Comparing and Combining File-based Selection and Similarity-based Prioritization towards Regression Test Orchestration

Renan Greca, Breno Miranda, Milos Gligoric, Antonia Bertolino
renan.greca@gssi.it,bafm@cin.ufpe.br,gligoric@utexas.edu,antonia.bertolino@isti.cnr.it
Gran Sasso Science Institute, Federal University of Pernambuco, The University of Texas at Austin, ISTI-CNR
Italy, Brazil, USA, Italy

ABSTRACT

Test case selection (TCS) and test case prioritization (TCP) techniques can reduce time to detect the first test failure. Although these techniques have been extensively studied in combination and isolation, they have not been compared one against the other. In this paper, we perform an empirical study directly comparing TCS and TCP approaches, represented by the tools Ekstazi and FAST, respectively. Furthermore, we develop the first combination, named Fastazi, of file-based TCS and similarity-based TCP and evaluate its benefit and cost against each individual technique. We performed our experiments using 12 Java-based open-source projects. Our results show that, in the median case, the combined approach detects the first failure nearly two times faster than either Ekstazi alone (with random test ordering) or FAST alone (without TCS). Statistical analysis shows that the effectiveness of Fastazi is higher than that of Ekstazi, which in turn is higher than that of FAST. On the other hand, FAST adds the least overhead to testing time, while the difference between the additional time needed by Ekstazi and Fastazi is negligible. Fastazi can also improve failure detection in scenarios where the time available for testing is restricted.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

regression testing, test case selection, test case prioritization, test orchestration, Fastazi

ACM Reference Format:
Renan Greca, Breno Miranda, Milos Gligoric, Antonia Bertolino. 2022. Comparing and Combining File-based Selection and Similarity-based Prioritization towards Regression Test Orchestration. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524481.3527223

1 INTRODUCTION

Software regression testing is actively researched [3, 27, 35], and many techniques have been proposed, including test case selection [17] and prioritization [19].

Both selection and prioritization techniques aim at detecting regression failures, but they follow different strategies. In test case selection (TCS), when a new software version is released, a subset of test cases is selected from the available test suite aiming at exercising all the latest code changes. TCS is proposed as an alternative to a *retest-all* (i.e., running all tests at each version) strategy that is not sustainable in many practical cases. On the other hand, in test case prioritization (TCP) the test suite is re-ordered, aiming at executing first those test cases that are more likely to fail. As, of course, we cannot know in advance which test cases discover which failures, different TCP criteria have been proposed, such as code coverage.

Many TCS and TCP approaches have been proposed [14, 21, 24, 31]. Our research goal here is not that of inventing yet another approach, but rather to understand if and how TCS and TCP should be used in combination, i.e.: when a new software version is released, is it more convenient to apply a TCS approach or instead a TCP one? Intuitively, a combination of both techniques would provide the most benefit, but what are the resulting challenges and drawbacks of this approach? Notwithstanding the vast literature on regression testing, such type of questions remain largely unanswered.

To address such concerns, we focus here on regression testing techniques that have been conceived for practical relevance and scalability. Specifically, as a representative TCS approach we adopt Ekstazi [9] while for TCP we use FAST [25]: we selected these two approaches due to their cost-effectiveness and simplicity of application, as well as their availability; finally, also for convenience as the authors of this paper include authors of both tools.

Concerning TCS, in an empirical study conducted in 2014 [10], the authors observed that many techniques were not adopted in practice and developers mostly continued to perform manual selection of test cases. Motivated by this study, Gligoric et al. [9] proposed Ekstazi, a lightweight TCS technique that leverages file dependencies. Besides the original paper on Ekstazi, several follow-up studies showed the benefit of file-based selection over other approaches [21, 37].

Concerning TCP, in a recent study Miranda et al. [25] showed that many existing techniques do not scale-up to large test suites. They hence proposed the FAST approach that applies Locality-Sensitive Hashing (LSH) techniques [22] for similarity-based prioritization. In the original work, the authors assess FAST against several competing TCP techniques, showing that it gives comparable effectiveness but with higher efficiency.

This work stems from the simple yet powerful idea of comparing these two approaches—TCS by Ekstazi and TCP by FAST—and possibly taking the advantages of each while overcoming their potential shortcomings. We make the following two observations: *i)* Ekstazi comes with no notion of test case priority: it assumes that all the selected test cases are run and makes no distinction about whether a failure is found by the first or the last executed test case; *ii)* FAST reorders tests with the goal to detect failures early, but does not consider recent code changes, whereas we know from practice that these are related with failures, e.g., [6, 20].

By combining Ekstazi and FAST, we aim at developing a *practical* and *effective* approach to regression testing that we call Fastazi. Fastazi is meant to be practical because it combines two scalable techniques, and effective because it overcomes the above shortcomings of each. In particular, this combined approach aims to decrease *developer feedback time*, which is the time it takes for a developer to receive a test failure notification once testing begins.

Clearly Fastazi is one instance within a plethora of possible combinations of many existing TCS and TCP approaches, and further studies should be conducted to evaluate different combinations. Indeed, following the case made by Harman [12], research in combining multiple criteria in the context of one regression technique is very active, e.g., [7, 8]. Much less attention has been devoted so far to using multiple criteria while combining different regression techniques, which we call *regression test orchestration*. Di Nardo et al. [5] applied and assessed minimization, selection and prioritization techniques on a single industrial case study, but only considering coverage-based criteria; Silva et al. [30] proposed to combine prioritization and selection based on function criticality (assessed manually); Najafi et al. [26] evaluated selection and prioritization based on test execution history on a large industrial system; Shi et al. [29] combined and evaluated test reduction (based on coverage) and selection (based on changes). Fastazi is the first regression test orchestration approach that combines file-based TCS with similarity-based TCP.

We compared Ekstazi, FAST and their orchestration through Fastazi using a set of 12 projects (from the Defects4J repository [15]). Our results shows that for most subjects, executing a change-aware selection of test cases (in random ordering) detects the first failure faster than executing the whole prioritized suite (based on similarity). However, we also observed that adding FAST ordering on top of Ekstazi selection further improves effectiveness at negligible additional cost.

