A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

Lionel Briand
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
briand@simula.no

ABSTRACT

Randomized algorithms have been used to successfully address many different types of software engineering problems. This type of algorithms employ a degree of randomness as part of their logic. Randomized algorithms are useful for difficult problems where a precise solution cannot be derived in a deterministic way within reasonable time. However, randomized algorithms produce different results on every run when applied to the same problem instance. It is hence important to assess the effectiveness of randomized algorithms by collecting data from a large enough number of runs. The use of rigorous statistical tests is then essential to provide support to the conclusions derived by analyzing such data. In this paper, we provide a systematic review of the use of randomized algorithms in selected software engineering venues in 2009. Its goal is not to perform a complete survey but to get a representative snapshot of current practice in software engineering research. We show that randomized algorithms are used in a significant percentage of papers but that, in most cases, randomness is not properly accounted for. This casts doubts on the validity of most empirical results assessing randomized algorihtms. There are numerous statistical tests, based on different assumptions, and it is not always clear when and how to use these tests. We hence provide practical guidelines to support empirical research on randomized algorithms in software engineering.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General;
I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Reliability, Theory

Keywords

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 [27] and path coverage [22]. Because often these problems are undecidable, deterministic algorithms that are able to provide optimal solutions in reasonable time do not exist. The use of randomized algorithms [45] is hence necessary to address this type of problems.

The most well-known example of randomized algorithm in software engineering is perhaps *random testing* [13, 6]. Techniques that use random testing are of course randomized, as for example DART [22] (which combines random testing with symbolic execution). Furthermore, there is a large body of work on the application of *search algorithms* in software engineering [26], as for example Genetic Algorithms. Since practically all search algorithms are randomized and numerous software engineering problems can be addressed with search algorithms, randomized algorithms therefore play an increasingly important role. Applications of search algorithms include software testing [42], requirement engineering [8], project planning and cost estimation [2], bug fixing [7], automated maintenance [44], service-oriented software engineering [9], compiler optimisation [11] and quality assessment [33].

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 effectiveness of a randomized algorithm, it is important to study the *probability distribution* of its output or various performance metrics [45]. For example, a practitioner might want to know what is the execution time of those algorithms *on average*. But randomized algorithms can yield very complex and high variance probability distributions, and hence looking only at average values can be misleading, as we will discuss in more details in this paper.

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 we want to find failures in a software by using random testing (assuming that an automated oracle is provided). As a way to assess its performance, we can sample test cases at random until the first failure is detected. In the first experiment, we might find a failure after sampling 24 test cases (for example). We hence repeat this experiment a second time (if a pseudo-random generator is employed, we need to use a different seed for it) and then, for example, trigger the first failure when executing the second random test case. If in a third experiment we obtain the first failure 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 such randomness might hinder 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* [53]. There are many books on various aspects of statistics (e.g., [53, 10, 37, 25, 62]), and that research field is still growing [62]. Notice that though statistical testing is used in most if not all scientific domains (e.g., medicine and behavioral science), 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., [15, 29]) in the phenomena under study, the use of statistical tests is also required but the constraints we work with are quite different from those of randomized algorithms, such as for example the size of data samples and the types of distributions we deal with.

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 biology [47] and medicine [30]. In this paper, we intend to do the same for randomized algorithms in software engineering, as they entail specific properties and the application of statistical testing is far from easy, as we will see.

To assess whether the results obtained with randomized algorithms are properly analyzed in software engineering research, and therefore whether precise guidelines are required, we carried out a small scale systematic review. We limited our analyses to the year 2009 as our goal was not to perform an exhaustive systematic review but to obtain a representative, recent sample on which to draw conclusions. We focused on research venues that deal with all the aspects of software engineering, such as IEEE Transactions of Software Engineering (TSE), IEEE International Conference on Software Engineering (ICSE) and International Symposium on Search Based Software Engineering (SSBSE). The review shows that statistical analyses are either missing, inadequate, or incomplete. For example, though journal guidelines in medicine require a mandatory use of standardized *effect size* measurements [25] to quantify the effect of treatments, we have not found a single case in which this was used to measure the relative effectiveness of a randomized algorithm. Furthermore, in half of the surveyed empirical analyses, randomized algorithms were evaluated based on the results of only one run and all the empirical analyses in TSE were based on a maximum of five runs.

