Scalable Approaches for Test Suite Reduction

Emilio Cruciani*, Breno Miranda†§, Roberto Verdecchia*‡, and Antonia Bertolino§
*Gran Sasso Science Institute | L'Aquila, Italy
†Federal University of Pernambuco | Recife, Brazil
‡Vrije Universiteit Amsterdam | Amsterdam, The Netherlands
§ISTI – Consiglio Nazionale delle Ricerche | Pisa, Italy
*emilio.cruciani@gssi.it | †bafm@cin.ufpe.br | ‡roberto.verdecchia@gssi.it | §antonia.bertolino@isti.cnr.it

***Abstract*—Test suite reduction approaches aim at decreasing software regression testing costs by selecting a representative subset from large-size test suites. Most existing techniques are too expensive for handling modern massive systems and moreover depend on artifacts, such as code coverage metrics or specification models, that are not commonly available at large scale. We present a family of novel very efficient approaches for similarity-based test suite reduction that apply algorithms borrowed from the big data domain together with smart heuristics for finding an evenly spread subset of test cases. The approaches are very general since they only use as input the test cases themselves (test source code or command line input). We evaluate four approaches in a version that selects a fixed budget B of test cases, and also in an adequate version that does the reduction guaranteeing some fixed coverage. The results show that the approaches yield a fault detection loss comparable to state-of-the-art techniques, while providing huge gains in terms of efficiency. When applied to a suite of more than 500K real world test cases, the most efficient of the four approaches could select B test cases (for varying B values) in less than 10 seconds.**

***Index Terms*—Clustering, Random projection, Similarity-based testing, Software testing, Test suite reduction.**

I. Introduction

In recent years testing has consistently been the most actively investigated topic of main software engineering conferences [6]. One prominent problem in software testing research can be abstracted as: Given a software S and an associated test suite T, how can we *efficiently* verify whether S passes on T, or -if not- identify the failing test cases? In this formulation, the emphasis is on the term "efficiently": Otherwise, the easy solution would be to just execute S on T. The research targets the common practical case that along the development process S needs to be repeatedly tested on T (see, e.g., [15]) and the plain *retest-all* strategy may be too costly considering the available resources (e.g., time).

To address the above question, in the last three decades many techniques have been proposed, which can be roughly divided in two groups: those that aim at reordering the test cases in T so that those more likely to fail are executed first (*test case prioritization*), and those that select a subset $T' \subseteq T$ that should ideally include the failing test cases, if any; the latter group of techniques is referred to as *test case selection* or *test suite reduction*,[1] depending on whether when choosing T' the changes made to S are considered (*modification-aware* regression testing) or not [34].

[1]Some authors use the term *minimization* in place of reduction when the not selected test cases are permanently removed from the test suite. Here, in line with [34], we will consider the two terms as interchangeable.

The proposed techniques have been evaluated and compared against each other using metrics relative to their fault detection effectiveness (e.g., the Average Percentage of Fault Detection of the reordered test suite, or the loss in faults detected by the reduced test suite T'); for test reduction and selection, also metrics relative to cost savings, e.g., the size or the execution time of T' are compared against those of the full suite T.

Another important factor that should be taken into account is the *cost of the technique itself*, both in terms of the computational effort and of the resources it requires. In other words, when evaluating whether investing on an automated approach aimed at reducing the cost of testing is worth, a complete cost-benefit analysis should also include the overheads implied by the approach [18].

However, not many of the proposed techniques have considered such implied costs. In 2004, Orso and coauthors already noticed that in regression testing efficiency and precision need to be traded off, because "precise techniques are generally too expensive to be used on large systems" [29]. Gligoric and coauthors [16] were the first to observe that the time consumed by any regression test technique should include an analysis phase, an execution phase, and a collection phase. They noticed that most authors only considered the savings in execution, a few measured also the analysis time, but no one before them measured also the last phase in which the information needed to apply the technique is collected. As pointed out by Elbaum and coauthors [15], at scale industries need approaches "that are relatively inexpensive and do not rely on code coverage information". In fact, for white-box techniques, the cost of collecting and saving up-to-date code coverage information should also be considered as part of the collection phase. This is confirmed by Herzig [19], who observes that code coverage is not for free as assumed in many works, and can cause up to 30% of time overhead!

In a recent work [28], we addressed the prioritization of very large test suites and showed that as the size of the test suite grows, most existing approaches become soon not applicable. That work proposed the *FAST* family of similarity-based test prioritization approaches that outperformed in efficiency and scalability all the compared approaches, except for the white-box greedy total approach. If we count the often ignored costs of measuring coverage, then *FAST* appears as the only *scalable* prioritization approach.

This paper introduces a family of scalable approaches for *test suite reduction*, called the *FAST-R* family. As in [28], *FAST-R* approaches are similarity-based and borrow techniques from the big data domain. However, with respect to [28] we apply here several new techniques that allow us to achieve even more efficient results. In *FAST* we used minhashing and locality-sensitive hashing algorithms [25]. *FAST-R* approaches adopt other efficient heuristics that are used to derive a set of B evenly spread points in a big data space. Precisely, one approach called *FAST++* applies the *k-means++* algorithm [4], while another one called *FAST-CS* uses a recent importance sampling algorithm to construct *coresets*, a clustering technique that scales up to massive datasets [5]. Moreover, we further enhance the scalability of both approaches by applying the *random projection* technique, that reduces the space dimensionality while preserving the pairwise distances of the points [21].

FAST++ and *FAST-CS* are extremely "practical" techniques in the sense required by all of [15], [16], [19], [28]: $i)$ thanks to the heuristics imported from the big data domain they are computationally very efficient; $ii)$ to reduce a test suite T they require no other information beyond T itself.

Based on the applied algorithms, the most natural scenario for *FAST++* and *FAST-CS* is that of finding a fixed budget B of test cases. This is referred in literature as *inadequate* test suite reduction. In the paper we also show how they can be adapted to perform *adequate* reduction, i.e., preserving coverage: We apply a filtering strategy and search for the most dissimilar test cases only among the ones that cover not yet covered elements. However we acknowledge that at large scale such adequate scenario is not realistic, because as already said coverage information cannot be assumed.

Although originally proposed for prioritization, we note that *FAST* approaches [28] could be easily adapted for test reduction: Instead of ordering the whole test suite, the algorithm is stopped when the budget B (or the desired coverage) is reached. Accordingly, we also include in *FAST-R* and evaluate the reduction version of *FAST-pw* and *FAST-all* (the most precise and the most efficient of the *FAST* family).

Summarizing, this paper proposes four test suite reduction approaches (two original ones and two adapted from [28]) that can be applied in two testing scenarios: under a fixed budget or for adequate test suite reduction.

We evaluated the four proposed approaches on commonly used C and Java benchmark programs against state-of-the-art reduction techniques, obtaining comparable results for effectiveness but notable improvements in efficiency. More interestingly, to validate our claims on the scalability of the approaches, we applied all four of them to the budget reduction of a test suite formed by more than 500K Java test cases collected from GitHub. At such large scale, not considering the preparation time, *FAST-pw* and *FAST++* required several hours to reduce the suite, e.g., $\sim$37 hours and $\sim$11 hours respectively for a 10% size, but *FAST-all* required 25 seconds and *FAST-CS* 9 seconds. Actually, *FAST-CS* looks as a real breakthrough as it *took less than 10 seconds for the reduction independently from the percentage*, and needed just 5 minutes for preparation in contrast to more than 3 hours taken by *FAST-all*.