In summary, our contributions include:

- an empirical study comparing TCS against TCP, and their orchestration against each technique alone;
- the novel Fastazi approach to regression testing that combines filed-based TCS and similarity-based TCP;
- a replication package[1] including Fastazi implementation and all data from the study.

For practitioners our results signify not only a further confirmation of change-aware selection validity, but also the convenience of executing the selected test cases in prioritized order based on their similarity. In fact, using state-of-art scalable techniques as FAST over the selected test subset can help detect failures faster at virtually no cost. For researchers, this paper signifies the importance of studying regression techniques as an orchestration rather than individually, and opens up the floor for many potential experiments in which various TCS techniques are compared against, or combined with, various TCP techniques.

In the next section we provide a short summary of the TCS and TCP approaches that we compare and combine, while in Section 3 we present Fastazi. The study methodology is described in Section 4 and the results are discussed in Section 5. In Section 6 we overview related work, and in Section 7 we draw brief conclusions.

2 BACKGROUND

2.1 Test Case Selection (TCS)

In regression testing, not all tests are relevant to a particular code change: if only a small part of one file was updated, it is unlikely that the entire project would be affected and the full regression test suite would have to be run. TCS addresses the challenge of selecting a subset of tests that is representative of the entire suite in a given situation [28, 35]. In other words, given a test suite T, TCS can be described as a function $S(T)$ that selects a subset of T to be used for testing the current version of the system under test. We say that a TCS technique is *safe* if it guarantees that all tests whose outcome may be affected by a change are included in the selected subset [28]. Common approaches for TCS are change-based, history-based, and model-based [17].

Ekstazi [9] is a *change-based* and *coarse-grained* approach to TCS. It works by collecting test case dependencies (i.e., set of used classes by each test case) during an initial run of the entire test suite, then by selecting the test cases based on the changes applied to those dependencies from one version of the software to another. In doing this Ekstazi applies a *file-level granularity*: any code changed within a file that is related to a test case will result in that test being selected. To compare two versions of a file, Ekstazi uses cyclic redundancy check (CRC). For example, consider a test t that invokes a function a. If a change is made to another function b located in the same file as a, t will be selected (as the CRC of the file changed).

The result of this approach is an over-approximation of the subset of selected tests. Although Ekstazi selects, on average, more tests than fine-grained TCS solutions (e.g., those that track dependencies on methods), the authors demonstrated that the actual selection time is much faster than the alternatives. Consequently, the total end-to-end time (i.e., time to select tests + time to execute selected tests) tends to be lower, even if more tests are selected.

We chose Ekstazi for our study for its efficiency and ease of use: Ekstazi is publicly available as a plug-in for various Java build systems. Furthermore, aiming eventually at an orchestration of TCS plus TCP with the objective of reducing feedback time, we considered that a prioritized test suite could mitigate the drawbacks of over-selecting test cases.

2.2 Test Case Prioritization (TCP)

Another challenge of regression testing is to detect failing tests fast. The objective of TCP is to re-order test cases according to some definition of priority, in order to get faster feedback from the

[1]Available at: https://doi.org/10.5281/zenodo.5851288

test execution. A prioritized test suite still contains all test cases, so there is no loss of failures detection ability (assuming that test results are independent) – what changes is the amount of time that it takes for one or more failures to be detected. TCP can be described as a function $P(T)$ that provides a permutation of T. Some criteria often used for TCP include similarity-based, coverage-based, and history-based [19].

FAST [25] utilizes test source code as input for a *similarity-based algorithm* to prioritize the tests. Inspired by big data techniques, string representations of test cases are transformed using minhashing signatures, which are then ordered according to their similarity. The benefits of FAST are low overhead and scalability, which make it usable for large software projects. We chose it because of its low running times and relatively simple implementation.

FAST authors [25] examined several possible variations of it that trade off efficiency for accuracy when choosing the next test(s). These are all stochastic by nature; as the authors point out, if two test cases are ranked equally, the tie is solved randomly. In our experiments with FAST we observed that FAST-pw (which is one of the variations) produced consistently similar permutations when executed more than once with the same test suite. This was an expected result given that FAST-pw is designed to always select the test case that is the furthest away from the set of already-prioritized tests. It does so by computing the similarity between each candidate test and the set of already-prioritized tests in a pairwise fashion. Furthermore, FAST-pw was able to rank failing tests higher than other variations. Therefore, in this paper we consider the FAST-pw variant, and in the following we refer to it simply as FAST.

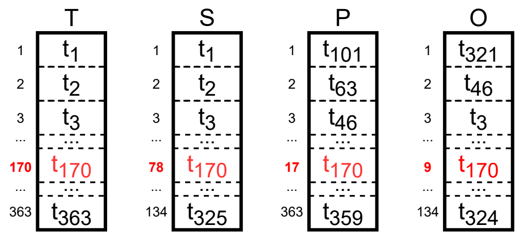
3 FASTAZI

Many researchers have shown that TCS and TCP provide substantial benefits to regression testing [3, 17, 19, 28]: a good selection decreases the overall testing time, while a good prioritization allows for detecting failures faster. However the two concepts are not mutually exclusive, and an orchestration of both may provide even further improvements, e.g., [6, 32].

If a test suite is selected *and* prioritized, both testing time and feedback time can be decreased. Recall that TCS can be defined as a function $S(T)$ that produces a subset of tests and TCP as a function $P(T)$ that outputs a permutation of T. Then, the goal of an approach that orchestrates TCS and TCP is to generate another function, $O(T)$, whose output is smaller than T and ordered to speed up failures detection. When discussing possible ways of orchestrating TCS and TCP, two approaches stand out.

Parallel execution. One approach is to independently perform the prioritization and selection of the entire test suite, and then arranging the selected tests according to the ordering given by prioritization. This approach has the advantage of allowing parallel execution of $S(T)$ and $P(T)$ and merging their outputs, instead of having one depend upon the other. To combine the outputs, it is sufficient to go through the prioritized list of tests and remove the ones that are not included in the selection.