Given our survey's results, we hence found 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 [47, 30] and for other types of empirical analyses in software engineering [15, 29], they are not necessarily applicable in the context of randomized algorithms. Our objective is therefore 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 that were either not discussed or just briefly mentioned in [3] .

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

- We provide a systematic review of 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.

- We provide practical guidelines on the use of statistical testing that are tailored to randomized algorithms in software engineering applications and the specific properties and constraints they entail.

The paper is organized as follows. Section 2 presents the systematic review we carried out. Section 3 presents the concept of statistical difference in the context of randomized algorithms. Section 4 compares two kinds of statistical tests and discussed their implications in our context. The problem of censored data and how it applies to randomized algorithms is discussed in Section 5. How to measure effect sizes and therefore the practical impact of randomized algorithms is presented in Section 6. Section 7 investigates the question of how many times randomized algorithms should be run. The problems associated with multiple tests is discussed in Section 8. Practical guidelines on how to use statistical tests in our context are summarized in Section 9. The threats to validity of our work are discussed in Section 10. Finally, Section 11 concludes the paper.

2. SYSTEMATIC REVIEW

Systematic reviews are used to gather, in a unbiased and comprehensive way, published research on a specific subject and analyze it [31]. Systematic reviews are a useful tool to assess general trends in published research, and they are becoming increasingly common in software engineering [36, 15, 29].

In our review we want to analyze: (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 to analyze those data.

To answer RQ1, we selected two of the main venues that deal with all aspects of software engineering: IEEE Transactions of Software Engineering (TSE) and IEEE International Conference on Software Engineering (ICSE). We also considered the International Symposium on Search-Based Software Engineering (SSBSE), which is a specialized venue devoted to search algorithms. Because our goal is not to perform an exhaustive survey of existing works, but simply to get an up-to-date snapshot of current practice regarding the application of randomized algorithms in software engineering research, we only considered 2009 publications.

We only retained full length research papers and, as a result, 20 short papers at ICSE and eight at SSBSE were excluded. A total of 104 papers were considered: 48 in TSE, 50 in ICSE and nine in SSBSE. These papers were manually checked to verify whether in their empirical analyses randomized algorithms were used. This left a total of 16 papers using randomized algorithms: three in TSE (6.25% of the total 48), four in ICSE (8% of the total 50) and all the nine papers in SSBSE (100%).

Notice that we excluded papers in which it was not clear whether randomized algorithms were used. For example, the techniques described in [28, 58] use external SAT solvers, and those might be based on randomized algorithms, though we cannot say for sure. Furthermore, even if a paper focused on presenting a deterministic, novel technique, we included it when randomized algorithms were used for comparison purposes (e.g., fuzz testing [18]). Table 1 summarizes the results of this systematic review for the final selection of 16 papers. The first thing that results clear is that randomized algorithms are widely used in software engineering (RQ1): We found them in $6\% - 8\%$ of the articles in TSE and ICSE.

To answer RQ2, the data in Table 1 show the number of times a technique was run to collect data regarding its performance on each artifact in the case study. Most of the time, data are collected from only one run of the randomized algorithms. Only six cases out of 16 show at least 30 runs.

Regarding RQ3, only 5 out of 16 articles include empirical analyses supported by some kind of statistical testing. More specifically, we can see *t*-tests and U-tests for when algorithms are compared, and linear regressions when prediction models are built. However, no standardized *effect size* measures (Section 6) are reported in any of these articles to quantify the relative effectiveness of algorithms in an interpretable form.

Results in Table 1 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 1 cast serious doubts on the validity of most existing results.

Notice that some of empirical analyses in Table 1 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 [61]. However this algorithm was not compared against simpler alternatives or even random search. If we look more closely at the reported results in order to assess the implications of that lack of comparison, we see that the total number of fitness evaluations was 400 (a population size of 40 individuals that is evolved for 10 generations). This is an extremely low number (for example, for test data generation in branch coverage it is often the case of using 100,000 fitness evaluations for *each* branch [27]) and we can conclude that there is very limited search taking place, which implies that a random search would have likely yielded similar results. This is directly confirmed in the reported results in [61], in which in half of 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. This should not be surprising as the search operators were tailored to the specific, small set of bugs of the case study, which then led to an easy search problem. As discussed in [3], a search algorithm should always be compared against at least random search in order to check that the algorithm is not simply successful because the search problem is easy.