The original contributions of this work include:

- The *FAST-R* family of scalable approaches for inadequate test suite reduction.
- A variant of all the approaches for adequate test suite reduction.
- A large-scale experimentation for evaluating the efficiency and effectiveness of the approaches in three scenarios, including a very large-scale test suite.
- An open-source automated framework along with all the data used for the experiments to support verifiability.

The paper is structured as follows. In the next section we survey related work. In Section III we present the approaches used. In Section IV and V, respectively, we present the evaluation methodology and the achieved results. Finally, Section VI draws conclusions and hints at future work.

II. Related work

This work is related to software regression testing and more specifically to test suite reduction techniques. The literature on software regression testing is huge: Two surveys [13], [35] provide a broad overview of prioritization, reduction (or minimization, used here in interchangeable way), and selection techniques. In particular, Yoo and Harman [35] reviewed the literature until 2009. Concerning reduction techniques, most of the surveyed works consists of heuristics over white-box coverage criteria, at various level of granularity (including statement, branch, function, or call-stack). Some approaches augment the coverage information with additional inputs by the tester (e.g., weighting coefficients or priority assignments), which may be costly or even biased [35]. Among the few "interesting exceptions" doing black-box reduction, they report some combinatorial, fault-based, and model-based techniques. More recently, Do [13] surveys further advances over [35]. In particular, for test suite reduction she reviews four more recent techniques, two of which are again coverage-based, and two ones introduce specific reduction techniques: one for GUI testing [3], and another for combinatorial interaction testing [7]. Note that both surveys [13], [35] include no work on similarity-based test suite reduction, as we propose here.

A recent systematic survey by Rehman and coauthors [23] focuses specifically on test suite reduction. The study surveyed the literature between 1990 and 2016, identifying a set of 113 relevant primary studies. Based on the adopted algorithms, they classify the approaches into: Greedy (mostly coverage-based), Clustering, and Search-based, plus hybrid combinations thereof. Our approach would fit in the Clustering group, in which out of the surveyed 113 studies they only find three works: one [8] using machine learning algorithms, and two [27], [33] using hierarchical clustering.

We take here a distance from most of the techniques surveyed in the above studies, since *FAST*-R is expressly motivated by considerations of scalability and practical applicability. In this perspective, our approach is more closely related to few recent works based on coarse-grained heuristics, clustering, and similarity.

In recent years some collaborative efforts between academic and industrial researchers start to appear that develop coarse-grained approaches trading precision with efficiency/scalability. Strictly speaking such works focus on test case selection and not test suite reduction, in that the choice of tests to execute is modification-aware. For example, Knauss and coauthors [24] use a statistical model that relates the changed code fragments (or churns) with test outcomes on Ericsson systems; considering a continuous integration development environment, Elbaum and coauthors [15] propose a strategy apt for Google testing process, which combines test case selection during pre-submit testing and test case prioritization in post-submit testing. Both selection and prioritization apply heuristics based on failure history and execution windows. By relying on very efficient algorithms, our *FAST*-R approaches can scale up to large industrial systems as the above works, while not sacrificing much of precision in deriving a representative subset of the test cases.

Our similarity-based approach is related to several techniques that exploit the diversity among test cases for guiding selection. Some techniques build on the notion of adaptive random testing (ART) [10] that, in a few words, first selects a random set of test cases and then filters them based on their distance from the already selected test cases. Several variants instantiations of ART have been proposed, including ART-D [20] and ART-F [36] that we use as competitors to *FAST*-R and that are further described in Section IV.

Some black-box approaches use similarity to reduce model-based test suites. Both test case reduction [2] and test case selection [9], [17] techniques have been proposed. These techniques have been conceived for industrial use: For example Hemmati and coauthors [17] pursue as a main goal a selection of test cases adjusted to the available testing budget. However, all such model-based approaches rely on the assumption that a formal model of program behavior, e.g., a LTS, is available. In contrast, *FAST*-R does not need to assume anything else beyond the test cases themselves.

A few works have proposed to leverage clustering of test cases as we do here, e.g., [11], [30]. However they calculate the similarity between two test cases based on code coverage information, which as said already could be too expensive at the testing scale we aim.

III. The approaches

Given a test suite T and some fixed budget $B \leq |T|$, the goal of similarity-based test suite reduction is to select B evenly spread test cases out of the test suite. If we model each test case as a point in some D-dimensional space, then the problem could be thought of as that of finding the central points of B clusters. The problem of clustering is *NP*-hard, but we are able to perform scalable similarity-based test suite

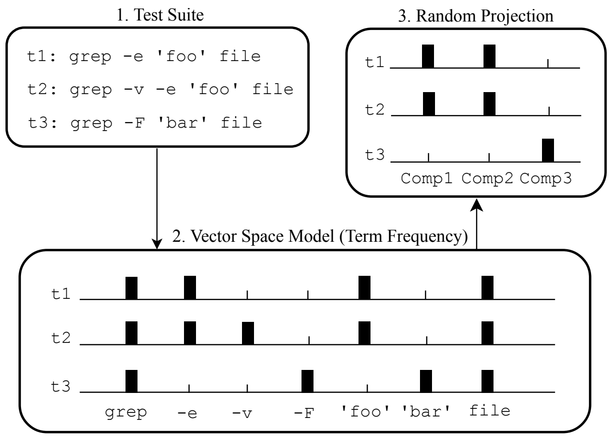


Fig. 1: Visual representation of *FAST-R* preparation phase.

reduction by borrowing a technique from the big data domain and using it in combination with some efficient heuristics.

We consider an Euclidean space, a metric space where the distance between any two points is expressed by the Euclidean distance – what one could think of as the straight line connecting them. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$ be two points; the *Euclidean distance* between them is defined as $d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{D}(\boldsymbol{x}_i - \boldsymbol{y}_i)^2}$.

In the *preparation phase* of our approaches (Fig. 1) we transform test cases into points in the Euclidean space via the *vector-space model*: The textual representation of each test case, e.g., test source code or command line input (Fig. 1.1), is mapped into an n-dimensional point where each dimension corresponds to a different term of the source code and n is equal to the total number of terms used in the whole test suite. The components are weighted according to *term-frequency* scheme, i.e., the weights are equal to the frequency of the corresponding terms (Fig. 1.2).

The computation of the Euclidean distance between any two n-dimensional points can be expensive when n is large. To overcome this problem we exploit a dimensionality reduction technique called *random projection*. Roughly speaking, random projection works because of Johnson-Lindenstrauss Lemma [21], which states that a set of points in a high-dimensional space can be projected into a much lower-dimensional space in a way that pairwise distances are nearly preserved. In particular we use *sparse random projection* [1], [26], an efficient implementation of the technique that is suitable for database applications (Fig. 1.3).

We model the clustering problem as a k-*means* problem, with $k = B$. Given n points in a metric space, the goal of k-means is to find a k-partition $P = \{P_1, \dots, P_k\}$ of the points that minimizes the sum of the squared Euclidean distances between each point to its closest center of one partition. Formally, the goal is to find $\arg\min_P \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in P_i} d(\boldsymbol{x}, \boldsymbol{\mu_i})^2$, where $\boldsymbol{\mu_i}$ is the center of the points belonging to partition P_i.