Sequential execution. Another possible approach to the idea is performing selection first and then prioritizing the output. The advantage of this approach is reducing the running time of the prioritization, which would focus on the tests impacted by the changes



T: complete test suite; S: selection by Ekstazi;
P: prioritization by FAST; O: orchestration by Fastazi.

Figure 1: Sample outputs of Ekstazi, FAST and Fastazi

and thus more likely to fail. However, this also means that the selection and prioritization steps cannot be performed simultaneously (although it is still possible to parallelize the preparation steps).

Intuitively, it is not clear which option should be more effective or efficient than the other. Indeed, our experiments show that the effectiveness and efficiency of the parallel and sequential approaches are statistically equivalent (according to the same analysis detailed in section 5). For lack of space, henceforth Fastazi results always refer to the sequential execution, while the results of the parallel combination are available in the replication package.

As an example, Figure 1 contains sample outputs from Ekstazi, FAST and Fastazi[2]. Colored red, t_{170} is the failing test within a test suite T of 363 test cases. In S, the output of Ekstazi, this test is found in the 78th position, because several tests were excluded during the selection, while in the output P of FAST, it is moved up to the 17th position. Finally, the output of Fastazi, O, which is selected and prioritized, promoted the test to the 9th position.

Algorithm 1 provides an abstract view of Ekstazi and FAST[3], and outlines how Fastazi works in practice. Ekstazi requires tests to be compiled before performing selection, while FAST needs the hash signature of each test before prioritizing the suite. These two steps are independent and can be performed in parallel (they are both abstracted by the function GetHashesAndModified). After that, Ekstazi can perform its selection normally, and FAST prioritizes the resulting list of tests.

4 EVALUATION METHODOLOGY

4.1 Research Questions (RQs)

We evaluate Ekstazi against FAST, and their combination (Fastazi) against either of them, considering first their *effectiveness* in failure detection (RQ1). Then, based on the real example shown in Figure 1, we hypothesize that the potential gain in effectiveness of a combined approach could be better observed under a *limited test budget* (RQ2). Finally we also compare their *efficiency* (RQ3). Precisely, we formulate the following research questions:

RQ1: How do Ekstazi, FAST, and Fastazi compare in terms of effectiveness? For the scope of this study, the comparison between the respective effectiveness of the three approaches can be based on how quick they are in detecting the failures. As FAST uses the whole

[2]This example is based on results of the experiments on `Chart v26`. Actual test names are omitted for clarity.
[3]For a complete understanding of Ekstazi and FAST, refer to [9, 25].

Algorithm 1 Ekstazi, FAST and Fastazi overview

```
function GetHashesAndModified(files $F$)
    $M, C \leftarrow$ ExistingHashes(F)
    $M' \leftarrow \emptyset$                          ▷ Minhashes for FAST and CRC for Ekstazi
    $F' \leftarrow \emptyset$
    for $f \in F$ do
        $M'[f] \leftarrow$ ComputeHashes($f$)
        if $M[f] \neq M'[f]$ then
            Append($F', f$)
    return $M', F'$                                ▷ Updated hashes and modified files.
function Ekstazi(test suite $T$, files $F$)
    $S \leftarrow \emptyset$
    for $f \in F$ do
        for $t \in T$ do
            if TestDependsOn($t, f$) then
                Append($S, t$)
    return $S$                                     ▷ A selected test suite.
function FAST(test suite $T$, hashes $M$)
    $P \leftarrow \emptyset$
    while $|P| \neq |T|$ do
        $t \leftarrow$ PickNextTest($T, P, M$)      ▷ Pick the test that is furthest away from the so-far-ordered tests $P$ based on $M$.
        $P \leftarrow$ Append($P, t$)
    return $P$                                     ▷ A prioritized test suite.
function Fastazi(test suite $T$, files in the project $F$)
    Compile($T$) • $M, F \leftarrow$ GetHashesAndModified($F$)   ▷ Compiles the test suite using the build system and, in parallel, computes hashes and detects modified files.
    $S \leftarrow$Ekstazi($T, F$)
    $P \leftarrow$FAST($S, M$)
    return $P$                                     ▷ A selected and prioritized test suite.
```

test suite, we know it will detect all regression failures as a retest-all technique. Also, Ekstazi is developed as a safe TCS technique, thus it should, as well, detect all failures found by retest-all. Consequently, Fastazi too detects all failures. Thus, we refine the above question into the following two sub-questions:

RQ1.1: Between Ekstazi and FAST, which tool detects failures running fewer tests? While both Ekstazi and FAST have been shown to be effective in failure detection, we do not know whether when a new project version is released, potential regression failures would be revealed earlier by selecting those test cases that are affected by the changes (and randomly ordered) or instead by prioritizing test cases based on their similarity.

RQ1.2: How does Fastazi compare against Ekstazi and FAST with respect to feedback time? It is unclear if, and by how much, a combination of both techniques would provide lower feedback time from a test suite. With this question, we aim to discover if the orchestration of TCS and TCP has a positive and substantive impact to the regression testing workflow.

RQ2: How does a limited testing budget affect the effectiveness of the three approaches? While in RQs 1.1 and 1.2 we compared Ekstazi, FAST, or Fastazi without considering possible time constraints, with this RQ we aim at assessing whether, and how, testing under limited resources impacts each of the three approaches. This problem is similar to cost-bounded selection [4] (i.e., selecting test cases according to a predetermined budget), which can be a concern in large-scale industrial projects [6]. TCS and TCP each provide benefits when it is not possible to test 100% of the test suite in each execution, but they cannot assumed to be safe in these circumstances. Perhaps an orchestrated test suite would viable at even stricter testing budgets.

RQ3: How do Ekstazi, FAST and Fastazi compare in terms of time efficiency? With this question, we aim to discover what is the additional cost in terms of time required by either technique alone, and then by their orchestration. Inevitably, the orchestration increases total testing time, and we aim at assessing such drawback.