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 we provide specific guidelines on how to use statistical tests to support comparisons among randomized algorithms.

Table 1: Results of systematic review.

Reference	Venue	Repetitions	Statistical Tests
[1]	TSE	1/5	U-test
[41]	TSE	1	None
[48]	TSE	1	None
[43]	ICSE	100	*t*-test, U-test
[61]	ICSE	100	None
[18]	ICSE	1	None
[34]	ICSE	1	None
[4]	SSBSE	1000	Linear regression
[21]	SSBSE	30/500	None
[14]	SSBSE	100	U-test
[20]	SSBSE	50	None
[38]	SSBSE	10	Linear regression
[32]	SSBSE	10	None
[40]	SSBSE	1	None
[35]	SSBSE	1	None
[57]	SSBSE	1	None

3. 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, let us consider just one alternative randomized algorithm, and let us call it $\mathcal{B}$. 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 [22]).

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

To enable statistical analysis, we 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 used to assess whether there is enough empirical evidence to claim a difference between the two algorithms (e.g., the novel technique $\mathcal{A}$ is better than the current state of the art $\mathcal{B}$). A *null hypothesis* H_0 is typically defined to state that there is no difference between $\mathcal{A}$ and $\mathcal{B}$. A statistical test is used to verify whether we 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 4.

There are two possible types of error when performing statistical testing: (I) we reject the null hypothesis when it is true (we are claiming that there is a difference between two algorithms when actually there is none), and (II) we accept H_0 when it is false (there is a difference but we claim 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 we accept 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 our context we 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* [10] 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 our comparison of techniques $\mathcal{A}$ and $\mathcal{B}$, let us assume we obtain a p-value equal to 0.06. Even if one technique seems significantly better than the other in terms of effect size (Section 6), we 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 is type of reasoning can be counter-productive. The tradition of using $\alpha = 0.05$, discussed by Cowles [12], 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 [23, 24] who believe that such thresholds are arbitrary, and that researchers should simply report *p-values* and let the reader decide in context.

When we need to make a choice between techniques $\mathcal{A}$ and $\mathcal{B}$, we would like to use the one that is more likely to outperform the other. Whether we get a p-value lower than α bears little consequence from a practical standpoint, as in the end we *must* select an alternative, e.g., we must select a testing technique to verify the system. However, as we will show in Section 7, obtaining p-values lower than $\alpha = 0.05$ should not 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 6.

In practice, the selection of an algorithm would depend on the p-value of comparisons, the cost difference among algorithms (e.g., in terms of inputs), and the estimated effect size. Given a context-specific decision model, the reader, using such information, could then decide which technique is more likely to maximize benefits and minimizes risk. In the simplest case where compared techniques would have comparable costs, we would simply select the technique with the best performance 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.

4. PARAMETRIC VS. NON-PARAMETRIC TESTS

The two most used statistical tests are the *t*-test and the Mann-Whitney U-test. These tests are used to compare two 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 underlining 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 on the distribution of the data. *Why* there is 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, 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 tests should be used [16]. 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, but is often not necessary. In fact, the Central Limit theorem tells us that *t*-test and Welch test are robust even when there is strong departure from a normal distribution [53, 56]. But in general we cannot know how many data points (n) we need to reach reliable results. A rule of thumb is to have at least $n = 30$ for each data sample [53].

There are three main problems with such an approach: (1) if we need 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 might be powerful enough; (2) the rule of thumb $n = 30$ stems from analyses in behavioral science, and, to the best of our knowledge, 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. We now discuss points (2) and (3) in more details by accounting for the specific properties of the application of randomized algorithms in software engineering, using software testing examples. This choice was motivated by the fact that half the publications in search-based software engineering are on software testing [26].

