FAST Approaches to Scalable Similarity-based Test Case Prioritization

Breno Miranda
Federal University of Pernambuco
Recife 50740-540, Brazil
bafm@cin.ufpe.br

Emilio Cruciani
Gran Sasso Science Institute
L'Aquila 67100, Italy
emilio.cruciani@gssi.it

Roberto Verdecchia*
Gran Sasso Science Institute
L'Aquila 67100, Italy
roberto.verdecchia@gssi.it

Antonia Bertolino
ISTI - CNR
Pisa 56124, Italy
antonia.bertolino@isti.cnr.it

ABSTRACT

Many test case prioritization criteria have been proposed for speeding up fault detection. Among them, similarity-based approaches give priority to the test cases that are the most dissimilar from those already selected. However, the proposed criteria do not scale up to handle the many thousands or even some millions test suite sizes of modern industrial systems and simple heuristics are used instead. We introduce the *FAST* family of test case prioritization techniques that radically changes this landscape by borrowing algorithms commonly exploited in the big data domain to find similar items. *FAST* techniques provide scalable similarity-based test case prioritization in both white-box and black-box fashion. The results from experimentation on real world C and Java subjects show that the fastest members of the family outperform other black-box approaches in efficiency with no significant impact on effectiveness, and also outperform white-box approaches, including greedy ones, if preparation time is not counted. A simulation study of scalability shows that one *FAST* technique can prioritize a million test cases in less than 20 minutes.

CCS CONCEPTS

• **Software and its engineering** →**Software testing and debugging;** • **General and reference** →*Verification;* Metrics; • **Information systems** →*Similarity measures;*

ACM Reference format:
Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST Approaches to Scalable Similarity-based Test Case Prioritization. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
DOI: 10.1145/3180155.3180210

*Also with Vrije Universiteit Amsterdam, 1081HV, The Netherlands.

ICSE '18, Gothenburg, Sweden
 978-1-4503-5638-1/18/05...$15.00
DOI: 10.1145/3180155.3180210

1 INTRODUCTION

Test case prioritization (TCP) is a very active topic in software testing research [4, 15, 21, 33]. The goal of TCP is to speed up fault detection: It re-orders a test suite so that those test cases that are most likely to fail are executed first.

TCP is typically applied in regression testing [30, 33]: As changes are introduced into a software system, previously saved test cases need to be re-executed to ensure quality and stability. By Herzig [16], answering the question "what to test next" is one of the five top things that require automation in industrial software testing. TCP can help to detect faults more quickly and to provide confidence that, should testing be stopped before all test cases are run, the ones executed are the most effective. In the years, researchers have proposed many prioritization criteria that exploit different information related to the test cases: Early approaches were based on code coverage [9, 30]; more recently, black-box criteria based on system models [31], on requirements [1], or on historical failure data [28] are investigated, as with the growing scale of regression testing, coverage measures can hardly be afforded.

Since [30], the effectiveness of TCP techniques is evaluated by their Average Percentage of Faults Detected (APFD). This measure captures how fast a re-ordered test suite detects faults, which is certainly an important concern in test prioritization. However, APFD does not consider how fast the prioritization approach itself is. As said by Henard et al. [15], *if prioritization takes too long, then it eats into the time available to run the prioritized test suite.*

As also noticed in [34], for real-world software the size of a test suite can often exceed the size of the system under test. In contrast, the time available for test execution cycles decreases, especially with companies migrating towards rapid release [23] or continuous integration [8] practices. Memon et al. [27] report that every day at Google an amount of 800K builds and 150M test runs are performed on more than 13K code projects. In line with [8, 27], we notice that most TCP approaches in the literature cannot handle such scale. Our experimental results show that some TCP approaches become soon inefficient even for small-medium size benchmarks.

In this paper, we look at TCP from a novel perspective: We acknowledge that test suite sizes grow at fast pace, and existing techniques and tools for regression testing become inadequate. Making the appropriate scale distinctions, the management of test suites in large scale industrial projects is becoming a *big data* problem. Big data are "datasets whose size is beyond the ability of typical database software tools

to capture, store, manage, and analyze" [24]. In a similar way, we coin the term "big testsets" to denote sets of test cases whose dimensions go beyond the capacity of existing testing tools. A testset dimension can be "big" due to either the number of test cases (e.g., millions of tests) or the size of test case related information (e.g., coverage measures for huge programs), or the combination of both. We borrow well established data mining techniques for handling big testsets.

Indeed, finding "similar" items is a fundamental problem in data mining, and very efficient algorithms exist to address this problem. In software testing, diversity has been shown to be an effective measure for both selecting and prioritizing test cases, and several similarity-based approaches have been proposed [11, 14, 17, 19, 34]. However, such approaches do not scale up to big testsets. In this paper, we present *FAST*,[1] a family of similarity-based TCP techniques that employs *minhashing* and *locality-sensitive hashing* algorithms [20] for quickly finding "diverse" test cases within a big set. *FAST* can perform white-box (WB) or black-box (BB) prioritization, and can be tuned to yield higher or lower efficiency by trading off with precision in similarity estimation.

We assessed the effectiveness, efficiency, and scalability of five techniques from the *FAST* family applied to both WB (function, statement, and branch coverage) and BB testsets (i.e., 20 techniques in total), against several competing approaches (17 techniques in total). The results on 10 commonly used benchmarks (both C and Java) showed that the fastest techniques from the *FAST* family outperformed the BB competitors in efficiency with no significant impact on effectiveness. In comparison with WB approaches, they required the shortest prioritization time net of preparation time, even against greedy approaches. Indeed, our simulation study shows that after preparation is done, *FAST* efficiency is independent from the size of the test cases, and one *FAST* technique can prioritize 1M size testsets in less than 20 minutes, whereas even greedy total performance degrades when dealing with big dimensions.

Summarizing, the contributions of this work include:

- The first proposal of exploiting data mining algorithms for similarity-based testing of big testsets.
- The definition and implementation of the *FAST* family of similarity-based TCP techniques.
- A large scale experimentation of 20 *FAST* techniques, compared for effectiveness, efficiency, and scalability against 17 competing approaches.
- The release of an automated framework and all the data used for the experiments in order to support replicability and follow up studies.

The paper is structured as follows. Related work is overviewed in the next section. Background information behind similarity-based TCP and the employed algorithms is provided in Section 3. The *FAST* approaches, the performed experiments, and the results achieved are described in Sections 4, 5, and 6, respectively. Section 7 concludes the paper.

[1] *FAST* is a recursive acronym for *FAST* Approaches to Similarity-based Testing.

2 RELATED WORK

This work proposes a family of novel TCP approaches based on similarity that explicitly addresses scalability.

Related work on test case prioritization. The literature on TCP is huge, testifying the great interest in the topic from both academic and industrial perspectives. For an overview of existing work we refer to [4, 12, 33]. Catal and Mishra [4] present a systematic mapping study of TCP over the period 2001-2011. The work from Yoo and Harman [33] is a broad survey on regression testing. Concerning TCP, they categorize techniques based on the information used for ordering test cases, including coverage-based, interaction-testing (which considers different combinations of components), and "others" (including distribution-based, human-based, probabilistic, requirement-based, model-based). Hao et al. [12] provide an overview of TCP research considering five aspects: test adequacy criteria, algorithms used, measures adopted, constraints considered, and application scenarios. Notably, they observe that efficiency is important and TCP becomes unbearable when the prioritization time gets close to test execution time. This work fully shares such consideration.