4.2 Evaluation metrics

The primary objective of TCS is to reduce the total number of tests executed per run, while TCP, on the other hand, has the goal of detecting failures quickly and reduce the feedback time of the test suite. Thus, the metric for an orchestration should somehow measure both of these objectives.

For **RQ1**, we utilize a metric called *Time To First Failure* (TTFF) [36]. Given a test suite T, its TTFF indicates the position of the first test to detect a failure. A low TTFF indicates that the test suite provides quick feedback. TTFF is a useful metric to evaluate both TCS and TCP, because it simultaneously encourages a tight selection of truly relevant tests and a prioritization that puts a failing test at the top of the list. However, since the output S of TCS is a subset of T, its size might be smaller than the output P of TCP. Therefore, for fairness, all TTFF results in this paper are normalized according to size of T. For example, if $|T| = |P| = 1000$, $|S| = 100$ and a failing test is in the 100th position of P but the the 50th position of S, then $TTFF(P) = 0.10$ and $TTFF(S) = 0.05$.

We also utilize *Average Percentage of Faults Detected* (APFD), the most popular metric for evaluating TCP solutions [19]. It is not designed to evaluate TCS and thus may not provide a fair comparison for Ekstazi; however, as previously explained, we assess here effectiveness in terms of how fast failures are detected by the compared techniques, and for this APFD provides an intuitive, well known assessment.

Regarding **RQ2**, when considering a limited testing budget, we use the output from **RQ1** and create versions of the suites that are cut off at certain points, according to the budget restriction. This data is analyzed in two ways: first, we observe, for each version of each subject, the proportion of the 30 variations that were able to detect the failure or not. Then, we also reduce this number into a binary form: 1 if the test suite detects the failure in all of its 30 variations, and 0 otherwise. This has the effect of punishing suites that are somehow inconsistent, rewarding those that catch the failure every time, since it can be important that an approach is consistent and reliable.

Finally, for **RQ3**, we measure the time taken to execute the discrete steps of the approach. For this, we use the GNU `time` utility (user+sys CPU time) to measure each step of the experiment individually, allowing us to understand where are the bottlenecks of the approaches.

4.3 Experiment design and execution

The goal of the experiment is to compare four possible arrangements of the test suite: the tests selected by Ekstazi; the test suite prioritized by FAST; the orchestration of both with Fastazi; and a random ordering of the test suite to provide a base case. Considering that both Ekstazi and FAST have been previously compared to several competing TCS and TCP approaches [21, 25, 37], we deemed it not necessary to add further alternatives in a direct comparison between the two tools.

Table 1: Subjects Used in the Evaluation.

Subject	# Versions	Min. # Tests	Max. # Tests
Chart	26	303	363
Cli	30	24	85
Closure	168	236	258
Codec	8	34	52
Collections	4	157	165
Compress	39	44	133
Gson	18	77	119
Jsoup	93	12	39
JxPath	4	27	33
Lang	28	87	178
Math	100	137	821
Time	23	121	123
Total	541	n/a	n/a

Min. # Tests and Max. # Tests show the smallest and largest test suites, respectively, among all versions of a certain subject.

We utilize as subjects 12 projects available as part of the Defects4J repository [15] that contains multiple versions of Java-based open-source software projects of different sizes. Each version is comprised of one commit containing a bug, the commit that fixed the bug, and metadata such as the files related to the bug, and which tests would detect it. Table 1 shows basic statistics about each project used in our evaluation. For each project, we show the number of versions used and minimum and maximum number of test cases (across versions). A few versions were skipped, either because their bugs are listed as deprecated by Defects4J, or because we ran into compilation issues for them (e.g., due to Java version incompatibility).

We used Ekstazi version 5.3, available on the project's website[4], as a plug-in for the Maven and Ant build systems. A script is used to automatically incorporate the Ekstazi task into a project's build script, allowing us to easily perform test selection over multiple versions of different subjects.

In the case of FAST, we used the source code from the replication package of the original paper[5]. This code was modified by us with two purposes. The first was to make FAST version-aware by storing the hash signatures of test cases between versions so they do not need to be re-computed unless there is a modification. This is important because computing the hashes is the most time-consuming part of FAST, so storing these representations for unchanged tests greatly reduces overhead after an initial execution. In addition, it was updated to guarantee that the input and output of both Ekstazi and FAST are in the same format.

Fastazi was not incorporated into the build system, but its results can be easily generated by using the output of Ekstazi as input for Fastazi, as shown in Algorithm 1. Observe that change-based TCS provides no benefit in the initial version of a project, since there are no changes to be detected; thus the first output of Fastazi, for each experiment subject, is identical to using FAST in isolation.

To collect the metrics, we did not actually execute the test suites given by each approach. First we collect the outputs of the approaches as text files containing lists of tests and then we calculate the metrics according to the position of the failing test(s) (ground truth given by Defects4J).

[4]http://ekstazi.org
[5]https://github.com/icse18-FAST/FAST

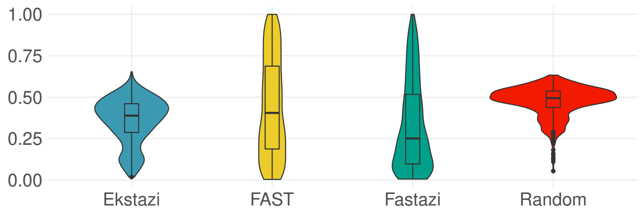


Figure 2: Normalized TTFF of different approaches

When measuring TTFF, the default order of test executions could have a large impact on (unprioritized) test suites; hence, for fairness we shuffled the output of Ekstazi 30 times and reported the average of these repetitions. Similarly, to account for the nondeterministic behavior of FAST, Fastazi and random, their outputs are also generated 30 times to reduce any potential noise in the data[6].

The experiments were executed in a Docker container running Ubuntu 20.04 LTS, using Java OpenJDK 1.8.0, Apache Maven 3.6.3, and Apache Ant 1.10.7. On all the projects, JUnit version was set to 4.12. The host computer was running macOS 11.0.1 on a 6-core Intel Core i7 processor, with 32GB RAM and SSD storage.