There exist efficient techniques that are able to find an approximate solution to k-means. One is *k-means++* [4],

Algorithm 1 *FAST++*

```
Input: Test Suite $T$; Budget $B$
Output: Reduced Test Suite $R$
1: $P \leftarrow \mathrm{RandomProjection}(T)$                ▷ Preparation phase
2: $s \leftarrow \mathrm{FirstSelection}(P)$
3: $R \leftarrow \mathrm{List}(s)$
4: $D \leftarrow \mathrm{Distance}()$          ▷ Squared distance to closest point in R
5: $D(s) \leftarrow 0$
6: while $(\mathrm{Size}(R) < B)$ do
7:     for all $t \in P$ do
8:         if $d\big(P(t), P(s)\big)^2 < D(t)$ then
9:             $D(t) \leftarrow d\big(P(t), P(s)\big)^2$       ▷ Squared Euclidean distance
10:    $s \leftarrow \mathrm{ProportionalSample}(P, D)$
11:    $R \leftarrow \mathrm{Append}(R, s)$
12:    $D(s) \leftarrow 0$
13: return $R$
```

which achieves an $\mathcal{O}(\log k)$ approximation ratio[2] in expectation and finds the centers of the clusters in k linear time iterations. The algorithm is the de facto standard technique for the initialization phase of k-means algorithms. After the initial centers are selected, standard k-means algorithms would iteratively compute the clusters. In our case, to be more efficient, we stop at this stage and use the k selected centers as the test cases of the reduced test suite. The reduction approach that exploits *k-means++* as greedy reduction strategy is called *FAST++* (Algorithm 1).

FAST++ starts by preprocessing the test suite T, mapping each test case into a vector according to the vector-space model and then lowering its dimensionality via random projection (Line 1). After the preparation phase, the reduction algorithm works only on the projected data P on which the greedy selection of *k-means++* is applied. First, pick the first point uniformly at random[3] (Line 2). Then, until B points have not been selected: $i)$ for each projected point $t \in P$, compute the squared distance $d(t, R)^2$ between t and its nearest center in R that has been already picked (Lines 7, 8, 9); this can be done incrementally by maintaining the minimum distance and computing only the distance with the last selected point (Lines 8, 9); $ii)$ pick next point s with probability proportional to its distance to R (Line 10).

Another possible approach to simplify the clustering problem is that of using *coresets*. Given a set of points S, a coreset is a small subset of S that well approximates the geometric features of S. One usually constructs a coreset first and then finds the centers of the clusters on it, reducing the complexity of the problem while still having theoretical guarantees on the solution. In our case, though, the size of the reduction grows linearly with the size of the test suite making this standard approach less efficient – the complexity of the problem would not lower much. Instead, exploiting a recent extremely efficient algorithm developed for massive datasets [5], we construct a coreset of size B and use it as reduced test suite. The algorithm is based on *importance sampling*: All points have nonzero probability of being sampled, but points that are far from the center of the dataset (potentially good centers for a clustering) are sampled with higher probability. We call the reduction approach that use this technique *FAST-CS* (Algorithm 2).

Algorithm 2 *FAST-CS*

```
Input: Test Suite $T$; Budget $B$
Output: Reduced Test Suite $R$
1: $P \leftarrow \mathrm{RandomProjection}(T)$                ▷ Preparation phase
2: $\mu \leftarrow \mathrm{Mean}(P)$
3: for all $t \in P$ do
4:     $Q(t) \leftarrow \frac{1}{2\,|T|} + \frac{d\big(P(t),\, \mu\big)^2}{\sum_{t' \in P} d\big(P(t'),\, \mu\big)^2}$     ▷ Importance sampling
5: $R \leftarrow \mathrm{ProportionalSampleWithoutReplacement}(P, Q, B)$
6: return $R$
```

FAST-CS starts with the preparation phase to compute the set of projected points P (Line 1). Then, it only requires two full passes on P: First it computes the mean of the data points (Line 2) and then it uses it to compute the importance sampling distribution (Lines 3, 4). The probability of each point to be sampled is a linear combination of the uniform distribution (first term in Line 4) and of the distribution which is proportional to the squared Euclidean distance between the data point and the mean of the data (second term in Line 4). Then B points are sampled out of P without replacement with probability proportional to their importance sampling probability (Line 5) and used as reduced test suite.

Both *FAST++* and *FAST-CS* have also been adapted to be *adequate*, i.e., to perform a reduction that guarantees some fixed coverage. [4] Getting coverage information of each test case as an extra input, both the proposed approaches are able to reduce the test suite such that some fixed coverage is achieved. This is possible thanks to a *filtering phase*. In *FAST++*, all test cases which would not add any extra coverage are filtered out after each selection and the next selection is carried out only among the remaining ones. As for *FAST-CS*, $\log|T|$ test cases are picked at each subsequent iteration and then importance sampling probabilities are recomputed setting to 0 the ones relative to test cases which are filtered out. Picking $\log|T|$ tests per iteration instead of just one makes the algorithm scale better to big test suites. Moreover, this choice does not increase the size of the reduced test suite since the selected test cases are still diverse among them and thus the chance of covering different parts of the software under test is still high. Finally, instead of stopping when the reduction reaches size B, both adequate approaches stop whenever the reduction achieves some fixed coverage.

As said, this work was inspired by the FAST family of test case prioritization approaches [28]: Roughly speaking, those approaches could be also used for the goal of test suite reduction by only picking the first B test cases of the prioritized test suite. To assess also their efficiency and effectiveness when applied to test suite reduction, we modified

[2] In a minimization problem, an α-approximation algorithm finds a solution which is not worse than α times the optimum.

[3] Note that this is to stick with *k-means++* algorithm, but any other criterion for the choice of the first test case is possible.

[4] The pseudocodes of adequate versions are not reported for lack of space, but they can be found online [12].

all the original algorithms to stop after B test cases are prioritized. Moreover we adapted them to be adequate as well, again using the same *filtering phase* introduced in *FAST++* and *FAST-CS*.

IV. Evaluation Methodology and Setup

We conducted some experiments to evaluate the effectiveness and the efficiency of the proposed approaches in different application scenarios. As a first scenario we considered the case in which test resources are limited and a tester can only run a small subset of test cases from an existing test suite: We call this the *budget scenario*, because we fix a priori a reduction percentage of test suite size. In this scenario we can apply the natural version of the proposed approaches. As a second case we considered *adequate scenario*, in which the code coverage measures of the whole test suite are preserved. To study this scenario, we applied the adequate version of the approaches. We also studied a third case, called the *large-scale scenario*, in which we apply the inadequate reduction on a very large test suite.

A. Research Questions

We address the following research questions (RQs):

RQ1: How **effective** are the proposed test suite reduction approaches in comparison with state-of-the-art techniques?

The goal of test suite reduction is to reduce the size of a test suite while maintaining its fault detection effectiveness. Thus the effectiveness of reduction approaches is commonly measured in terms of the Fault Detection Loss (FDL), and for adequate approaches also in terms of Test Suite Reduction (TSR). Consequently we articulate the above RQ1 into the two following subquestions:

RQ1.1: [FDL] What is the **fault detection loss** of the proposed approaches compared with that of state-of-the-art techniques?