Related work on similarity-based TCP. Both Jiang et al. [17] and Zhou et al. [34] have proposed ART-based prioritization techniques guided by code-coverage. ART (Adaptive Random Testing) [5] is a variant of random test generation that tries to spread as evenly as possible the test inputs in the input domain. In our experimentation we compare the *FAST* approaches against both [17, 34], which are further described in Section 5. Also the approach proposed by Fang et al. [10] is based on code coverage information, from which they exploit the execution frequency profiles.

Among black-box approaches, Ledru et al. [19] propose a similarity-based approach solely considering the strings that express the test cases, i.e., the input data or the JUnit test cases. We also compare *FAST* against this approach (see Section 5). Noor and Hemmati [28] develop a history-based approach in which, among new or modified test cases, those that are the most similar to failing ones are prioritized. *FAST* does not currently use history data.

Related work on scaling up test prioritization. It is important to consider the applicability of proposed TCP approaches to real world testing environments. Busjaeger and Xie [2] identify heterogeneity, scale, and cost as the practical realities to address in TCP and propose to rank the test cases by using machine learning techniques trained on five features (code coverage, text path and content similarity, failure history, and test age). The ROCKET approach by Marijan et al. [25] implements an automated TCP approach considering failure history and test execution time, and compares it against manual approach. Elbaum et al. [8] propose a regression test strategy for continuous integration environments based on execution history data that combines techniques of test case selection and prioritization. We observe that approaches conceived for handling huge test suites generally embed specific heuristics conceived on the basis of the studied industrial

process. In contrast, *FAST* is a generally applicable approach as it does not embed any *ad-hoc* heuristic.

3 BACKGROUND

3.1 Similarity-based TCP

The TCP problem can be defined as follows [30]: Given a test suite S, the set P of permutations of S, and an award function $f : P \to \mathbb{R}$, then TCP consists in finding a $T \in P$ such that $f(T) \geq f(T')$ for all $T' \in P$ with $T' \neq T$.

Ideally, the award function f refers to the rate at which faults are detected in the given ordering. In reality, TCP approaches can only be based on surrogate criteria [33], and as discussed in the previous section fairly different techniques have been suggested. In particular, the idea at the basis of similarity-based approaches for test prioritization (STP in short) is to reward the diversity between test cases.

In Figure 1 we provide the scheme of a generic STP process. It consists of three main activities: (i) encoding of test-related information; (ii) evaluating similarity; (iii) picking out the next test case(s). The similarity between two test cases can be evaluated in many different ways, also depending on the adopted test strategy. For example, in coverage-based testing similarity between test cases is evaluated by considering set similarity measures among their respective coverage, as in [17, 34]; in model-based testing, instead, by considering the overlap between the traces covered by the tests over a state-based model of the system under test, as in [3, 13]. Several other features related to test cases have been taken into account, e.g., historical data failure, the test input string, etc. Therefore, before applying any STP, a preparation phase is needed (Step 1), in which the information related to the test cases is collected and encoded (other processing of such data may also be needed depending on the approach). Such data is then processed to calculate the similarity with respect to the already picked test cases (Step 2). In fact STP proceeds in iterations: At each iteration there exists a set of already picked test cases (denoted as "so-far ordered tests"), to which the test cases yet to be ordered are compared. Following Step 2, we obtain a ranking of the coded test information. From this, one or more test cases are picked and added to the set of so-far ordered tests (Step 3). The process terminates when the whole testset is ordered (compatibly with available resources).

3.2 Algorithms for Similarity Estimation

Finding similar items is a fundamental problem in *data mining*, and very efficient techniques have been developed to solve it [20]. Typical tasks that face such problem include finding plagiarized documents, detecting mirror pages, identifying articles coming from the same source. As the number of test cases to prioritize grows in size up to millions [8, 16], the idea of applying such efficient techniques in STP seems the natural way to go.

The naive approach for similarity computation between n items needs to perform all the pairwise comparisons and becomes inefficient as n grows. In this work we measure the

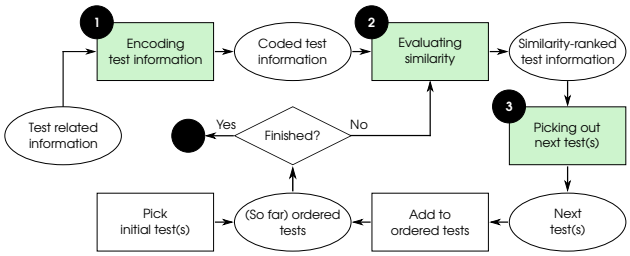


Figure 1: Overview of STP.

similarity of two sets A and B as their Jaccard similarity $\mathrm{JS}(A, B) = |A \cap B| \,/\, |A \cup B|$, and the distance between them as their Jaccard distance $\mathrm{JD}(A, B) = 1 - \mathrm{JS}(A, B)$. This choice allows us to leverage and combine techniques able to drastically reduce the cost of computing similarity.

The first technique, called *Shingling* [22], is used to represent items as sets of "shingles", out of which the similarity is computed. Given a string of characters, a k-shingle of that string is the set of its substrings of length k. For example, the 3-shingles set of the string *gzip* is $\{gzi, zip\}$. Any document can be represented by a set of k-shingles and if two documents are similar they will have many shingles in common. We use k-shingles in BB prioritization to make a set out of the string representation of the input test cases.

Sets of shingles, though, are larger than the original data and for huge datasets it is not practical to use them directly. The second technique we leverage is *Minhashing* [20], which derives compact representations of sets, called *signatures*. Minhash signatures have the nice property of preserving the Jaccard similarity between the sets they represent. A minhash of a set S is computed as follows: A given hash function g is used to hash all the elements in S and the minimum resulting value becomes the minhash of S. This process is repeated multiple times, e.g., using h different hash functions, to generate a sequence of minhashes which forms the signature of S.[2] The estimation of the Jaccard similarity between two sets can be computed by counting the fraction of minhashes that agree in the signatures of the sets. For example, the signatures $(1, 1, 3, 2, 4)$ and $(1, 2, 1, 2, 3)$ have an estimated Jaccard similarity of $2/5$ since they agree in the first and fourth minhash.

Minhashing helps to compress large items into small signatures, but the signature pairs to be compared can still be beyond feasibility if the number of items is high. The third technique we use is *Locality-Sensitive Hashing* (LSH) [20], which reduces the scope of comparison to only a subset of items that are likely to be similar, the *candidate set*. LSH works on the *signature matrix*, i.e., the matrix that has minhash signatures as columns, by dividing it into b bands of r rows each and applying a hash function to them. In particular, for each band, the vectors of r integers located in the columns are hashed into several *buckets*: If it happens

[2] A signature of length h guarantees an expected error of $\mathcal{O}(1/\sqrt{h})$ in the similarity estimation.

that two of these vectors hash into the same bucket then it means that a portion of their signatures agree and the pair is added to the candidate set. In the end the candidate set will contain all the pairs of sets which are likely to be similar, i.e., all the pairs that have a Jaccard similarity over a certain threshold s. Such threshold depends on the choice of the values b and r and a good approximation is $s \approx (1/b)^{(1/r)}$.