Random testing, when 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) 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 [6]. Given θ the probability of sampling a test case that covers the desired testing target, then the expectation of random testing is $\mu = 1/\theta$ and its variance is $\delta^2 = (1-\theta)/\theta^2$ (see [17]). Figure 1 plots the density function of a geometric distribution with $\theta = 0.01$ and a normal distribution with same μ and δ^2. In this context, the density function represents the probability that, for a given number of sampled test cases l, we cover the target after sampling exactly l test cases. For random testing, the most likely outcome is $l = 1$, whereas for a normal distribution it is $l = \mu$. Notice that the geometric distribution is discrete (i.e., it is defined only on integer values), whereas a normal distribution is continuous. Furthermore, the density function of the normal distribution is always positive for any value, whereas for the geometric distribution it is equal to 0 for negative values, where in this context the values are the number of sampled test cases. Therefore, a testing technique can *never* follow a normal distribution in a strict way, although it might be a reasonable approximation.

As it is easily visible in Figure 1, the geometric distribution has a very strong departure from normality! Comparisons of novel techniques versus random testing (and this is the practice when search algorithms are evaluated [26]) using *t*-tests are hence very arguable. Furthermore, in contrast to many physical and behavioral phenomena, the probability distributions of search algorithms are often strongly departing from normality. A common example is when the search landscape of the addressed problem has trap-like regions [49].

The Central Limit theorem states that the *sum* of n random variables converges to a normal distribution [17]. For example, consider the result of throwing a dice. There are only six possible outcomes, each one with probability $1/6$. If we consider the *sum* of two dice (i.e., $n = 2$), we have 11 possible outcomes, from value 2 to 12. Figure 2 shows that with $n = 2$, in the case of dice, we already obtain a distribution that resembles the normal one, even though with $n = 1$ it is very far from normality. In our context, these random variables are the results of the n runs of the analyzed

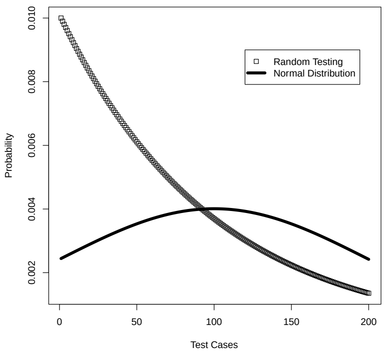

Figure 1: Density functions of random testing and normal distribution given same mean $\mu = 1/\theta$ and variance $\sigma^2 = (1-\theta)/\theta^2$, where $\theta = 0.01$.

algorithm. This theorem has three assumptions: the n variables should be independent and their mean μ and variance δ^2 should exist (i.e., they should be different from infinite). When using randomized algorithms, having n independent runs is usually trivial to achieve (we just need 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 [17]. Similarly, the existence of mean and variance in search algorithms is not always guaranteed, as discussed next.

If the performance of a randomized algorithm is bounded within a predefined range, then the mean and variance would always exist. For example, if an algorithm is run for a prefix amount of time to achieve structural coverage for software testing, and there are k structural targets, then the performance of the algorithm would be measured with a value between 0 and k. Therefore, we would have $\mu \leq k$ and $\delta^2 \leq k^2$, so using a *t*-test would be 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 we run a search algorithm on the same problem n times, and we have n variables X_i representing the number of test cases sampled in each run before triggering the first failure, we would estimate the mean with $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i$, and hence conclude that the mean exists. As Petersburg Game shows, this can be wrong, because $\hat{\mu}$ is only an *estimation* of μ, which might not exists.

For most search algorithms convergence in finite time is proven under some conditions (e.g., [54]), 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., [59]), complex non-standard search operators are required. Without formal proofs, 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 up-

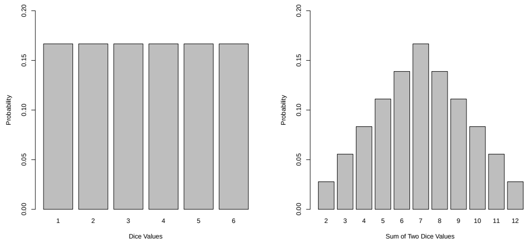

Figure 2: Density functions of the outputs of one dice and the sum of two dice.

per 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 (for example) if it has found no failure so far. But in these cases, we are actually dealing with *censored* data [37] (in particular, right-censorship) and this requires proper care in terms of statistical testing and the interpretation of results, as discussed in Section 5.