To answer RQ1.1 we measure: FDL $= \frac{|F|-|F'|}{|F|}$, where F is the set of faults detected by T and F' is the set of faults detected by T'.

RQ1.2: [TSR] What is the **test suite reduction** achieved by the proposed approaches compared with that of state-of-the-art techniques?

To answer RQ1.2 we measure: TSR $= \frac{|T|-|T'|}{|T|}$.

We answer RQ1.1 in both budget and adequate scenarios, and RQ1.2 only in the adequate scenario.

To evaluate the efficiency we address the following RQ:

RQ2: How much **time** is taken by the proposed approaches to produce the reduced test suite?

We measure the time spent in preparation and in reduction. We answer RQ2 in all the three scenarios: In the budget and adequate scenarios we compare the time taken by the proposed approaches against state-of-the-art competitors; in the large-scale scenario we could only apply our proposed techniques, as all competitors approaches require coverage information that at such scales are not available.

B. Compared reduction approaches

We recall that the *FAST-R* family of proposed approaches consists of the newly devised *FAST++* and *FAST-CS* plus the modified reduction versions of *FAST-pw* and *FAST-all*, first introduced for prioritization [28].

The competitor approaches we consider are ART-D [20] and ART-F [36], which belong to the family of Adaptive Random Testing techniques [10]. In brief, they both work by first deriving a candidate set of test cases from those not yet selected that would increase coverage, and then selecting from within the candidate set the most distant test case from those already selected. The two techniques differ on the candidate set size (Dynamically changing in ART-D and Fixed in ART-F) and on the adopted distance metric (Jaccard and Mahattan, respectively). We selected these approaches because they also aim at obtaining an evenly spread set of test cases as in our approaches, and also because in the results reported in [28] they were among the best competitors to *FAST*. Differently from *FAST-R*, ART-D and ART-F use coverage measures.

Finally, we also applied the GA (Greedy Additional) approach [31], which for its simplicity and effectiveness is often considered as a baseline. GA selects the test case that covers the highest number of yet uncovered elements.

For all three competitors we consider three variants, applied to coverage of function, statement, and branch.

C. Experiment material

To evaluate the *budget scenario* and the *adequate scenario* we took 5 C and 5 Java programs as experimental subjects. The C programs (consisting of Flex v3, Grep v3, Gzip v1, Sed v6, and Make v1) were gathered from the Software Infrastructure Repository (SIR) [14]. For each of these programs subsequent versions are available, each containing a varying number of seeded faults. In our experiment we considered for each program the version containing the highest number of *difficult to reveal faults*, i.e., faults that are discovered by less than 50% of the test cases. This was done to avoid including in the experiment “anomalous” versions, e.g., versions in which most faults are revealed by the majority of the test cases or no faults are revealed at all. In total, the C subjects amounted to 52,757 LoC containing 49 faults, and were accompanied by a test suite comprising 2,938 test methods.

The 5 Java programs taken into account (namely Closure Compiler, Commons Lang, Commons Math, JfreeChart, and Joda-Time) were taken from the Defects4J database [22]. Such database provides a set of programs available in different versions, each containing a single real fault. For our experiment, we considered the first version of the programs. In total, the Java Subjects amounted to 320,990 LoC and were accompanied by a test suite comprising 1198 test classes.

To evaluate the *large-scale scenario*, we used a set of more than 500K real-world test cases gathered through the GitHub hosting-service. To efficiently collect a high number of heterogeneous test cases, we selected classes committed to the master branches of the available Java repositories, precisely commits adding a single class which adheres to common naming conventions for JUnit classes. In total through this process we collected 514,272 test cases, amounting to roughly 39 million LoC for a total size of 14 GB.

D. *Experiment procedure*

The experiment was performed on an AMD Opteron™ 6376 with 2.3GHz CPU, 16MB L2 cache, 64GB RAM, running Ubuntu 16.04.5 LTS. The procedure varied according to the scenario considered. More specifically:

1) Budget scenario: We fixed a set of budgets B for each experimental subject (both C and Java). The budgets considered ranged between 1% and 30% of the total test suite size of each subject with a step increase of 1%. While the *FAST-R* approaches only required the test suite for the reduction process, all competitors could take in input 3 different coverage types, namely function, statement, and branch. We therefore performed a single study for the *FAST-R* approaches and 3 for each of the competitors. We used each compared approach to reduce the test suite of the experimental subjects by considering all B budgets. The metrics considered were *fault detection loss*, *preparation time*, and *reduction time*. The measurements were repeated 50 times for each study given the stochastic nature of the approaches.

2) Adequate scenario: The *FAST-R* approaches require coverage information for the filtering phase as an extra input to have an adequate reduction. The competitor approaches instead require exclusively the coverage information. For this scenario we considered function, statement, and branch coverage. We used the compared approaches to reduce the test suite of each experimental subject (both C and Java) so to maintain the coverage prior of the reduction. We measured *fault detection loss*, *test suite reduction*, *preparation time*, and *reduction time*. The measurements were repeated 50 times for each study given the stochastic nature of the approaches.

3) Large-scale scenario: As for the *budget-scenario*, we considered a set of budgets B ranging from 1% to 30% of total test suite size of the subjects, with a step increase of 1%. In this setting we exclusively evaluated *FAST-R* approaches, as the other approaches require coverage information, which in this scenario is not available. To answer RQ2, we applied the approaches to the GitHub dataset for each possible reduction of B, and measured *preparation time* and *reduction time*.

V. Results

In this section we report and discuss the results. Note that with the aim of supporting independent verification and replication, we make available the artifacts produced as part of this work [12]. The replication package includes approaches, input data, statistical analyses, and additional results.

A. *The budget scenario*

1) Fault Detection Loss: The box plots of Figure 2 display the FDL of the compared approaches and more details are provided in Table I. The results are grouped by programming language because the C and Java programs investigated contain different types of faults (see Section IV-C). The approaches

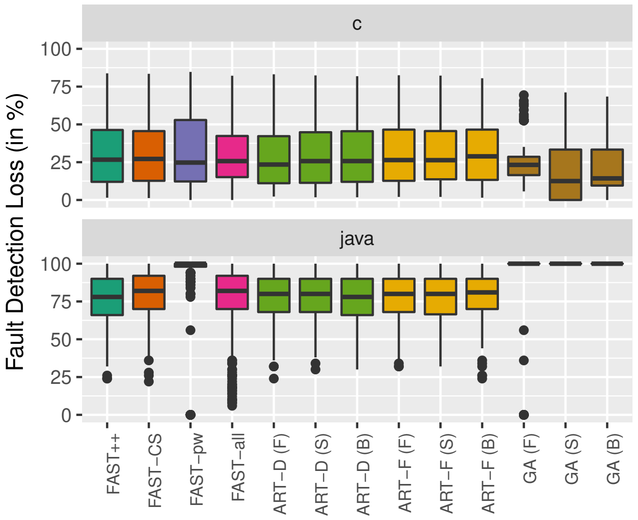


Fig. 2: FDL for the test suite reduction approaches (in %).

from the *FAST-R* family are applied in a black-box fashion while the competitors target different coverage criteria. For this reason we have three boxes for each competitor (the letter insides the parenthesis is the first letter of the targeted criterion: function, statement, or branch). For this metric, the lower the result, the better.