4 APPROACH

Algorithm 1 provides a pseudocode description of *FAST*. In the preparation phase (Lines 2 and 3), *FAST* uses the coded test information T to create the minhash signatures M. Note that *this is the only operation where FAST handles BB and WB inputs differently.* While for WB the code coverage information can be directly represented as sets (regardless of the coverage criterion), for BB the string representation of the test cases needs to be preprocessed into k-shingles. We used $k = 5$ to have a suitable set representation.[3] Once the minhash signatures are computed, T is not required anymore and only M is used during the prioritization process. We used a number of hash functions $h = 10$, which guarantees an expected error not greater than 0.32 in the estimation of the Jaccard similarity (and distance) between two signatures. Even if the error in the estimation is high, the choice of the next test case is performed over a subset of tests that are all dissimilar from the so-far-prioritized ones.

Algorithm 1: FAST prioritization.

```
Input  : Coded test suite info T; (optional) selection function f.
Output : Prioritized test suite P.
1  P ← EmptyList()
2  I ← GetTestCaseIDs(T)
3  M ← MHSignatures(T)                          ▷ No need of T from here on
4  B ← LSHBuckets(M)
       ▷ M(v): Cumulative signature of so-far-ordered test cases
5  M(v) ← MHSignature(∅)
6  while |P| ≠ |I| do
7      C_s ← LSHCandidates(B, M(v))
8      if C_s = ∅ then
9          M(v) ← MHSignature(∅)
10         C_s ← LSHCandidates(B, M(v))
11     C_d ← (I − P − C_s)                      ▷ Complement of C_s
12     s ← Select(M(v), M, C_d, f)
13     M(v) ← UpdateMHSignature(M(v), M, s)
14     M ← Remove(M, s)
15     P ← Append(P, s)
16 return P
17 function Select(M(v), M, C, f)
18     if no f then                                          ▷ FAST-pw
19         return arg max_{c ∈ C} { EstimateJD(M(v), M(c)) }
20     else                                                  ▷ FAST-f
21         return RandomSample(C, f)
```

In Line 4, the collection of LSH buckets is computed. Basically, B contains b buckets, one for each band, and each bucket keeps track of all the test cases colliding there. We defined the number of bands $b = 10$ and rows $r = 1$ such that the number of rows in the signature matrix is equal to the signature size, i.e., $h = r \cdot b$. These values guarantee a similarity threshold $s \approx 0.1$ for the candidate set. Notice that, while in finding the most similar items a higher similarity threshold would be better, for the context of STP we want to select the test cases that are *dissimilar* from the so-far-prioritized ones. Intuitively, with a similarity threshold $s \approx 0.1$ the candidate set will contain almost all the test cases but the ones that are dissimilar to $M(v)$ (i.e., having Jaccard distance greater than 0.9). In fact, the actual candidate set C_d used by *FAST* is computed (Line 11) as the complement of C_s, excluding the so-far-prioritized test cases.

The candidate sets are created inside the while loop (Lines 6 to 15), where the actual prioritization happens, as follows: $M(v)$ is divided into b bands; each band is hashed; and if there is a collision with the corresponding bucket in B, then the test cases of that bucket are added to the candidate set C_s. $M(v)$ is initialized in Line 5 and is updated whenever new test cases are selected (Line 13) to keep track of the cumulative signature of the so-far-ordered test cases. Whenever C_s is empty, we reset $M(v)$ (Lines 8 to 10) and recompute C_s. For *FAST*, such operation is analogous to what is done by some TCP approaches that reset the coverage vector when 100% of the achievable coverage is accomplished.

The function `Select` (Line 17) is where the *FAST* approaches differentiate from each other. *FAST-pw* computes the estimated Jaccard distance between $M(v)$ and each test case in the candidate set C_d using minhash signatures, and selects the candidate that is the farthest away from $M(v)$. The other approaches, instead, use a function f that is provided as input to the algorithm to select a random subset of C_d of size $f(|C_d|)$. In Line 15, the newly selected test case(s) are appended to the prioritized test suite P. For the experiments in this work we considered the following functions that progressively increase the efficiency of the prioritization: *one*, *log*, *sqrt*, *all*. In general, f is a generic function that can be tuned to achieve the right balance between the efficiency and the accuracy required in a particular context.

5 EXPERIMENTS

We describe the experiments conducted to assess the effectiveness, efficiency, and scalability of *FAST* in comparison with existing prioritization techniques.

5.1 Research Questions

The ultimate goal of any TCP approach is to reveal faults as quickly as possible. Therefore, our first and second research questions investigate the effectiveness and efficiency of *FAST*:

RQ1: [Effectiveness] How does *FAST* compare with other existing prioritization approaches in terms of *fault detection rate*?

RQ2: [Efficiency] How does *FAST* compare with other existing prioritization approaches in terms of *time* required for the prioritization?

[3] The typical BB input is smaller than 50^5 characters, which is the number of all the possible 5-shingles, considering an average of 50 characters (letters and symbols).

Effectiveness and efficiency of TCP have been extensively explored by researchers in previous work, e.g., [14, 15, 21]. However, one dimension that is usually not explicitly considered is the one of scalability. A given TCP approach might be effective and efficient for small-sized programs, but *to what extent does it scale to big, real-world-sized, programs?* We address this concern in our third research question:

RQ3: [Scalability] How does *FAST* compare with other existing prioritization approaches in terms of *scalability*?

5.2 Compared TCP Approaches

In order to conduct our experiments we had to decide which TCP approaches to consider for the comparison with *FAST*. We limited the scope of our study to *TCP approaches that require only test cases and/or coverage information as input*. This decision is justified by the fact that other types of inputs used by some TCP techniques (e.g., models or requirements) are not easily available for experimentation. Moreover, as these are the only inputs that *FAST* techniques require, this decision supports a fair comparison.

Because *FAST* is based on similarity, we started by searching the literature for state-of-the-art STP techniques that would meet our selection criterion. At this phase, the following approaches were selected (in bold within brackets we introduce the acronym used in the experiment description):

Jiang et al. [17] [**ART-D**] proposed a family of ART-based TCP techniques guided by coverage information. At each iteration, a *candidate set* is dynamically[4] created by randomly picking test cases from the set of not-yet-prioritized tests as long as they can increase coverage. The test case within the so built candidate set that is the farthest away from the set of already-prioritized tests is selected. The authors proposed and assessed different set distance functions; for our experiments we implemented the version that performed better (i.e., "*maxmin*").

Zhou et al. [34] [**ART-F**]: The essence of this TCP approach is the same of ART-D. The main differences are in the way the candidate set is created and in the distance metric adopted. While in [17] the candidate set has a flexible size, Zhou et al. [34] proposed a fixed[5] size (i.e., 10) for the candidate set. Besides, the authors used Manhattan distance instead of the Jaccard distance adopted by Jiang et al. [17].

Ledru et al. [19] [**STR**]: As said in Related work, this approach only uses test input strings. A greedy algorithm is applied that, at each iteration, picks the test case that is the most distant from the set of already prioritized ones. Several distance functions are evaluated, and Manhattan distance is the one recommended.

Considering other non-similarity based approaches, two natural choices are the well-known [30] Greedy Total [**GT**] and Greedy Additional [**GA**], which pick as the next test case the one that covers the largest number of entities in total or among those yet uncovered, respectively.

[4]This is why the D in the name label.
[5]This is why the F in the name label.

We finally looked at the results from some recent studies comparing TCP approaches, with the intent of selecting those emerging as the best ones. Excluding among the best TCP techniques indicated in [15] those already included in our list or using models in input, we could add 2 more competitors, one WB and one BB, described below.

