis to Search-Based Software Engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 3–8, 2010.

[79] A. Marchetto and P. Tonella. Search-based testing of Ajax web applications. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 3–12, 2009.

[80] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur. Scalable and Effective Test Generation for Role-Based Access Control Systems. *IEEE Transactions on Software Engineering (TSE)*, pages 654–668, 2009.

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

[82] P. McMinn. How Does Program Structure Impact the Effectiveness of the Crossover Operator in Evolutionary Testing? In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 9–18, 2010.

[83] T. Menzies, S. Williams, B. Boehm, and J. Hihn. How to avoid drastic software process change (using stochastic stability). In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 540–550, 2009.

[84] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering (TSE)*, 32(3):193–208, 2006.

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

[86] M. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[87] P. A. Nainar and B. Liblit. Adaptive bug isolation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 255–264, 2010.

[88] S. Nakagawa. A farewell to Bonferroni: the problems of low statistical power and publication bias. *Behavioral Ecology*, 15(6):1044–1045, 2004.

[89] S. Nakagawa and I. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 82(4):591–605, 2007.

[90] A. Ngo-The and G. Ruhe. Optimized Resource Allocation for Software Release Planning. *IEEE Transactions on Software Engineering (TSE)*, 35(1):109–123, 2009.

[91] S. Nijssen and T. Back. An analysis of the behavior of simplified evolutionary algorithms on trap functions. *IEEE Transactions on Evolutionary Computation (TEC)*, 7(1):11–22, 2003.

[92] A. Nori and S. K. Rajamani. An empirical study of optimizations in yogi. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 355–364, 2010.

[93] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[94] T. Perneger. What's wrong with Bonferroni adjustments. *British Medical Journal*, 316:1236–1238, 1998.

[95] M. Polo, M. Piattini, and I. García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability (STVR)*, 19(2):111–131, 2009.

[96] S. Poulding and J. Clark. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Transactions on Software Engineering (TSE)*, 36(6):763–777.

[97] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[98] J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51(11):1534–1548, 2009.

[99] J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2 edition, 1994.

[100] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE transactions on Neural Networks*, 5(1):96–101, 1994.

[101] G. Ruxton. The unequal variance t-test is an underused alternative to Student's t-test and the Mann-Whitney U test. *Behavioral Ecology*, 17(4):688–690, 2006.

[102] S. Sawilowsky and R. Blair. A more realistic look at the robustness and type II error properties of the t test to departures from population normality. *Psychological Bulletin*, 111(2):352–360, 1992.

[103] C. A. Schaefer, V. Pankratius, and W. F. Tichy. Engineering parallel applications with tunable architectures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 405–414, 2010.

[104] N. Schneidewind. Integrating testing with reliability. *Software Testing, Verification and Reliability (STVR)*, 19(3):175–198, 2009.

[105] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering (FASE)*, 2011.

[106] M. Shevertalov, J. Kothari, E. Stehle, and S. Mancoridis. On the Use of Discretized Source Code Metrics for Author Identification. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 69–78, 2009.

[107] M. Shousha, L. Briand, and Y. Labiche. A uml/marte model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. *IEEE Transactions on Software Engineering (TSE)*, 38(2), 2012.

[108] D. Siegmund. *Sequential analysis: tests and confidence intervals*. Springer, 1985.

[109] C. L. Simons, I. C. Parmee, and R. Gwynllyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering (TSE)*, 36(6):798–816, 2010.

[110] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 35–49, 2005.

[111] M. Stumptner and F. Wotawa. A model based approach to software debugging. In *International Workshop on Principles of Diagnosis*, 1996.

[112] T. Thum, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 254–264, 2009.

[113] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.

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

[115] P. Tonella, A. Susi, and F. Palma. Using Interactive GA for Requirements Prioritization. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 57–66, 2010.

[116] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[117] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 364–374, 2009.

[118] J. White, B. Doughtery, and D. Schmidt. Ascent: An algorithmic technique for designing hardware and software in tandem. *IEEE Transactions on Software Engineering (TSE)*, 36(6), 2010.

[119] R. Wilcox. *Fundamentals of modern statistical methods: Substantially improving power and accuracy*. Springer Verlag, 2001.

[120] C. Wohlin. *Experimentation in software engineering: an introduction*, volume 6. Springer Netherlands, 2000.

[121] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[122] J. Xiao and W. Afzal. Search-based resource scheduling for bug fixing tasks. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 133–142, 2010.

[123] Q. Yang and M. Li. A cut-off approach for bounded verification of parameterized systems. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 345–354, 2010.

[124] S. Yoo. A Novel Mask-Coding Representation for Set Cover Problems with Applications in Test Suite Minimisation. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 19–28, 2010.

[125] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering (TSE)*, 36(1):81–95, 2010.

[126] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 435–444, 2010.

[127] Y. Zhang and M. Harman. Search Based Optimization of Requirements Interaction Management. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 47–56, 2010.

[128] R. Zhao, M. Lyu, and Y. Min. Automatic string test data generation for detecting domain errors. *Software Testing, Verification and Reliability (STVR)*, 20(3):209–236, 2010.