The visual assessment of the data can show us that, overall, the median fault detection loss is similar for all the approaches with two exceptions: while GA outperformed all the competitors for C (statement and branch), it performed very poorly and always missed the fault for the Java subjects (statement and branch); *FAST-pw*, on its turn, was comparable to most of the competitors when considering the C subjects but, similarly to GA, it performed poorly for Java.

After the visual inspection we proceeded with the statistical analysis of the data. We adopted a non-parametric statistical hypothesis test, the Kruskal-Wallis rank sum test, as our data could not be assumed to be normally distributed. We assessed at a significance level of 5% the null hypothesis that the differences in the FDL values are not statistically significant. For the particular case of C programs when targeting the function coverage criterion, the null hypothesis could not be rejected ($p\text{-}value = 0.8605$), i.e., no significant difference in fault detection loss was observed. For all the other cases the observed differences in FDL are statistically significant at least at the 95% confidence level.

Provided that significant differences were detected by the Kruskal-Wallis test we performed pairwise comparisons to determine which approaches are different.[5] The results are

[5]A significant Kruskal-Wallis test indicates that at least one reduction approach stochastically dominates one or multiple competitors, but does not identify the dominance relationship among pairs of techniques.

TABLE I: Fault detection loss in the *budget* scenario.

Approach	C			Java		
	Mdn	σ	δ	Mdn	σ	δ
FAST++	26.67	21.20	(b)	78.00	17.06	(a)
FAST-CS	27.14	21.00	(b)	82.00	15.41	(b)
FAST-pw	24.75	28.89	(b)	100.00	22.02	(c)
FAST-all	25.71	20.00	(b)	82.00	21.52	(ab)
ART-D (F)	23.44	20.53	(b)	80.00	17.29	(ab)
ART-F (F)	26.40	21.15	(b)	80.00	17.06	(ab)
GA (F)	23.21	15.11	(b)	100.00	26.71	(d)
ART-D (S)	25.71	21.04	(b)	80.00	16.91	(ab)
ART-F (S)	26.33	21.09	(b)	80.00	17.05	(ab)
GA (S)	12.50	22.40	(a)	100.00	0.00	(e)
ART-D (B)	25.75	21.10	(b)	78.00	16.96	(ab)
ART-F (B)	28.87	21.43	(b)	81.00	16.90	(ab)
GA (B)	14.29	20.17	(a)	100.00	0.00	(e)

Mdn is the median *fault detection loss*, σ is the standard deviation, and δ is the group for the pairwise comparisons after the Kruskal-Wallis test.

TABLE II: Reduction times for the *budget* scenario (including and excluding preparation time).

Approach	Total Time			Reduction Time		
	Mdn	σ	δ	Mdn	σ	δ
FAST++	0.37	0.21	(b)	0.01	0.05	(b)
FAST-CS	0.47	0.24	(c)	0.01	0.00	(a)
FAST-pw	7.94	12.36	(i)	0.10	0.21	(d)
FAST-all	7.82	12.40	(i)	0.02	0.03	(c)
ART-D (F)	0.18	0.56	(a)	0.18	0.56	(e)
ART-F (F)	0.46	2.19	(c)	0.46	2.19	(f)
GA (F)	0.20	0.27	(a)	0.20	0.27	(e)
ART-D (S)	2.50	8.50	(f)	2.50	8.50	(j)
ART-F (S)	4.87	31.89	(h)	4.87	31.89	(k)
GA (S)	3.26	5.02	(g)	3.26	5.02	(k)
ART-D (B)	0.62	4.80	(d)	0.62	4.80	(g)
ART-F (B)	1.41	19.77	(e)	1.41	19.77	(i)
GA (B)	0.77	2.79	(d)	0.77	2.79	(h)

Mdn is the median time (total or reduction), σ is the standard deviation, and δ is the group for the pairwise comparisons after the Kruskal-Wallis test.

displayed in Table I inside the parenthesis.[6] The statistical analysis confirmed the conclusions drawn from the visual inspection: with the exception of GA and *FAST-pw* that had varying performance depending on the programming language (and coverage criterion, when applicable), all the approaches investigated had overall comparable FDL.

2) Time: The results obtained by the approaches in terms of efficiency are displayed in Table II. It contains the total time (which includes preparation time) and the time for doing only the reduction itself. For this metric, we do not make any distinction between the programming languages (C or Java) because the efficiency of the approaches could be impacted only by the size of the test case representation adopted.

[6]If two approaches have different letters, they are significantly different (with $\alpha = 0.05$). If, on the other hand, they share the same letter, the difference between their ranks is not statistically significant. For example, looking at the results for the Java subjects in Table I, we can tell that *FAST-all*(ab) is not different from *FAST++*(a) and it is also not different from *FAST-CS*(b), even though *FAST++*(a) is different from *FAST-CS*(b). The approach (or group of approaches) that yields the best performance is assigned to the group (a).

TABLE III: Fault detection loss in the *adequate* scenario.

	Approach	C						Java		
		TSR			FDL			TSR		
		Mdn	σ	δ	Mdn	σ	δ	Mdn	σ	δ
Function	FAST++	97.76	1.92	(b)	33.33	22.88	(bc)	17.97	6.48	(b)
	FAST-CS	96.72	1.94	(c)	28.57	21.49	(bc)	16.39	6.12	(c)
	FAST-pw	97.01	1.99	(c)	33.33	27.16	(b)	20.28	5.76	(bc)
	FAST-all	57.51	36.53	(e)	0.00	19.13	(a)	2.76	5.11	(d)
	ART-D	30.64	32.20	(d)	0.00	20.72	(a)	0.00	0.62	(e)
	ART-F	21.49	34.70	(e)	0.00	19.22	(a)	0.00	0.40	(f)
	GA	98.21	1.67	(a)	28.57	20.64	(c)	30.33	7.57	(a)
Statement	FAST++	90.54	2.87	(b)	14.29	23.82	(d)	7.27	5.17	(b)
	FAST-CS	88.32	3.56	(c)	14.29	24.60	(d)	6.45	5.26	(c)
	FAST-pw	87.85	3.91	(d)	14.29	24.20	(d)	7.37	5.36	(b)
	FAST-all	28.51	32.27	(e)	0.00	20.97	(b)	0.00	1.33	(d)
	ART-D	6.17	8.26	(f)	0.00	9.28	(a)	0.00	0.41	(e)
	ART-F	3.44	6.57	(g)	0.00	6.62	(a)	0.00	0.31	(e)
	GA	91.62	2.74	(a)	12.50	25.46	(c)	12.30	4.67	(a)
Branch	FAST++	88.65	3.56	(b)	14.29	24.43	(d)	22.58	5.63	(b)
	FAST-CS	86.10	4.55	(c)	12.50	25.00	(d)	21.53	5.98	(c)
	FAST-pw	86.45	5.17	(d)	0.00	26.62	(c)	19.72	5.60	(d)
	FAST-all	15.97	22.96	(e)	0.00	10.41	(b)	6.76	5.18	(e)
	ART-D	4.92	6.49	(f)	0.00	8.75	(a)	0.26	0.79	(f)
	ART-F	2.43	4.68	(g)	0.00	3.49	(a)	0.00	0.62	(f)
	GA	90.27	3.52	(a)	14.29	23.78	(e)	35.94	5.45	(a)

Mdn is the median *fault detection loss*, σ is the standard deviation, and δ is the group for the pairwise comparisons after the Kruskal-Wallis test. Results for FDL are not displayed for Java because all the approaches were able to always reveal the existing fault with the reduced test suite.