5 RESULTS

5.1 RQ1: Effectiveness

The answer to **RQ1** contains two parts: first, we compare the effectiveness of Ekstazi and FAST against each other (**RQ1.1**), and then, we assess whether orchestration TCS and TCP ultimately improves effectiveness (**RQ1.2**). For the sake of space we show the results for both subquestions within unified plots and tables.

The TTFF results are displayed as violin plots in Figure 2, in which each version of each subject is one data point (totaling 541). The violin plots display, in addition to the median and interquartile ranges, the full distribution of the data, which allows us to identify the different peaks in a distribution. For the TTFF metric, the lower the result, the better.

The visual assessment of the data shows us that the median TTFF achieved by Ekstazi and FAST are both close to 45% (the two leftmost plots in Figure 2), although there is a large difference in the distribution of the results. This can be explained in part by the experiment design – since Ekstazi's TTFF is an average of 30 permutations of S, the value tends to be close to the center. Indeed, we can see that the median for Random is very close to 50%, while Ekstazi is lower than that because S is frequently smaller than T.

When adding Fastazi to the comparison, we can see that its median TTFF is much lower, at around 25%, which is slightly over half the medians of Ekstazi and FAST. Both FAST and Fastazi can, in some instances, produce a TTFF close to 100%, meaning that the failing test is found at the very end of the test suite. In the case of FAST, this is explained by the fact that similarity-based TCP can occasionally produce poor results if there are multiple similar test cases out of which only one reveals the failure. With Fastazi, this happens less frequently; when it does, it is caused by performance

[6]We experimented with values between 10 and 50 and found that 30 provided a good amount of data without severely impacting the running time.

of both Ekstazi (selecting nearly 100% of the test suite) and FAST (ranking the failing test low) in specific subject versions.

After the visual inspection we proceeded with the statistical analysis of the data. 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[7]. We assessed at a significance level of 5% the null hypothesis that the differences in the TTFF values are not statistically significant. The observed differences in TTFF were statistically significant at least at the 95% confidence level (*p-value* $< 2.2e\text{-}16$).

Provided that significant differences were detected by the Kruskal-Wallis test we performed pairwise comparisons to determine which approaches are different[8]. The results are displayed in Table 2 (column *Group* for TTFF). 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. The approach (or group of approaches) that yields the best performance is assigned to the group (a). Looking at the results in Table 2, we can tell that Fastazi is different from (better than) Ekstazi (b). Ekstazi, on its turn, is different from (better than) FAST (c), and all the approaches are different from (better than) Random (d).

Table 2: TTFF and APFD for the different approaches.

Approach	TTFF			APFD		
	Med	*SD*	*Group*	*Med*	*SD*	*Group*
Fastazi	0.25	0.27	(a)	0.75	0.27	(a)
Ekstazi	0.39	0.14	(b)	0.62	0.14	(b)
FAST	0.41	0.29	(c)	0.60	0.29	(c)
Random	0.49	0.09	(d)	0.51	0.09	(d)

Med is the median, *SD* is the standard deviation, and *Group* displays the result for the pairwise comparisons after the Kruskal-Wallis test.

To understand the effect of choosing one technique over another on the effectiveness of the test suite, we measured the effect size using the Vargha and Delaney $\hat{A}_{12}$ measure [33], which tells us the probability of an observation from one group being larger than an observation from the other group. The results are displayed in Table 3. For interpreting the results, the $\hat{A}_{12}$ measure ranges from 0 to 1, and when the measure is exactly 0.5 the two techniques (in the column name) have equal performance. When $\hat{A}_{12} > 0.5$, the first technique outperforms the second, and when $\hat{A}_{12} < 0.5$, the second technique outperforms the first. Vargha and Delaney suggest that the effect size is *small* if the measure is over 0.56, *medium* if over 0.64, and *large* if the measure is over 0.71. As an example, when comparing Fastazi against Random for the subject Chart, Fastazi outperforms Random with a *large* effect ($\hat{A}_{12} = 0.82$) on the testing effectiveness. We can see that Ekstazi generally outperforms FAST, most of the time with a negligible or small effect, but there are cases where FAST outperforms Ekstazi. Fastazi, on its turn, outperforms Ekstazi and FAST with a non-negligible effect in the vast majority of the cases (18 out 24). The effect of choosing Fastazi over Ekstazi or FAST on the test effectiveness is large or medium in 11 cases.

[7]We used `kruskal.test()` from the `Stats` package in `R`.
[8]A significant Kruskal-Wallis test indicates that there is a significant difference between approaches, but does not identify which pairs of approaches are different.

Table 3: Effect size per subject.

Subject	Fastazi vs Random	Fastazi vs FAST	Fastazi vs Ekstazi	Ekstazi vs FAST
Chart	0.82 (L)	0.79 (L)	0.57 (S)	0.78 (L)
Cli	0.85 (L)	0.56 (S)	0.81 (L)	0.23 (L)
Closure	0.62 (S)	0.55 (N)	0.56 (S)	0.51 (N)
Codec	0.88 (L)	0.66 (M)	0.66 (M)	0.52 (N)
Collections	0.50 (N)	0.66 (M)	0.44 (N)	0.63 (S)
Compress	0.82 (L)	0.65 (M)	0.59 (S)	0.58 (S)
Gson	0.69 (M)	0.58 (S)	0.60 (S)	0.47 (N)
Jsoup	0.63 (S)	0.64 (M)	0.51 (N)	0.66 (M)
JxPath	0.50 (N)	0.56 (S)	0.44 (N)	0.57 (S)
Lang	0.66 (M)	0.63 (S)	0.65 (M)	0.58 (S)
Math	0.83 (L)	0.66 (M)	0.64 (M)	0.57 (S)
Time	0.60 (S)	0.61 (S)	0.52 (N)	0.61 (S)

L, M, S and N indicate large, medium, small and negligible effect size, respectively.

While TTFF captures how many test cases are required to reveal the first failures, the APFD metric measures the speed at which failures are revealed.