Even under proper conditions for using a parametric test, one aspect that is often ignored is that *t*-test and U-test are two different approaches to analyze two different properties. Let us use a random testing example in which we identify the first test case that triggers a failure Considering a failure rate θ, the mean value of sampled test cases done by random testing is hence $\mu = 1/\theta$. Let us assume that a novel testing technique $\mathcal{A}$ yields a normal distribution of the required number of test cases to trigger a failure. If we further consider the same variance as random testing and mean that is 85% of the one 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 but 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. 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 [17]. On one hand, half of the data points for random testing would be lower than 0.7μ. On the other hand, for $\mathcal{A}$ we have half of the data points above 0.85μ, and a significant proportion between 0.7μ and 0.85μ. This explains the apparent contradiction in results.

From a practical point of view, which statistical test should be used? Based on the discussions in this section, in contrast to [55] and in line with [39], we suggest to use Mann-Whitney U-test rather than *t*-test and Welch test. However, the full motivation will become clear only once we discuss censored data, effect size, and the choice of n in the next sections.

In the discussion above, we have assumed that both $\mathcal{A}$ 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 *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.

5. CENSORED DATA

Assume that the result of an experiment is dichotomous: either we find a solution to solve the software engineering problem at hand (*success*), or we do not (*failure*). For example, in software testing, if our goal is to cover a particular target (e.g., a specific branch), we can run a randomized algorithm with a time limit L. We will stop the algorithm as soon as we find a solution, otherwise we stop it after time L. The choice of L depends on the available computational resources. Another example is bug fixing [61] where we find a patch within time L, or we do not.

These types of experiments are dealing with *right-censored* data, and they properties are equivalent to survival/failure time analysis [37]. Let X be the random variable representing the time a randomized algorithm takes to solve a software engineering problem, and let us consider n experiments in which we collect X_i values. We are dealing with right-censorship since, assuming a time limit L, we will not have observations X_i for the cases $X > L$. There are several ways to deal with this problem [37] and we will limit our discussion to simple solutions.

One interesting special case is when we cannot say for sure whether we have achieved our target, e.g., generation of test cases that achieve code branch coverage. Even when using a time limit L, in these cases we are not tackling censored data. 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 we cannot know whether we have covered all feasible targets. However, if in the experiments the comparisons are made reusing a case study from the literature, and if we want to know whether within a given time we can obtain better coverage than reported studies, then such experimenst can be considered dichotomous despite infeasible targets.

Let us consider the case in which we need 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 $\mathcal{B}$. Let us assume that we run $\mathcal{A}$ n times collecting observations X_i, and we do the same for $\mathcal{B}$. Using a time limit L, to evaluate which of the two algorithms is better, we can consider their *success rate*, i.e., the number of times out of the n runs in which they find a valid solution. 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 [37].

If there is no statistically or practically significant difference between the two success rates, from a practical standpoint, the practitioner would then be interested to know which technique yields a valid solution in *less* time. This is particularly important if the success rates are high. There can be different ways to analyze such cases, such as considering artificial censorships at different times before L. For example, we can consider censorship at $L/2$, i.e., the success rate with half the time. Note that such analysis does not require to run any further experiments. Another way is to apply a Mann-Whitney U-test, recommended above, using only the times of successful runs, for which X_i and Y_i are lower than L. One more complex situation is when one algorithm shows a significantly higher success rate, but takes more time to produce valid solutions. Since these two variables are not necessarily correlated, a careful decision must then be made in these situations.

6. 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 large enough n we can obtain statistically difference even if that difference is 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 [25, 29, 47]. In their systematic review of empirical analyses in software engineering, Kampenes *et al.* [29] found out that standardized effect sizes were reported in only 29% of the cases. In our review, we found none.