If we consider only the reduction time, the *FAST-R* approaches outperformed all the competitors. *FAST++* and *FAST-CS* are much more efficient than *FAST-pw* and *FAST-all* during the preparation phase. Indeed, even if we consider total time, *FAST++* and *FAST-CS* are still very efficient: They would beat all the competitors with the exception of the ones targeting function coverage, which is a very coarse-grained criteria that allows the approaches to finish the reduction task after just a few iterations.

B. The adequate scenario

1) Test Suite Reduction and Fault Detection Loss: For the adequate scenario the *FAST-R* approaches are still applied in a black box fashion but they can use coverage information (which entities are covered by which test cases) in the reduction phase to filter out test cases that cannot contribute to increase coverage anymore. For this reason we report the results of our study grouped by programming language and by coverage criteria.

An ideal reduction approach should be capable of maximizing TSR while maintaing the same fault detection effectiveness of the original test suite. Thus, it is important to analyze TSR and FDL together. Figure 3 displays, for each approach, a side-by-side box plot for each metric (TSR in the left, and FDL in the right). For better readability, in Figure 3 we display $1 - TSR$, which represents the size (reported in %) of the reduced test suite, such that we can visually interpret the two metrics in the same direction: the lower the value, the better.

Additional results from the statistical analysis are reported in Table III. For our analysis we again performed the Kruskal-

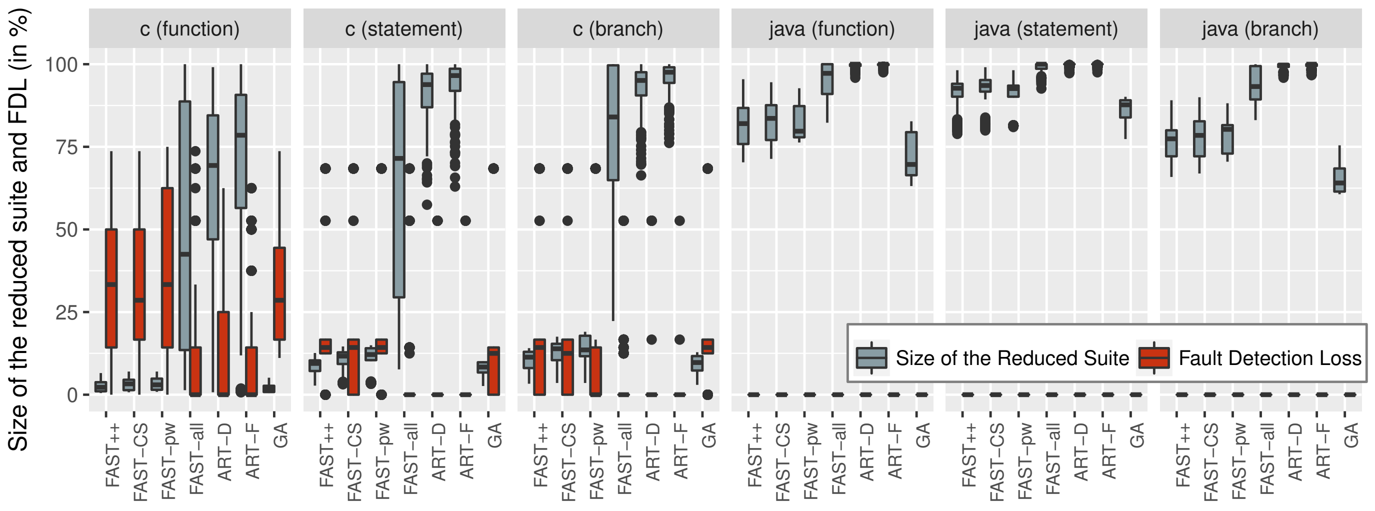


Fig. 3: Test Suite Reduction and Fault Detection Loss (in %).

Wallis rank sum test, followed by the pairwise multiple comparisons to identify the differences among the compared approaches. All the results reported in Table I are statistically significant at the 5% significance level.

Considering the results for C, two main groups seem to emerge: *FAST-all* and ART-* achieve the best results in terms of FDL, but at the price of having much bigger test suites (e.g., ART-F achieved a median reduction of 2.43% w.r.t. the original test suite considering branch); on the other hand, we have that the other members of the *FAST-R* family achieve reduction levels that are very similar to those of GA while remaining competitive with GA in terms of FDL. In fact, for some cases, the three members from the *FAST-R* family outperformed GA in terms of FDL (e.g., column δ of C-FDL-Branch).

For the particular case of Java, all the approaches were able to always reveal the existing fault with the reduced test suite. That explains why we do not have the FDL column for Java in Table III. GA always achieved the best results in terms of TSR, followed by the *FAST-R* family, then by the ART-based approaches. This result was somehow expected as GA aims at reaching the maximum achievable coverage with as few test cases as possible, while the *FAST-R* approaches aim at maximizing the diversity of the reduced test suite without having coverage as a target to be achieved.

2) Time: Table IV summarizes the results of the statistical analysis of our data after the Kruskal-Wallis test and the pairwise comparisons. Overall, the performance of the approaches remained stable when compared with the efficiency study for the budget scenario. With the exception of *total time* for *function*, where GA performed better, at least one of the *FAST-R* approaches achieved the best performance for all the other cases.

The ART-based approaches are, in general, not competitive: ART-D performs better than *FAST-pw* and *FAST-all* only when we consider *total time* for *function*. Then their performance degrades very quickly as we move from coarse- to fine-grained coverage criteria.

TABLE IV: Reduction times for the *adequate* scenario (including and excluding preparation time).

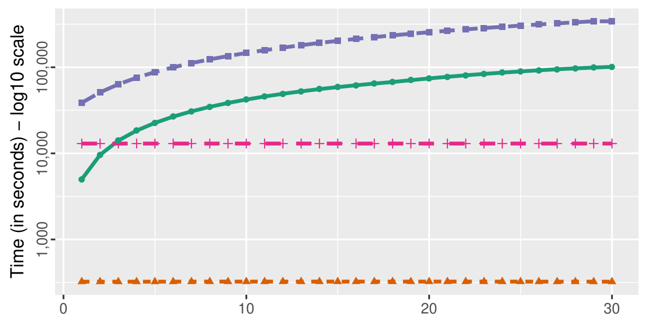
	Approach	Total Time			Reduction Time		
		Mdn	σ	δ	Mdn	σ	δ
Function	FAST++	0.48	0.40	(b)	0.04	0.24	(a)
	FAST-CS	0.51	0.44	(b)	0.07	0.22	(b)
	FAST-pw	8.41	16.10	(d)	0.10	0.31	(d)
	FAST-all	8.64	15.75	(d)	0.14	0.13	(bc)
	ART-D	3.76	7.86	(c)	3.76	7.86	(e)
	ART-F	18.45	38.35	(e)	18.45	38.35	(f)
	GA	0.10	0.51	(a)	0.10	0.51	(cd)
Statement	FAST++	0.93	2.39	(a)	0.50	2.22	(c)
	FAST-CS	1.01	1.86	(a)	0.58	1.78	(c)
	FAST-pw	8.15	12.48	(c)	0.44	1.89	(b)
	FAST-all	8.49	12.05	(c)	0.43	1.39	(a)
	ART-D	151.87	122.65	(d)	151.87	122.65	(e)
	ART-F	287.76	761.10	(e)	287.76	761.10	(f)
	GA	4.25	5.82	(b)	4.25	5.82	(d)
Branch	FAST++	0.65	0.56	(a)	0.26	0.39	(a)
	FAST-CS	0.75	0.58	(b)	0.37	0.38	(b)
	FAST-pw	9.05	15.79	(d)	0.37	0.54	(b)
	FAST-all	9.05	14.29	(d)	0.40	0.28	(a)
	ART-D	30.27	79.49	(e)	30.27	79.49	(d)
	ART-F	55.10	415.23	(f)	55.10	415.23	(e)
	GA	1.62	1.27	(c)	1.62	1.27	(c)

Mdn is the median time (total or reduction), σ is the standard deviation, and δ is the group for the pairwise comparisons after the Kruskal-Wallis test.