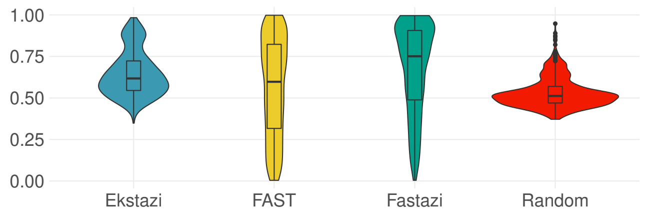


Figure 3: APFD of different approaches.

The observed APFD results are displayed as violin plots in Figure 3. For the APFD metric, the higher the better. Visual assessment of the results lead to the same conclusion as for TTFF: Ekstazi and FAST have similar medians, although FAST sometimes performs very poorly, while Fastazi has a higher median than both and mitigates most instances of poor performance from FAST. It is also visible that the peak of the distribution of Fastazi leans towards the highest possible values, while Ekstazi peaks at around 0.6.

Statistical analysis results are reported in Table 2 (right side). We performed again the Kruskal-Wallis rank sum test, followed by the pairwise multiple comparisons. All results in Table 2 are statistically significant at the 5% significance level. Both the groups assigned to each approach and the results of the effect size analysis were the same as the ones observed for the TTFF metric.

Summary of RQ1: While statistically significant differences were observed for the comparison between Ekstazi and FAST, a further investigation of the effect size revealed that the effect of choosing Ekstazi over FAST is either small or negligible in almost all the cases. Fastazi, on the other hand, outperformed Ekstazi and FAST with a non-negligible effect in the vast majority of the cases, suggesting that adopting Fastazi can help improving the testing effectiveness.

5.2 RQ2: Effectiveness under a limited budget

To answer RQ2, we proceeded with a detailed analysis of the impact of limiting the number of test cases with respect to those that would be run by Ekstazi. We investigated the impact on the failure detection capability of all the approaches when the testing budget is gradually reduced from 100% (no budget restrictions) to 25% of the test suite selected by Ekstazi, at steps of 25%. We discuss our findings first at a higher level, then with a more in-depth analysis of the results for each of the subjects considered in our study.

Figure 4 depicts the impact on failure detection capability on the different approaches. The results are grouped per budget (25% to 100%) and each approach is represented by a violin plot. For each version of each subject we counted how many times, out of the 30 repetitions (see subsection 4.2) each approach would be able to reveal the failure under the different budget restrictions (the number of observation in each violin plot is thus the same as the total number of versions, i.e., 541). The vertical axis varies from 0 to 30, respectively the minimum and maximum number of times an approach could reveal the failure across the 30 repetitions. Notice that for this RQ it is not a concern whether the failure is revealed by the first or the last test case, as this was already answered by **RQ1**; the concern here is whether the failure is revealed.

We can draw several observations from Figure 4: *i*) the median number of times the random approach can reveal the failure decreases almost uniformly as the budget becomes stricter; *ii*) because Ekstazi is the result of Ekstazi selection with random ordering, the observed medians and distributions are always slightly better than random, but following a similar trend as the one observed for random; *iii*) Fastazi outperforms the other approaches up to a budget restriction of 50%; *iv*) for the more restrictive budget of 25% the median of Ekstazi and even random are better than those of the Fastazi approach. Looking at the shape of the violin plots, however, we can see that even with a lower median Fastazi appears to have more observations leaning towards the maximum possible value.

To better understand such a behavior we analyze the data again from a different perspective in Figure 5, in which we observe the impact on failure detection capability on a per subject basis. This time, however, instead of counting how many times the failure would be revealed across the 30 repetitions, we are interested in the cases where the approach would consistently reveal the failure across all the repetitions for a given version. In this way we do not reward the cases where an approach would be able to reveal a failure by pure chance. Each subject is represented by a grouped bar plot and the height of each bar represents the number of times the approach was able to consistently reveal the failure, both in absolute (left vertical axis) and in relative terms (right vertical axis). For example, the maximum value in the left vertical axis for `Closure` is 168, which is the number of versions we considered for that subject and, at the same time, the maximum number of failures that can be revealed (one per version). The primary horizontal axis (bottom) represents the budgets, from 10% to 100%, whereas the secondary horizontal axis (top) shows what a given budget restriction would mean with regards to the whole test suite. This is important because the size of the test suite varies greatly across the subjects. For example, while a budget restriction of 50% for `Collections` means that 45% of the whole test suite is selected, only 23% of the whole test suite would be selected for `Chart` under the same budget restrictions (we recall that the budget restriction is calculated over the size of the test subset selected by Ekstazi).

By analyzing Figure 5 we can draw the following observations: *i*) with no budget restrictions (budget = 100%), Ekstazi and Fastazi were able to consistently reveal all the failures across the 30 repetitions; *ii*) for any other budget value below 100% Fastazi outperformed Ekstazi alone and FAST alone — in a very few cases FAST appears tied to Fastazi; *iii*) Ekstazi can consistently reveal some failures for almost all the budgets for `Chart`. For all the other subjects, it cannot reveal any failure for budgets restricted below 50%. For the particular cases of `Collections` and `Lang`, Ekstazi cannot reveal any failure consistently in the constrained budget scenario; *iv*) with the exception of `Codec`, `Collections`, and `JxPath`, Fastazi was able to consistently reveal some failures across the 30 repetitions for all the budgets, including the more restrictive budget of 10%.