Additional Spanning [**GA-S**] is a variant of GA that at each iteration picks the test case that covers the largest number of not-yet-covered entities among those in the "spanning set". In coverage testing an element subsumes another if covering the former guarantees also covering the latter: The notion of a spanning set was introduced in [26] to denote the subset of non-subsumed entities.

Feldt et al. [11] [**I-TSD**]: This approach proposes to use the Normalized Compression Distance between multisets introduced by [6] to measure the diversity of sets of test cases. While the measure is originally proposed as "universally" applicable to any test-related feature, the version performing better in [15] considers test inputs.

We also looked at the approaches compared in [21]. In this case, excluding the ones requiring additional information would yield to no new competing technique with respect to those already selected. Thus, summarizing, we collected as competing approaches: 2 BB ones, i.e., STR and I-TSD, and 5 WB ones, i.e., ART-D, ART-F, GA, GT, GA-S. For each WB approach, we implemented three variants, addressing function, statement, and branch coverage, totalizing 17 competitors techniques.

5.3 Evaluation Metrics

In order to assess prioritization effectiveness (RQ1), we use the Average Percentage of Faults Detected (**APFD**) [29]. APFD is calculated according to Equation (1), in which, given a test suite T containing n test cases and a set F of m faults revealed by T, for each ordering of T we denote as TF_i the position of the first test case that reveals fault i.

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n}. \qquad (1)$$

To answer our research questions on efficiency (RQ2) and scalability (RQ3) we assess the investigated TCP approaches in terms of **preparation time** and **prioritization time**. The preparation time considers the time spent by each TCP approach on tasks other than the prioritization itself (e.g., precomputing pairwise similarity between test cases), whereas the prioritization time considers only the time to process the already prepared test information and order the test suite. In reporting our results we also refer to **total time**, which is simply the sum of preparation and prioritization times. All times refer to the actually spent CPU time, which we measured by using Python's `time.clock()` function.[6]

5.4 Study Subjects

To answer research questions RQ1 and RQ2 we conducted experiments on 5 C and 5 Java programs. The C programs

[6]This is the function recommended by the Python Software Foundation for benchmarking or timing algorithms.

Table 1: Study subjects details.

Subject	LoC	Test LoC	#TM	#TC	Fault Type	#Faults
Flex	10296	-	670	-	seeded	9
Grep	10124	-	809	-	seeded	8
Gzip	4594	-	214	-	seeded	7
Sed	13413	-	370	-	seeded	6
Make	14330	-	875	-	seeded	19
Closure Compiler	90697	84585	8124	221	real	101
Commons Lang	21787	37957	2322	113	real	39
Commons Math	84323	86511	3877	385	real	7
JfreeChart	96382	49133	2278	356	real	26
Joda-Time	27801	53158	4160	123	real	27
Total:	373747	310344	23699	1198		249

LoC: lines of code; #TM: test methods; #TC: test classes.

(namely Flex, Grep, Gzip, Sed, and Make) were collected from the Software-artifact Infrastructure Repository (SIR) [7]. These programs are available in sequential versions, each containing a different number of seeded faults. The number of faults that can be revealed by the accompanying test suite varies greatly: Considering the extremes, in some cases no faults can be revealed, whereas in other cases multiple faults could be revealed by the vast majority of the test cases. To minimize the influence of these characteristics in our study, we selected, from each program, the version that contains the highest number of "hard-to-find" faults, i.e., faults that could not be detected by more than 50% of the test cases.

The 5 open-source Java programs investigated in our study (namely Closure Compiler, Commons Lang, Commons Math, JfreeChart, and Joda-Time) are integrated in the Defects4J framework [18]. For a given program, multiple versions are available, each containing a single fault. For our investigations we used as input for the TCP approaches the artifacts (i.e., coverage traces and test cases) from the first version of the program only. The effectiveness of the prioritized test suites are then assessed in the subsequent versions. For JfreeChart and Joda-Time, all the faults could be triggered by the test suite of the reference version. Thus, we could use all the available versions for these subjects. For the other programs, however, some faults could be revealed only by test cases that were not available in the reference test suite. For those programs, we considered the versions V_1 to V_{n-1}, with n being the first version for which the reference test suite could not trigger the existing fault. The number of versions we considered for each Java program is displayed in Table 1 (column "#Faults").

In total, we used 205 versions of 10 different programs.

5.5 Experiment Procedure

To answer RQ1 and RQ2, we applied the investigated TCP approaches to each experimental subject and measured: (i) the preparation time; (ii) the prioritization time; (iii) the APFD of the prioritized test suites. This process was repeated 50 times to account for the stochastic nature of the TCP approaches considered in our study (e.g., when more than one test case would be equally ranked by the TCP approach, a random choice is made to solve the tie). For the Java subjects, the APFD of the 50 prioritized test suites is computed for each of the subsequent versions considered for a given program.

To answer RQ3 we considered two dimensions that might hinder TCP scalability: (i) the size of the test suite; (ii) the size of the test case representation. Item (i) is important as it defines how many test cases need to be evaluated. While this might have little influence for some TCP approaches such as, e.g., GT, it might forbid the adoption of STP approaches that would depend on pairwise similarity computation. Item (ii), on its turn, is important as TCP approaches may use different test related information whose size can vary greatly: from just a few characters for command line programs, to thousands of words in the case of JUnit tests. For coverage traces, the size of test representation may change depending on both the coverage criteria and the program size.

To control these two dimensions in our experiments on scalability we used synthetic data. With respect to the first dimension, we considered test suite sizes that assume discrete values in the range from 1K to 1M as follows: from 1K to 10K in increments of 1K (i.e., 1K, 2K, 3K, ...); from 10K to 100K in increments of 10K; and from 100K to 1M in increments of 100K (yielding 28 different test suite sizes). With respect to the second dimension, we considered three different sizes for a test case representation: *small* for an average length of 100; *medium* for 1K; *large* for 10K elements. In all three cases we allowed for a variance of ±25%. We refer to generic "elements" for the size of test cases to be agnostic with respect to test criteria. Thus a small test case can be interpreted as a coverage trace from a test that covers ≈ 100 functions in the same way that it can be understood as the textual representation of a command line test case containing ≈ 100 words. Likewise, a large test case could be either a coverage trace for the branch coverage criterion of a big program or a large JUnit test method (or even a JUnit test suite). All combinations of the different test case and test suite sizes account for 84 different dimensions of testsets.

In order to collect data to answer RQ3, for each test case size (small, medium, large), we applied the investigated TCP approaches to the synthetic test suites, from the smallest (1K) to the largest (1M) one, and measured the preparation time and prioritization time taken by each approach.

All the experiments were performed on an Intel® Core™ i7-5960X with fixed 3.50 GHz CPU, 20M cache, 32GB RAM, running Linux openSUSE 13.2.