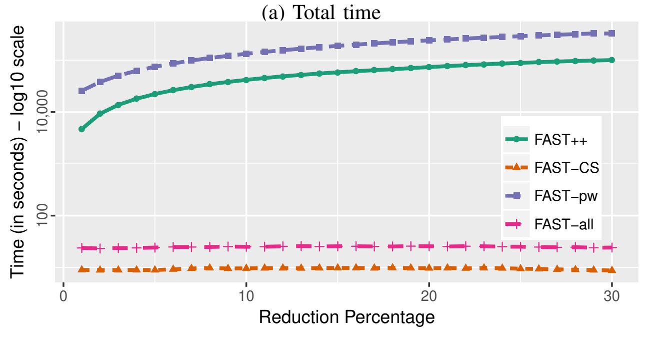
The very efficient preparation phase of *FAST++* and *FAST-CS* make them good candidates even if we had to consider the costs incurred by the preparation phase.

C. The large-scale scenario

The goal of this scenario is to provide empirical evidence to support our claim of scalability for the *FAST-R* approaches. The line plots in Figure 4 depict the time spent by the four *FAST-R* approaches to reduce the test suite formed by 500K+ test cases gathered from GitHub down to a budget B (with B varying from 1% to 30% of the full size). Precisely, we plot the total time in Figure 4.a and the reduction time in



(a) Total time



(b) Reduction time

Fig. 4: Time required to reduce 500K+ test cases to different reduction targets.

Figure 4.b. Note that in both plots we use on the vertical axis a logarithmic scale.

The slowest approach is *FAST-pw* and the fastest one is *FAST-CS*. While the reduction time of *FAST-CS* is about 9 seconds independently of the reduction percentage, the reduction time taken by *FAST-pw* varies with the reduction percentage between 25K seconds for 1% and 329K seconds (91+ hours) for 30% (see Figure 4.b), i.e., the time difference between the two approaches spans over 4 orders of magnitude. Considering the reduction time only, also *FAST-all* is quite efficient, as its time varied over the reduction percentage between 23 and 25 seconds.

The comparison of values between the two plots also evidences how the preparation phase of *FAST++* and *FAST-CS* is faster than the one of *FAST-pw* and *FAST-all*: for the former two the preparation time over the 500K+ GitHub test cases was ∼314 seconds, whereas for the latter two it grew up to ∼13K seconds, i.e., it took 40 times longer (see Table V).

Unfortunately in this scenario we could not measure FDL, but if the results of the budget scenario generalize, i.e., FDL is comparable to other state-of-the-art techniques, then we have here two approaches that in seconds can select a representative subset of dissimilar test cases from half million test cases. Considering also its lightweight requirements (Table V), *FAST-CS* is definitively the approach we would push to industrial applications.

For sake of completeness, we have also applied the *FAST-R* family to a synthetic set of coverage data for the large-scale test suite, and could also assess the scalability of the adequate versions of *FAST-R*. We do not show here the results for lack of space, but they are included in the replication package.

TABLE V: Time and space needed to compute and store prepared data by *FAST-R* in the *large-scale scenario*.

Algorithms	Input data	Preparation time	Prepared data
FAST++, *FAST-CS*	14.00 GB	314.43 s	22.05 MB
FAST-pw, *FAST-all*	14.00 GB	12979.29 s	83.85 MB

D. Answering the RQs

Trying to draw a concise summary of the results in the three scenarios, we can conclude that:

RQ1.1: The *FAST-R* family shows FDL values that are statistically comparable to those of competitor approaches (see Tables I and III).

RQ1.2: In terms of TSR, *FAST-R* approaches (excluding *FAST-all*) resulted the first for C, and second only to GA for Java (see Figure 3).

RQ2: Time-wise, *FAST-R* beats all competitors in reduction time, and remains competitive even including preparation time, resulting second only to ART-D and GA on function coverage (see Tables II and IV). At large-scale, *FAST-CS* achieves outstanding results (Figure 4), being able to prioritize a 500K+ test suite in 5 minutes, including the preparation time.

E. Costs and benefits

1) Preparation phase: The preparation phase of *FAST++* and *FAST-CS* is crucial for scalability; the random projection technique has a cost of $\mathcal{O}(nDd)$, where n is the number of points we are projecting, D is the actual dimensionality of the data, and d the dimensionality of the projected data. In this work, the test cases are projected into a space with $d = 10$ dimensions; a higher dimensionality would better preserve the original pairwise distances, but this choice seems to provide a good trade-off between effectiveness and efficiency [28]. As can be seen in Table V, *FAST++* and *FAST-CS* require much smaller preparation times, and also less space to store the prepared data on disk. This is because even if the dimensionality of the projected data (*FAST++*, *FAST-CS*) is the same as the length of the minhash signatures (*FAST-pw*, *FAST-all*), the sparsity of the random projection makes many of the components of the projected data null. This results in space savings through sparse representation of the data, which is not possible for the minhash signatures. Note that also the new preparation phase is suitable to scenarios in which additions/edits are made to the test suite, e.g., regression testing. In fact it is enough to update the random projection matrix to handle the increased dimensionality of the space and to process only new/modified test cases (the old are not affected by the updated matrix).

2) Budget version: With a fixed budget B, the reduction time complexity is $\mathcal{O}(nBd)$ for *FAST++* and $\mathcal{O}(nd)$ for *FAST-CS*: The former performs B iterations, each of which computes n distances in $\mathcal{O}(nd)$; the latter instead, just needs two full iterations on the data, with a cost of $\mathcal{O}(nd)$, to compute the distances between the mean and each point.

3) Adequate version: With a worst case analysis, the time complexity of *FAST++* would increase, since an adversarial input could make it select the entire test suite in n iterations other than canceling out the advantages of the filtering phase; in practice, though, filtering should lower problem complexity, iteration by iteration, making the algorithm much faster on non-adversarial input. Regarding *FAST-CS*, instead, the complexity increases to $\mathcal{O}(\frac{nRd}{\log n})$, where R is the size of the reduction. In fact, another pass of the data is needed after each filtering phase to recompute the importance sampling probability. The $1/\log n$ factor is due to the selection of $\log n$ points per iteration. Picking $\log n$ test cases per iteration, instead of just one, helps to mitigate the higher complexity w.r.t. the budget scenario and make the algorithm scale better on big test suites.