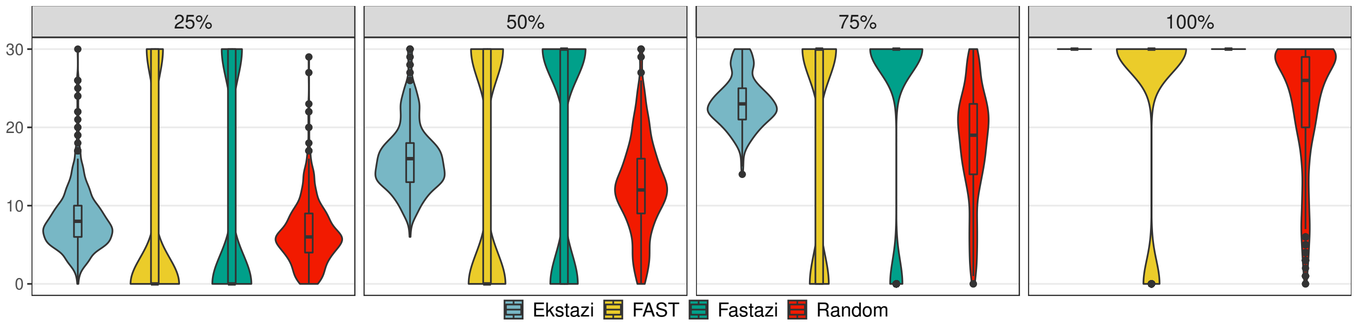
Summary of RQ2: Without controlling for the differences across subjects, Fastazi exposes the best failure detection capability even under restricted budgets, except for under 25% reductions in which Ekstazi and even random appear to show better median values. However, when we look from a per subject perspective and reward the approaches that consistently reveal failures, Fastazi outperform Ekstazi alone (with random ordering) and FAST alone (without TCS) for all the budgets considered.

5.3 RQ3: Efficiency comparison

To compare the time efficiency of Ekstazi, FAST, and Fastazi, we isolated the individual steps of each approach and measured the average time each step took, across the different versions of each subject program. In our measures, displayed in Table 4, the average build time (column 2) for each project was substantially longer than any cost added by Ekstazi, FAST, or Fastazi. This is an important observation because FAST can run its preparation phase (column 3), i.e, computing hashes of added/modified test cases, in parallel with the building process as it requires only test code. Fastazi takes advantage of this aspect to minimize the time overhead. Ekstazi, on the other hand, requires the code to be compiled before it can perform selection, so it cannot be run in parallel with the build.

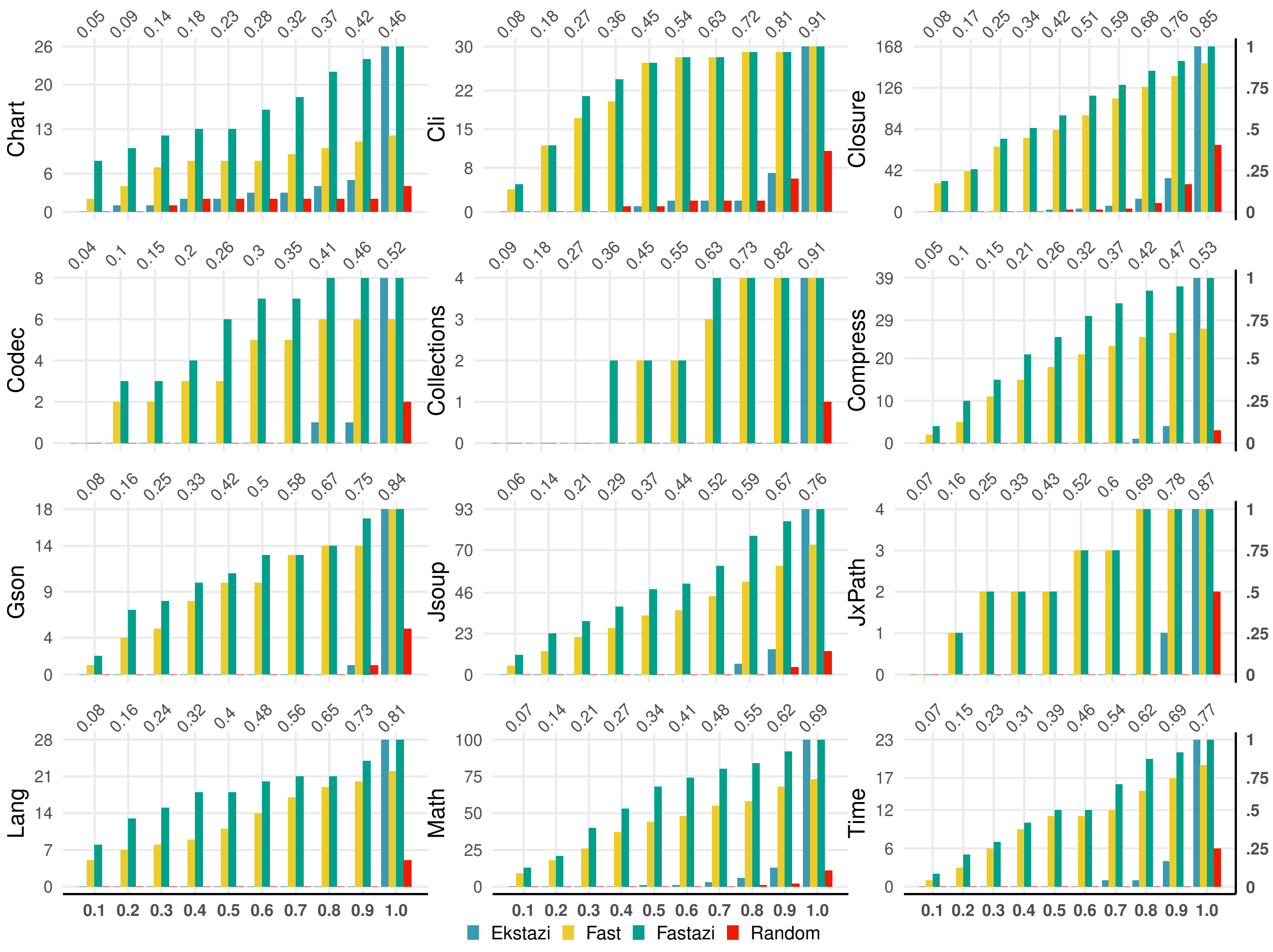
Table 4: Average Running Times (in ms).