6 RESULTS

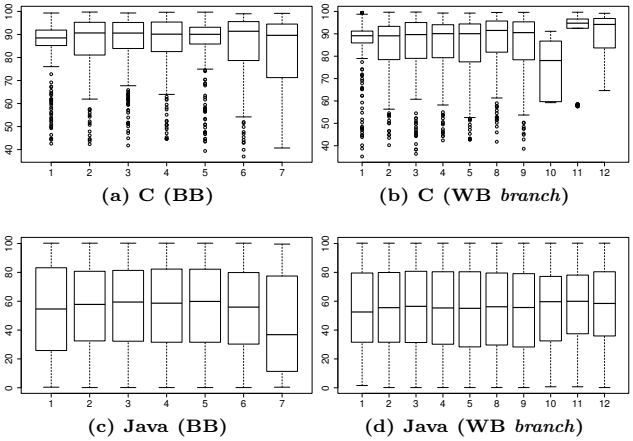
In this section we report and discuss the results. Note that with the aim of supporting the independent verification and replication, we make available the artifacts produced as part of this work.[7] The replication package includes, among others, the implementation of the algorithms, details on the hash function used,[8] input data, raw data used for the statistical analyses, and additional results.

[7] https://github.com/icse18-fast/FAST

[8] `xxhash` is the fastest non-cryptographic hash function available to the best of our knowledge.

6.1 RQ1: Effectiveness

The APFD results achieved by the considered BB and WB TCP approaches are displayed as box plots in Figure 2 (for the sake of space, for WB we display only the box plots for branch coverage) and in more detail in Table 2. Note that results for the same *FAST* techniques are reported among both WB and BB approaches. As explained in Section 4, for *FAST* techniques the only difference between WB or BB versions concerns the coding of test info: after that, the same algorithm can be identically applied.



The y-axis displays the average percentage of faults detected (APFD) and the numbers in the x-axis represent the prioritization approaches: *FAST*-pw(1), *FAST*-1(2), *FAST*-log(3), *FAST*-sqrt(4), *FAST*-all(5), STR(6), I-TSD(7), ART-D(8), ART-F(9), GT(10), GA(11), GA-S(12)

Figure 2: APFD for BB and WB TCP approaches.

Since the C and Java programs contain different types of faults (see Section 5.4), we analyze them separately to gather the effectiveness results.

As we could not assume our data to be normally distributed, we adopted a non-parametric statistical hypothesis test, the Kruskal-Wallis rank sum test, to assess at a significance level of 5% the null hypothesis that the differences in the APFD values for the different TCP approaches are not statistically significant. For the particular case of C programs when considering the BB approaches, the resulting *p-value* for the test was 0.2849, meaning that we cannot reject the null hypothesis, i.e., no significant difference in effectiveness was observed. For all the other cases, the resulting *p-values* were smaller than 2.2e-16 meaning that the observed differences in effectiveness are statistically significant at least at the 95% confidence level.

A significant Kruskal-Wallis test indicates that at least one TCP approach stochastically dominates another one, but does not identify the dominance relationship among pairs of techniques. To determine which TCP approaches are different, we performed pairwise comparisons after the Kruskal-Wallis test. The results are displayed in Table 2 inside the parenthesis. If two approaches have different letters, they are significantly different ($\alpha = 0.05$). If, on the other hand, they share the same letter(s), the difference between the means is not statistically significant. An approach can have more than one letter assigned to it. As an example, looking at the results for BB approaches and C subjects in Table 2, we can tell that *FAST*-pw (ab) is not different from *FAST*-log (a) and it is also not different from I-TSD (b), even though *FAST*-log (a) is different from I-TSD (b).

Among the WB approaches, the results confirm the ones reported in [15] regarding the effectiveness of GA and GA-S, as both appear always in the first group. Concerning *FAST*, the best results are achieved for function coverage: it goes in the first group, although with different members of the family for the C and Java subjects. Among the BB approaches, the *FAST* family consistently achieves the best group both for C and Java with the exception of the *FAST*-pw technique that for Java programs performed worse than STR. These results hint that *FAST* performs better in terms of effectiveness for coarse-grained criteria, which is the most natural target when considering big testsets.

> Answer 1: *APFD results vary across techniques and subject programs. A trend can be observed that FAST provides better effectiveness with more coarse-grained techniques (BB and function coverage).*

6.2 RQ2: Efficiency

To answer RQ2 we compared the investigated TCP approaches in terms of time required to perform the prioritization, both including and excluding the preparation time. We report the results in Table 3: Note that we do not discriminate between C and Java as efficiency results are only impacted by the size of test cases and coverage traces.

We applied the Kruskal-Wallis rank sum test to verify at a significance level of 5% the null hypothesis that the observed differences in the *total time* and in the *prioritization time* are not statistically significant. In both cases (total time and prioritization time only) regardless of the coverage criteria considered, the resulting *p-values* were always smaller than 2.2e-16, leading us to reject the null hypothesis.

To identify which TCP approach dominates the others in terms of efficiency, we proceeded with a pairwise comparison after the Kruskal-Wallis test. When considering BB TCP, the *FAST* family outperformed all the competitors. For WB TCP, the results vary between total or only prioritization time and among the coverage criteria. Overall, when considering total time, GT outperformed the competitor approaches for all the criteria. This result is aligned with our expectation as GT does not require any kind of preparation and simply performs a sorting of the test cases based on how many entities they cover. Nevertheless such approach was always followed by at least one member of the *FAST* family, which outperformed even the other greedy competitors (GA and GA-S). For the particular case of statement coverage GT is followed by the whole *FAST* family before the other TCP approaches.

Table 2: APFD results for the investigated BB and WB TCP approaches.

BB		
Approach	C	Java
FAST-pw	88.4/12.8 (ab)	54.6/29.8 (c)
FAST-1	90.6/12.6 (ab)	57.7/28.8 (a)
FAST-log	90.5/12.7 (a)	59.4/29.2 (a)
FAST-sqrt	90.1/12.8 (a)	58.6/29.3 (a)
FAST-all	90.0/13.1 (ab)	59.8/29.4 (a)
STR	91.3/13.3 (ab)	55.9/28.7 (b)
I-TSD	89.6/15.1 (b)	45.0/31.6 (d)

WB	Function		Statement		Branch	
Approach	C	Java	C	Java	C	Java
FAST-pw	89.4/12.0 (a)	51.4/29.6 (f)	88.3/12.0 (d)	52.9/30.4 (e)	89.2/12.8 (bc)	52.5/27.4 (de)
FAST-1	88.5/12.2 (cd)	56.5/28.1 (abc)	87.7/11.6 (e)	55.5/29.0 (bc)	89.1/12.5 (c)	55.5/27.7 (de)
FAST-log	89.2/12.5 (cd)	57.0/28.7 (ab)	89.2/11.0 (de)	54.1/29.2 (c)	89.7/13.7 (bc)	56.5/28.1 (cd)
FAST-sqrt	89.1/13.3 (cd)	56.5/27.9 (abc)	88.4/11.3 (de)	55.4/28.9 (b)	90.1/12.9 (c)	55.3/28.3 (de)
FAST-all	89.3/12.1 (bc)	54.1/28.3 (de)	88.5/12.3 (de)	52.9/28.8 (d)	90.1/13.4 (c)	55.1/29.2 (ef)
ART-D	92.1/12.5 (a)	54.2/29.5 (de)	91.8/14.3 (c)	56.5/29.2 (b)	91.5/12.2 (b)	56.1/29.2 (def)
ART-F	90.3/12.6 (ab)	55.0/29.3 (e)	90.6/12.4 (c)	54.6/28.7 (bc)	90.5/13.5 (bc)	55.6/29.2 (f)
GT	77.1/12.8 (e)	59.2/27.4 (bc)	77.3/13.3 (f)	59.2/27.4 (a)	78.1/13.3 (d)	59.7/28.3 (bc)
GA	88.4/09.8 (d)	59.2/28.0 (a)	96.6/15.6 (a)	51.4/26.3 (c)	94.8/15.2 (a)	59.9/26.8 (a)
GA-S	90.6/07.9 (a)	55.3/28.3 (cd)	94.4/11.1 (b)	58.3/27.5 (a)	94.2/09.3 (a)	58.5/27.3 (b)

Results are displayed in the format M/$\sigma(\delta)$, being M the median APFD, σ the standard deviation, and δ the group for the pairwise comparisons after the Kruskal-Wallis test. Different letters (δ) indicate significant differences between the approaches ($\alpha = 0.05$).