F. Threats to validity

Despite our best efforts, the presented results could suffer from validity threats. Following the classification in [32] we consider four aspects.

1) Construct validity: if the experiments we set are appropriate to answer the RQs. To answer RQ1 we measured FDL and TSR, which are *de facto* standard metrics [34]. This should nullify potential threats in answering RQ1. Concerning RQ2, we measured preparation and reduction time: A potential threat is that other factors than time could prevent *FAST-R* to scale up. To mitigate this risk, we used real world test suites in all scenarios.

2) Internal validity: if the observed results are actually due to the "treatment" and not to other factors. A common internal threat is the accuracy of measures that can be affected by random factors: To mitigate this threat we replicated all observations 50 times.

3) External validity: whether and to what extent the observations can be generalized. The experiments we performed are in line with other studies in the literature. We have observed the proposed techniques over C and Java subjects, of varying dimensions. Perhaps the programs and test sets from SIR and Defects4J may not well represent actual regression testing scenarios. However, we opted for such subjects because: i) we could not find other subjects providing faults information; ii) they are used in many other studies. Notwithstanding, from current observations we cannot draw general conclusions, and more experimentation is needed.

4) Reliability: whether and to what extent the observations can be reproduced by other researchers. To ensure reproducibility, as said we make available all data and settings related information.

VI. Conclusions and Future Work

This paper addressed the problem of reducing the size of test suites during regression testing. Our focus is on very large scale test suites; existing coverage-based or model-based techniques cannot be used in such scenario. We rather propose to apply similarity-based selection, which intuitively picks the test cases so that they are the most distant from each other, according to some distance metric.

To efficiently find a subset of B test cases we introduced two novel techniques, *FAST++* and *FAST-CS*, and adapted to the problem of reduction the *FAST-pw* and *FAST-all* techniques, previously proposed in [28] for test prioritization. All four approaches import smart heuristics from the big data domain and provide different degrees of precision and efficiency.

We evaluated the effectiveness and efficiency of the *FAST-R* family on the commonly used SIR and Defects4J benchmark programs. The effectiveness is measured through the metric of Fault Detection Loss. Moreover, even though we would not need to use coverage information, we have implemented an adequate variant of the four techniques on which we measured Test Suite Reduction. The results from both budget and adequate scenarios are that FDL and TSR remain both comparable with state-of-the-art reduction techniques (namely GA, ART-D, and ART-F). On the other hand, the efficiency of the proposed approaches, in terms of the reduction time, is better than all the compared approaches but GA applied to function coverage, already for the relatively small scale benchmarks. We also applied the *FAST-R* family to a much larger test suite (500K+ test cases) and here we got impressing results as presented in Section V.

For the future, we have made several plans to extend this work. We would like to challenge our *FAST-R* family on a real large scale testing scenario. Although in our studies we measured the time employed in deriving the reduced test suite on a real set of half million test cases, we could not also assess the effectiveness at such scale because we did not have fault data. Moreover, we would also like to apply the techniques in real development environment, to consider other possible process factors that may impact scalability. In particular, we would like to embed the techniques into a continuous development practice, where other criteria could be also considered when picking test cases.

In this work, we did not consider which files are modified when selecting test cases, as in (modification-aware) regression test selection. We could extend the proposed reduction approaches by filtering out the test cases that do not impact on modified files. Or, other smarter, more efficient heuristics could be conceived. This is an extension very relevant also in light of the conclusions in [34] that selection techniques achieve higher TSR than reduction ones, and are safer.

As already said in [28] in modern complex and distributed software systems, test suites deserve to be managed in the same way as big data, and we do hope that our novel approaches to test suite reduction contribute to further advance the fields towards scaling up test automation.

Acknowledgment

This research has been partly funded by the H2020 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] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *Journal of computer and System Sciences*, vol. 66, no. 4, pp. 671–687, 2003.

[2] P. D. L. M. Ana Emília V. B. Coutinho, Emanuela G. Cartaxo, "Test suite reduction based on similarity of test cases," in *7st Brazilian workshop on systematic and automated software testingâĂŤCBSoft*, 2013.

[3] S. Arlt, A. Podelski, and M. Wehrle, "Reducing gui test suites via program slicing," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 270–281. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610391

[4] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM Symposium On Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[5] O. Bachem, M. Lucic, and A. Krause, "Scalable k-means clustering via lightweight coresets," in *International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018, p. 11.

[6] A. Bertolino, A. Calabrò, F. Lonetti, E. Marchetti, and B. Miranda, "A categorization scheme for software engineering conference papers and its application," *Journal of Systems and Software*, vol. 137, pp. 114 – 129, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217302844

[7] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Interaction-based test-suite minimization," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 182–191. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486813

[8] L. C. Briand, Y. Labiche, Z. Bawar, and N. T. Spido, "Using machine learning to refine category-partition test specifications and test suites," *Inf. Softw. Technol.*, vol. 51, no. 11, pp. 1551–1564, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.06.006

[9] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing," *Softw. Test., Verif. Reliab.*, vol. 21, no. 2, pp. 75–100, 2011. [Online]. Available: https://doi.org/10.1002/stvr.413

[10] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2009.02.022

[11] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza, "Clustering support for inadequate test suite reduction," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 95–105.

[12] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction (online material)," Jan 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2550079

[13] H. Do, "Recent advances in regression testing techniques," in *Advances in Computers*. Elsevier, 2016, vol. 103, pp. 53–77.

[14] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[15] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635910

[16] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 211–222.

[17] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 6:1–6:42, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2430536.2430540

[18] K. Herzig, "Let's assume we had to pay for testing," Keynote at AST 2016, 2016. [Online]. Available: https://www.kim-herzig.de/2016/06/28/keynote-ast-2016/

[19] ——, "Testing and continuous integration at scale: Limits, costs, and expectations," in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ser. SBST '18. New York, NY, USA: ACM, 2018, pp. 38–38. [Online]. Available: http://doi.acm.org/10.1145/3194718.3194731

[20] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 233–244.

[21] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a hilbert space," *Contemporary mathematics*, vol. 26, no. 189-206, p. 1, 1984.

[22] R. Just, D. Jalali, and M. D. Ernst, "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*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[23] S. U. R. Khan, S. P. Lee, N. Javaid, and W. Abdul, "A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines," *IEEE Access*, vol. 6, pp. 11 816–11 841, 2018.

[24] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 19–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820678.2820684

[25] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge university press, 2014.

[26] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 287–296.

[27] Y. Liu, K. Wang, W. Wei, B. Zhang, and H. Zhong, "User-session-based test cases optimization method based on agglutinate hierarchy clustering," in *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, Oct 2011, pp. 413–418.

[28] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 222–232. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180210

[29] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 241–251.

[30] Y. Pang, X. Xue, and A. S. Namin, "Identifying effective test cases through k-means clustering for enhancing regression testing," in *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, vol. 2. IEEE, 2013, pp. 78–83.

[31] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, Oct 2001.

[32] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

[33] S. Sampath, S. Sprenkle, E. Gibson, L. L. Pollock, and A. L. Souter, "Analyzing clusters of web application user sessions," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083255

[34] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786878

[35] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430

[36] Z. Q. Zhou, A. Sinaga, and W. Susilo, "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*, Jan 2012, pp. 5584–5593.