Effect sizes can be divided in two groups: standardized and unstandardized. Unstandardized effect sizes are dependent from the unit of measurement used in the experiments. Let us consider the difference in mean 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 we might have $\Delta_1 = \mu^{\mathcal{A}} - \mu^{\mathcal{B}} = 100 - 1 = 99$, whereas on another testing artifact we might have $\Delta_2 = \mu^{\mathcal{A}} - \mu^{\mathcal{B}} = 100{,}000 - 200{,}000 = -100{,}000$. 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 this section we first describe which is the most known standardized effect size measure and why it should *not* be used. We then describe two other standardized effect sizes, and how to apply them in practice. The most known effect size is the so called d family which, in the general form, it 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 [25]). Though we obtain a measure that has no measurement unit, the problem is that it assumes the normality of the data, and strong departures can make it meaningless [25]. For example, in a normal distribution, roughly 64% of the points lie within $\mu \pm \sigma$ [17], 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" [25], because "standard deviations can be very sensitive to a distribution's shape" [25]. In this case, a non-parametric effect size should be preferred. Existing guidelines in [29, 47] just briefly discuss the use of non-parametric effect sizes.

The Vargha and Delaney's $\hat{A}_{12}$ statistics is a non-parametric effect size measure [60, 25]. Its use has been advocated in [39], and one example of its use in software engineering in which randomized algorithms are involved can be found in [51]. In our context, given a performance measure M, the $\hat{A}_{12}$ statistics 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 we would obtain higher results 70% of the time with $\mathcal{A}$. Though this type of non-parametric effect sizes is not common in statistical tools, it can be very easily computed [39, 25]. The following formula is reported in [60]:

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

where R_1 is the rank sum of the first data group we are comparing. The rank sum is a basic component in the Mann-Whitney U-test, and most statistical tools provide it. In that formula, 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, we would run two randomized algorithms the same number of times: $m = n$.

When dealing with dichotomous results (as discussed in Section 5), several types of effect size measures [25] 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" [25]. 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 $\psi = (a/n-a)/(b/n-b)$. There is no difference between the two algorithms when $\psi = 1$. The cases in which $\psi > 1$ implies 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 we compare two algorithms), 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 [25]. A $(1-\alpha)$ CI is a 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 we have $\hat{A}_{12} = 0.54$ 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 lead intuitively to decision making as benefits, which are directly related to effect size, can be compared to the costs of using alternative algorithms while accounting for uncertainty. To see how confidence intervals can be calculated, please see [25] and [60].

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 no enough statistical 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 $\psi = 1$.

7. NUMBER OF RUNS

How many runs do we need when we analyze and compare randomized algorithms? 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 [62].

Empirical studies of randomized algorithms 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., in 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., [5]) 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 runs should not be a problem and would solve most of the problems related to the power and accuracy of statistical tests. For example, as illustrated in [43, 14] in Table 1, even when 100 runs are used the U-test might be not powerful enough to confirm a statistical difference at a 0.05 significance level, even when the data seem to suggest such a difference.

Most discussions in the literature about statistical tests focus on situations with small numbers of observations (e.g., as in [55]). However, with thousands of runs, one would detect statistically significant differences on practically any experiment (see Section 3). It is hence essential to complement such analyses with the study of the effect size as discussed in Section 6. Even when having large numbers of runs may not be necessary for a set α level (e.g., 0.05) if differences of practical significance also show p-values less than α, additional runs would help tighten the confidence intervals for effect size and would be of practical value.

In Section 3, we 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. The choice of a non-parametric test would be driven by its effect size measure. In Section 6 we argued that effect size measures based on the mean (i.e., the d family) were not appropriate for randomized algorithms in software engineering. It would be pointless to detect statistical difference of mean values with a *t*-test if then we 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.

8. MULTIPLE TESTS

In most situations, we need to compare several alternative algorithms. Furthermore, if we are comparing different algorithm settings (e.g., population sizes in a Genetic Algorithm), then each setting technically defines a different algorithm. 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 size has been defined for those cases [25]. However, in our application context, we would like to know the performance of each algorithm compared against all other alternatives individually. Given a set of algorithms, we would not be interested to simply determine whether all of them have the same mean values. Rather, given K algorithms, we want 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 we have only one comparison, the probability of Type I error is equal to the obtained p-value. If we have 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 we 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* [50, 46]. Instead of applying each test assuming a significance level α, we would use an adjusted level α/Z. For example, if we want a 0.05 probability of Type I error and we have two comparisons, we would need to use two statistical tests with a 0.025 α, and then check whether both differences are significant (i.e., if both p-values are lower than 0.025). However, the Bonferroni adjustment has been seriously criticized in the literature [50, 46], and we largely agree with those critiques. For example, let us assume that for both those tests we obtain p-values equal to 0.04. If a Bonferroni adjustment is used, then both tests will not be statistically significant. A researcher could be tempted 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 [50, 46]. This would be particularly true in our application context in which 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. Though we do not recommend the Bonferroni adjustment, it is important to always report the obtained p-values, not just whether a difference is significant or not. If for some reasons the readers want to evaluate the results using a Bonferroni adjustment or any of its variants, then it is possible to do so. For a full list of other problems related to the Bonferroni adjustment, please see [50, 46]. Notice that there are other adjustment techniques that are equivalent to Bonferroni but that are less conservative [19]. However, the statistical significance of a single comparison would still depend on the number of performed and reported comparisons.