Table 3: Prioritization times (including and excluding preparation time) for the investigated TCP approaches.

BB		
Approach	Tot. Time	Prio. Time
FAST-pw	6.49/23.51 (e)	0.03/0.07 (e)
FAST-1	6.46/23.52 (d)	0.02/0.02 (d)
FAST-log	6.45/23.52 (b)	0.01/0.01 (b)
FAST-sqrt	6.46/23.52 (c)	0.02/0.01 (c)
FAST-all	6.45/23.52 (a)	0.01/0.00 (a)
STR	714.24/697.10 (f)	2.59/2.06 (f)
I-TSD	7402.71/5486.03 (g)	7402.71/5486.03 (g)

WB	Function		Statement		Branch	
Approach	Tot. Time	Prio. Time	Tot. Time	Prio. Time	Tot. Time	Prio. Time
FAST-pw	1.12/0.46 (g)	0.03/0.09 (g)	7.08/3.09 (f)	0.03/0.10 (f)	1.67/0.72 (f)	0.03/0.09 (g)
FAST-1	0.59/0.46 (f)	0.02/0.03 (e)	6.57/3.13 (e)	0.02/0.04 (d)	1.66/0.72 (e)	0.02/0.03 (e)
FAST-log	0.42/0.46 (e)	0.01/0.01 (d)	6.44/3.12 (d)	0.01/0.01 (c)	1.66/0.71 (d)	0.01/0.01 (c)
FAST-sqrt	0.41/0.46 (d)	0.01/0.01 (b)	6.45/3.10 (c)	0.01/0.00 (b)	1.66/0.70 (c)	0.01/0.01 (b)
FAST-all	0.39/0.46 (b)	0.01/0.00 (a)	6.42/3.10 (b)	0.01/0.00 (a)	1.65/0.68 (b)	0.01/0.00 (a)
ART-D	7.99/7.09 (h)	7.99/7.09 (i)	95.03/51.20 (g)	95.03/51.20 (i)	15.84/18.37 (g)	15.84/18.37 (i)
ART-F	30.28/17.68 (i)	30.28/17.68 (j)	224.89/139.00 (h)	224.89/139.00 (j)	38.52/56.36 (h)	38.52/56.36 (j)
GT	0.01/0.01 (a)	0.01/0.01 (c)	0.09/0.06 (a)	0.09/0.06 (e)	0.02/0.01 (a)	0.02/0.01 (d)
GA	0.51/0.56 (c)	0.51/0.56 (h)	11.19/5.20 (f)	11.19/5.20 (h)	1.26/1.28 (c)	1.26/1.28 (h)
GA-S	9.42/38.76 (j)	0.04/0.07 (f)	1906.63/4450.00 (i)	0.14/0.06 (g)	80.95/207.35 (i)	0.03/0.03 (f)

Results are displayed in the format M/$\sigma(\delta)$, being M the median time (total or prioritization), σ the standard deviation, and δ the group for the pairwise comparisons after the Kruskal-Wallis test. Different letters (δ) indicate significant differences between the approaches ($\alpha = 0.05$). For STR, the preparation time considers the time required for computing the pairwise similarity matrix; for GA-S, it refers to the time required to extract the spanning entities.

By taking into account exclusively the prioritization time, *FAST*-all and *FAST*-sqrt had the best performance for the three criteria, surpassing even GT. For statement coverage, the most demanding criteria in terms of time required, the best result achieved by a competitor was GT at fifth place.

A trend emerging from Table 3 is that the more demanding a criteria, the better the performance of the *FAST* approaches, especially by considering exclusively prioritization time. The reasons for this will be discussed in the following section while answering RQ3.

Answer 2: *The fastest members of FAST family outperform the BB approaches in terms of total time and all competitors when we consider only prioritization time.*

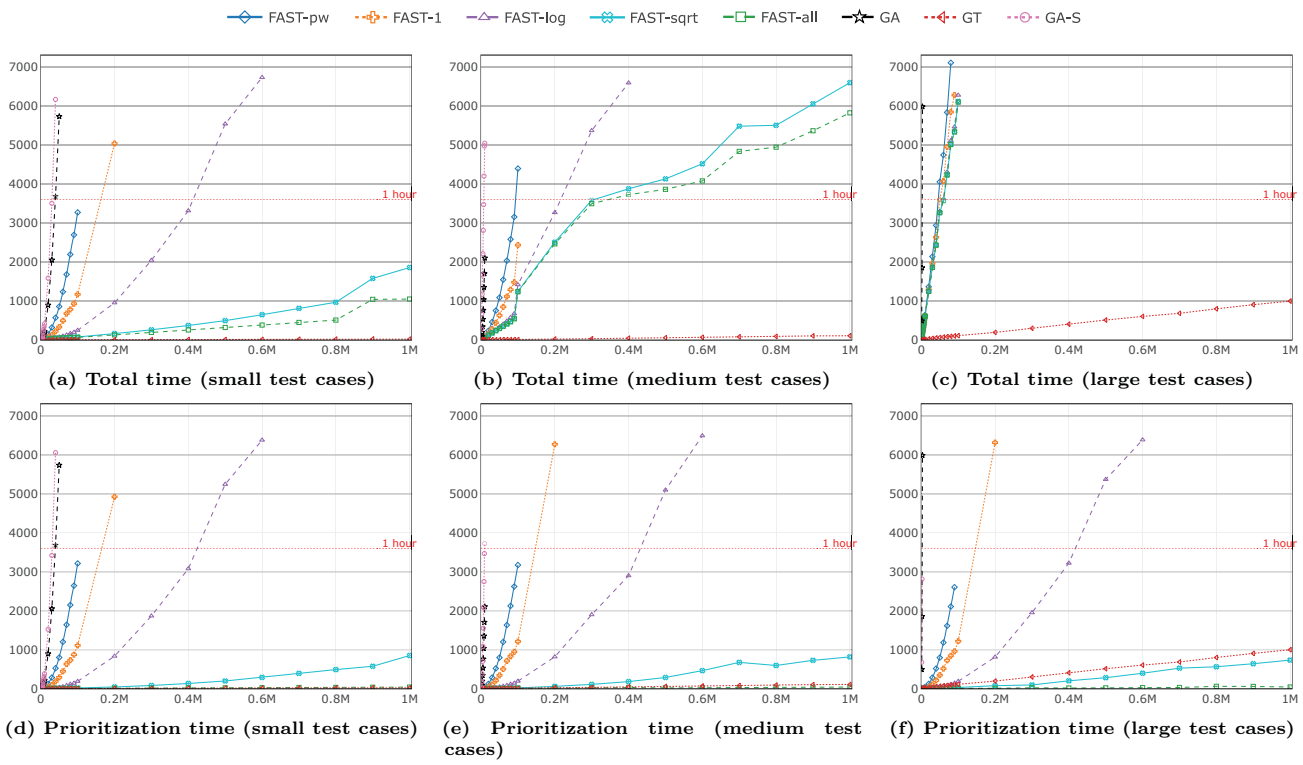
6.3 RQ3: Scalability

To answer RQ3 we assessed the TCP approaches with respect to the time required to prioritize synthetic test suites (representing both WB and BB), with sizes from 1K to 1M, and for three test case dimensions (small, medium, and large).

In Figure 3 we provide the line plots for the total time (Figures 3a to 3c) and for the prioritization time (Figures 3d to 3f) required by different TCP approaches. Although we have allowed all the approaches to proceed even further, for a clearer visualization the plots in Figure 3 report only the results for the testset sizes that could be prioritized within two hours. In fact, we considered that a TCP approach that can perform the task within two hours or less would allow developers to run the prioritization on their own machine in the timeframe of a meeting or lunch break. Hence we considered executions exceeding the two hours less interesting. We do not include in the plots the results for the STP approaches as they performed poorly and their results could not be easily visualized due to their long execution times. We summarize the results below in the format (*small*, *medium*, *large*), where for each test case size we report the largest testset size which could be completed within the two hours, while "Ø" indicates that the approach was not able to complete even the smallest testset: STR (4K, 2K, Ø), when considering total time, and (6K, 5K, 5K) when considering prioritization time; I-TSD (Ø, Ø, Ø); ART-D (6K, 2K, Ø); ART-F (6K, 1K, Ø).

The plots show that when considering total time GT outperforms all the competitors and can prioritize 1M size testset for the three sizes of test cases. The *FAST* family, on its turn, clearly outperformed GA and GA-S. Two members of the family, *FAST*-all and *FAST*-sqrt, prioritized the 1M set for small and medium test cases within the two hours limit, while *FAST*-log prioritized the 600K and 400K test suites for small and medium test cases respectively.

As expected *FAST* approaches are affected by the dimension of the test cases: The signature of each test case can be computed in linear time on the size of the test representation, thus for bigger test cases this step will take longer. However, once the preparation phase is completed, the only dimension that matters is the one of the test suite size (see the strong similarity of the line plots of the *FAST* approaches in Figures 3d to 3f). This answers our open question from the



(a) Total time (small test cases) (b) Total time (medium test cases) (c) Total time (large test cases)

(d) Prioritization time (small test cases) (e) Prioritization time (medium test cases) (f) Prioritization time (large test cases)

Figure 3: Total/Prioritization time (y-axis: seconds) to prioritize testsets of varying dimension (x-axis: number of test cases).

previous section: The relative performance of the *FAST* family improves, with respect to the competitors, as the criteria becomes more demanding (in terms of test case information used) because when we consider only the prioritization time the size of the test representation does not affect the *FAST* approaches in the same way that it affects the competitors.

Considering only prioritization time, GT still outperforms most *FAST* approaches but *FAST*-all that outruns GT for all the testsets for medium and large test cases. Besides that, *FAST*-sqrt outperforms GT for all the testsets with large test cases and also for some of the medium testsets.

> Answer 3: *Considering total time, all FAST techniques were second only to GT. If only prioritization time is counted, FAST-all surpass GT for medium and large testsets, and FAST-sqrt outruns it for some of the medium and all of the large testsets. Overall, except for FAST-pw, the bigger the testset, the better FAST's relative performance.*

6.4 On the Costs and Benefits of *FAST*

As it can be noticed from our results for RQ2 and RQ3, most of the cost associated with *FAST* lies in the preparation phase. BB input is first mapped into shingles in linear time. Then the preprocessing involves going through all the test cases, computing their signatures, and storing them for future use. In particular, the time complexity of computing a single signature is $\mathcal{O}(h)$, with h being the signature length, making the entire preprocessing phase costing $\mathcal{O}(hn)$, with a test suite of size n. The cost for computing Jaccard similarity of two sets is $\mathcal{O}(l)$, where l is the size of the biggest of the two sets, while estimating it through the minhash signatures costs $\mathcal{O}(h)$. LSH, on its turn, is able to create a candidate set in $\mathcal{O}(n)$.

Concerning space costs, the overall space required by *FAST* is $\mathcal{O}(shn)$, where s is the size of a hashed value. In contrast, the cost of storing a distance matrix for a generic STP approach with all the pairwise similarities (e.g., Ledru et al. [19]) is $\mathcal{O}(bn^2)$, where b is the size of the float representing a distance between two test cases.

In some settings, e.g., regression testing, the cost of updating existing information (e.g., signatures, distance matrix) is of interest. As discussed before, *FAST* can simply add information regarding a new test case by computing its signature. In contrast, the time required by a STP approach to update an existing distance matrix is that of comparing the newly introduced test case with all the existing ones: not only this operation is slower in terms of time complexity, but it also requires the entire test suite for the comparison.

It is also possible, even though we have not yet exploited this possibility, to use parallelization to reduce the running

time of *FAST*. In fact, the preprocessing phase, i.e., the computations of the signatures, of the buckets, and of the candidate set, are all parallelizable.

To decide which variant of *FAST* to use, one should consider the time available for the prioritization task. *FAST*-pw is the most precise in the ranking because it guarantees that the candidate test case that is the most dissimilar from the already-prioritized ones will be chosen as the next test, but it is also the most time-consuming. When the time required by *FAST*-pw cannot be afforded, other members of the family can be chosen. If the objective of the prioritization is to increase diversity, then the order of choice should be: *pw*, *one*, *log*, *sqrt*, and *all*.

FAST can provide greater benefits if it is adopted in an environment where the test case signatures can be reused as new test cases are added. A regression testing environment combined with the use of black-box representation for the test cases seems to be a perfect fit for *FAST*.

6.5 Threats to Validity

The results reported must be considered in light of potential threats to validity of the experiments. *Internal validity* concerns factors different from the treatment that could have affected the observed behavior [32]. One common threat is the selection of experimental subjects. In assessing effectiveness and efficiency we opted for benchmark programs that have been made available and used in similar studies, to favor replicability of results, but they could not be good representative of actual regression test scenarios. However, as the same subjects have been used for all approaches, possible threats apply to all. A similar argument can be done concerning the assessment of the time taken by the various techniques. To mitigate potential threats related to this point we have: (i) implemented all the algorithms in the same programming language; (ii) captured the process (CPU) time with checkpoints at the same places; (iii) performed all the experiments in the same machine. Concerning scalability assessment, our simulated scenarios could bias the results because the randomly generated test information could reproduce unusual situations. We have created multiple testsets (84) to try to reduce possible bias, but only many real world big testsets can prevent this threats. Concerning possible errors in the study implementation, all developed code has been rigorously inspected, all experiments have been repeated more than once, and all code and data are made available.

External validity concerns whether the results are generalizable beyond the experiment subjects. About effectiveness and efficiency our results may suffer from the limited number and the specific characteristics of the chosen subjects, although to mitigate this potential threat we covered two different languages. About scalability, our chosen modeling for testsets dimension could not be valid in real contexts. Besides, different parameter settings in the *FAST* algorithm might produce different results. This is not really a validity threat, though, it only implies that other *FAST* implementations can be done and have to be evaluated.