In Section 3 we stated that in software engineering in general, and for randomized algorithms in particular, we mostly deal with decision-making problems. For example, if we must test software and must choose one alternative among K different techniques. In this case, even if the p-values are higher than α, we need to test the software anyhow and we must make a choice. In this context, Bonferroni-like adjustments make even less sense. Just choosing one alternative at random because there is no statistically significant difference does not make much sense as it ignores available information.

9. PRACTICAL GUIDELINES

Based on the above discussions, we propose a set of practical guidelines for the use of statistical tests in experiments comparing randomized algorithms. Though we expect exceptions, given the current state of practice (Section 2 and [3, 29]), we believe that it is important to provide practical guidance that will be valid in most cases and enable higher quality studies to be reported. We recommend that practitioners follow these guidelines and justify any necessary deviation.

There are many statistical tools that are available. In the following we will provide examples based on R [52], 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 as follows:

- On each problem instance (e.g., program) 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.

- For detecting statistical differences, use the 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 5). 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 "w=wilcox.test(X,mu=D)", where D is the resulting measure of the deterministic algorithm. When we have number of successes a for the first algorithm and b for the second, you can 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.

- Always report standardized effect size measures. For dichotomous results, the odds ratio ψ (and its confidence interval) is automatically calculated with "f=fisher.test(m)". 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 (but the support for $\hat{A}_{12}$ is unfortunately limited). This is in fact a much easier to use substitute to p-values for decision making where 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 (so that effect sizes in the d family can be used). For dichotomous experiments, always report the values a and b (so that other types of effect sizes can be computed [25]).

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

- When analyzing more than two randomized algorithms, use pairwise comparisons followed by pairwise statistical tests and effect size measures.

- Always states the employed statistical tool (there can be subtle differences on how the tests are computed).

10. THREATS TO VALIDITY

The systematic review in Section 2 is based on only three sources, from which only 16 out of 132 papers were selected. A larger review might lead to different results, although we can safely argue 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, 29]. Last, papers sometimes lack precision and interpretation errors are always possible.

As already discussed in Section 9, our practical guidelines 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 are possible.

11. CONCLUSION

In this paper we report on a systematic review to evaluate how the results of randomized algorithms in software engineering are analyzed. This type of algorithms (e.g., Genetic Algorithms) are widely used to address many software engineering problems, such as test case selection. Similar to previous systematic reviews on related topics [3, 29], we conclude that the use of rigorous statistical methodologies are somehow lacking when investigating randomized algorithms in software engineering.

To cope with this problem, we provide *practical* guidelines targeting researchers in software engineering. In contrast to other guidelines in the literature for other scientific fields (e.g., [47] and [30]), the guidelines in this paper are tailored to the specific properties of randomized algorithms when applied to software engineering problems. The use of these guidelines is important in order to develop a reliable body of empirical results over time, which enable comparisons across studies and which will converge towards generalizable results of practical importance. Otherwise, as in many other aspects of software engineering, unreliable results would prevent effective technology transfer and would limit the impact of research on practice.

12. ACKNOWLEDGEMENTS

We would like to thanks Lydie Du-Bousquet 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.

13. 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)*, 2009.

[4] 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.

[5] 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)*, 2010.

[6] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 219–229, 2010.

[7] 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.

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

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

[41] 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.

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

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

[44] 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.

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

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

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

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

[49] 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.

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

[51] S. Poulding and J. Clark. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Transactions on Software Engineering (TSE)*, 2010.

[52] 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.

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

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

[55] 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.

[56] 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.

[57] 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.

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

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

[60] 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.

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

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