7 CONCLUSIONS AND FUTURE WORK

We have introduced the *FAST* family of approaches for fast test case prioritization. The simple yet powerful idea behind *FAST* is that of managing the big testsets of modern software development processes through the use of well-established techniques for big data. The results from our experiments both on real test subjects and on synthetic data support our idea. They showed that in comparison with BB techniques we can significantly improve prioritization efficiency, with no impact in effectiveness. The fastest members of *FAST* defeated all competitors, even greedy total, if we only count the prioritization time after preparation. More importantly, the results show that *FAST* can scale up to industrial demands: We prioritize one million test cases in less than 20 minutes.

Overall, the *FAST* family offers different effectiveness and efficiency results, allowing for a range of techniques spanning over BB and WB criteria and for fine-tuning the selection of the next test cases depending on size and time. The approach worked nicely on commonly used benchmarks, yet its most attractive target is obviously the realm of what we called big testsets, to which sophisticated or fine-grained techniques cannot be applied. Some recent works [8, 27] have addressed regression testing at "Google" scale by applying heuristics that use historical or dependency information. Indeed, a gap exists between academic (fine-grained) and practical (coarse) approaches. Our results hint that *FAST* can help reduce such gap and allows to use more refined criteria to test prioritization even for big testsets.

To the best of our knowledge, this is the first proposal to apply LSH techniques to address the growth of testing problem dimensions. We expanded here the idea for TCP, but we prospect a great potential for application of the same techniques to other testing tasks. There are several problems in testing that can leverage similarity computations. For example, an immediate follow up study will address test case selection that we did not include here for lack of space and time. We expect that variants of *FAST* can allow for efficiently selecting the most dissimilar test cases among thousands or millions of test cases. Another possibility is that of handling product lines, where huge numbers of product variations need to be tested and *FAST* could help in quickly finding the most diverse configurations among a large set.

Moreover, adopting multi-objective prioritization for *FAST* could be a good point for future work: The *FAST* algorithm could be adapted to consider other objective functions in addition to dissimilarity, although it is to be evaluated how this could affect the approach efficiency.

ACKNOWLEDGMENTS

This research has been partly funded by the European Project ElasTest in the H2020 Programme under GA No 731535. Breno Miranda wishes to thank the postdoctoral fellowship jointly sponsored by CAPES and FACEPE (APQ-0826-1.03/16; BCT-0204-1.03/17).

REFERENCES

[1] Md. J. Arafeen and Hyunsook Do. 2013. Test Case Prioritization Using Requirements-Based Clustering. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 312–321. DOI:http://dx.doi.org/10.1109/ICST.2013.12

[2] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 975–980. DOI:http://dx.doi.org/10.1145/2950290.2983954

[3] Emanuela Gadelha Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. 2011. On the use of a similarity function for test case selection in the context of model-based testing. *Softw. Test., Verif. Reliab.* 21, 2 (2011), 75–100. DOI: http://dx.doi.org/10.1002/stvr.413

[4] Cagatay Catal and Deepti Mishra. 2013. Test Case Prioritization: A Systematic Mapping Study. *Software Quality Control* 21, 3 (Sept. 2013), 445–478. DOI:http://dx.doi.org/10.1007/s11219-012-9181-z

[5] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of Test Case Diversity. *J. Syst. Softw.* 83, 1 (Jan. 2010), 60–66. DOI:http://dx.doi.org/10.1016/j.jss.2009.02.022

[6] Andrew R. Cohen and Paul M. B. Vitányi. 2015. Normalized Compression Distance of Multisets with Applications. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 8 (2015), 1602–1614. DOI: http://dx.doi.org/10.1109/TPAMI.2014.2375175

[7] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435.

[8] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 235–245. DOI:http://dx.doi.org/10.1145/2635868.2635910

[9] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Software Eng.* 28, 2 (2002), 159–182. DOI: http://dx.doi.org/10.1109/32.988497

[10] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. 2014. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal* 22, 2 (2014), 335–361. DOI:http://dx.doi.org/10.1007/s11219-013-9224-0

[11] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 223–233. DOI: http://dx.doi.org/10.1109/ICST.2016.33

[12] Dan Hao, Lu Zhang, and Hong Mei. 2016. Test-case Prioritization: Achievements and Challenges. *Front. Comput. Sci.* 10, 5 (Oct. 2016), 769–777. DOI:http://dx.doi.org/10.1007/s11704-016-6112-3

[13] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2011. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 327–336. DOI:http://dx.doi.org/10.1109/ICST.2011.12

[14] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving Scalable Model-based Testing Through Test Case Diversity. *ACM Trans. Softw. Eng. Methodol.* 22, 1, Article 6 (March 2013), 42 pages. DOI:http://dx.doi.org/10.1145/2430536.2430540

[15] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 523–534. DOI:http://dx.doi.org/10.1145/2884781.2884791

[16] Kim Herzig. 2016. Let's assume we had to pay for testing. Keynote at AST 2016. (2016). https://www.kim-herzig.de/2016/06/28/keynote-ast-2016/

[17] Bo Jiang, Zhenyu Zhang, Wing K. Chan, and T.H. Tse. 2009. Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 233–244.

[18] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. DOI: http://dx.doi.org/10.1145/2610384.2628055

[19] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing Test Cases with String Distances. *Automated Software Eng.* 19, 1 (March 2012), 65–95. DOI:http://dx.doi.org/10.1007/s10515-011-0093-0

[20] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA.

[21] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 559–570. DOI:http://dx.doi.org/10.1145/2950290.2950344

[22] Udi Manber. 1994. Finding similar files in a large file system. In *Usenix Winter*, Vol. 94. 1–10.

[23] Mika V. Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. 2015. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering* 20, 5 (01 Oct 2015), 1384–1425. DOI: http://dx.doi.org/10.1007/s10664-014-9338-4

[24] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H. Byers. 2011. Big data: The next frontier for innovation, competition, and productivity. (May 2011).

[25] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *2013 IEEE International Conference on Software Maintenance*. 540–543. DOI:http://dx.doi.org/10.1109/ICSM.2013.91

[26] Martina Marré and Antonia Bertolino. 2003. Using Spanning Sets for Coverage Testing. *IEEE Trans. Softw. Eng.* 29, 11 (Nov. 2003), 974–984. DOI:http://dx.doi.org/10.1109/TSE.2003.1245299

[27] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (SEIP'17)*. IEEE Press, Piscataway, NJ, USA, 233–242. DOI:http://dx.doi.org/10.1109/ICSE-SEIP.2017.16

[28] Tanzeem B. Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 58–68. DOI:http://dx.doi.org/10.1109/ISSRE.2015.7381799

[29] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proc. IEEE Int. Conf. on Software Maintenance, 1999.(ICSM'99)*. IEEE, 179–188.

[30] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. 2001. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on* 27, 10 (Oct 2001), 929–948. DOI:http://dx.doi.org/10.1109/32.962562

[31] Luay Tahat, Bogdan Korel, Mark Harman, and Hasan Ural. 2012. Regression test suite prioritization using system models. *Softw. Test., Verif. Reliab.* 22, 7 (2012), 481–506. DOI:http://dx.doi.org/10.1002/stvr.461

[32] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[33] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120. DOI:http://dx.doi.org/10.1002/stv.430

[34] Zhi Q. Zhou, Arnaldo Sinaga, and Willy Susilo. 2012. On the Fault-Detection Capabilities of Adaptive Random Test Case Prioritization: Case Studies with Large Test Suites. In *2012 45th Hawaii International Conference on System Sciences*. 5584–5593. DOI:http://dx.doi.org/10.1109/HICSS.2012.454