Project	Build	FAST (setup)	FAST (TCP)	Ekstazi (TCS)	Fastazi (TCS + TCP)
Chart	4167	1105	112	3224	3259 (35)
Cli	2997	165	10	137	147 (10)
Closure	6627	1403	83	1571	1637 (66)
Codec	4581	551	6	163	166 (3)
Collections	6627	1043	29	237	259 (22)
Compress	4986	326	10	309	314 (5)
Gson	4901	221	20	297	313 (16)
Jsoup	6098	195	3	222	224 (2)
JxPath	3643	67	3	227	230 (3)
Lang	5032	621	37	262	287 (25)
Math	6903	1129	265	747	907 (160)
Time	11521	1439	20	500	515 (15)



Each panel represents a different budget constraint (100% is defined as the percentage of the test size selected by Ekstazi). The vertical axis shows how many times, out of the 30 repetitions, each approach is able to reveal the failure.

Figure 4: Impact on failure detection capability in a budget-constrained scenario.



The vertical axes represent the number of failures revealed in absolute (left) and in relative terms (right), whereas the horizontal axes show the budgets w.r.t the number of tests selected by Ekstazi (bottom) and w.r.t the total number of tests in the subject's test suite (top).

Figure 5: Impact on failure detection capability grouped by subject and by budget.

Table 5: Time Efficiency Comparison.

Comparison	*p-value*	Significance	Effect Size ($\hat{A}_{12}$)
FAST-Ekstazi	0.000462	***	0.04 (large)
FAST-Fastazi	0.000366	***	0.03 (large)
Fastazi-Ekstazi	1	ns	0.55 (negligible)

ns = not significant, *** means *p-value* < 0.001

Looking at the average execution times for FAST, Ekstazi, and Fastazi (the three rightmost columns in Table 4) the two main things we can observe are: *i*) overall, FAST is the technique that incurs the least time overhead; and *ii*) the overhead of Fastazi with respect to Ekstazi running time is generally very small.

To confirm our observations we performed the non-parametric Kruskal-Wallis rank sum test, and the result (*p-value* = 4.5e-05) confirmed that at least one of the approaches was different from the others with respect to the time efficiency. Provided that significant differences were detected, we proceeded with pairwise comparisons to determine which approaches were different and the results are displayed in Table 5. Statistically significant differences were observed when comparing FAST with Ekstazi and Fastazi, but not when comparing Fastazi with Ekstazi. Finally, to understand if the observed differences in time efficiency are not only statistically significant but also meaningful to support practitioners in the decision of whether Fastazi should be adopted, we measured the effect size. The results can be interpreted in an analogous way of that explained in Section 5.1. The effect size for the comparison of FAST with Ekstazi and Fastazi was $\hat{A}_{12} = 0.04$ and $\hat{A}_{12} = 0.03$, respectively, indicating that *the effect on the time overhead when running Ekstazi or Fastazi is large.* On the other hand, the effect size for the comparison between Fastazi and Ekstazi was $\hat{A}_{12} = 0.55$, indicating that *the additional time overhead incurred by Fastazi when compared with Ekstazi is negligible.*

It is important to notice that such results concern the overhead time required by the studied techniques, which are anyhow one or two orders of magnitude shorter than the time required for actually running the whole test suites.

Summary of RQ3: When considering the three approaches in isolation, FAST is the most efficient one and the difference with respect to the time overhead incurred by the other approaches is *large.* The additional time overhead incurred by Fastazi for prioritizing the test cases selected by Ekstazi is not statistically significant and the effect size is negligible.

5.4 Takeaway

RQ1 shows that Fastazi is statistically more effective than Ekstazi, which in turn is more effective than FAST and all techniques outperform random prioritization. Furthermore, in **RQ2** it is shown that under limited test budgets Fastazi consistently outperforms Ekstazi and FAST. Meanwhile, **RQ3** indicates that Fastazi adds negligible overhead compared to Ekstazi, so incorporating TCP into a project that already uses TCS can be encouraged.

5.5 Threats to validity

We evaluated Fastazi using faults available in Defects4J. Our results and conclusions could be different had we used another bug repository. However, Defects4J is among the most popular bug repositories and is heavily used in research on regression testing. Additionally, it includes real faults, which strengthens our findings.

The fact that we use Defects4J means that we were running experiments on project versions that are potentially very far apart (e.g., years). In this setup, Ekstazi might select a very large number of tests, because it was designed for small code changes between two consecutive commits [9, 34]. However, Ekstazi ended up performing well even in our setup.

We defined the testing budget as the number of tests that one can run at each project version, which does not take into account the differences in individual test execution time. As we focus on unit tests, we do not expect that there would be substantial differences in execution time across tests.

To measure effectiveness, we used TTFF and APFD. As known the Defects4J subjects contain only one fault per version and hence the two measures behave similarly. To mitigate this issue, we need to perform more studies on subjects containing multiple faults, for which the APFD measure becomes more valuable.

In our experiments we assume that test execution is deterministic, which we know does not always hold in practice, i.e., tests are flaky [13, 23]. We have not observed any flaky behavior in our experiments: only the expected set of tests was failing in each run.

6 RELATED WORK

The literature on regression test selection and prioritization is huge and, for a comprehensive overview of proposed approaches, we refer to several existing surveys, both generally on regression testing, such as [3, 27], or specifically on TCS [17] and TCP [19]. In this section we focus on related work about comparing and combining TCS and TCP techniques.

Actually, while there are plenty of studies that compare among themselves different TCS techniques, e.g., [21, 31], or TCP ones, e.g., [14, 24], we could only find one earlier work by Najafi et al. [26] that evaluates selection and prioritization one against the other. The focus of Najafi et al.'s study is to understand if and how history-based techniques of either TCS or TCP, or their combination, can improve test effectiveness in the complex infrastructure of Ericsson company. Thus the objectives of their study are similar to ours, however their investigation is related to leveraging test execution history for both techniques. They conclude that in their context and considering cost savings, prioritization alone outperforms both selection alone and their combination. Differently from them, we aim here to combine different test criteria. Certainly a future promising direction can be that of further combining file-based selection and similarity-based TCP with the usage of historical test information.

As the above is the only paper we found that compares TCS and TCP, the remainder of the section we overview related work on combining them. Considering a Continuous Integration process, Elbaum et al. [6] propose to first apply TCS for the pre-submit testing of new or changed modules that must be integrated into the code base, and then TCP for the post-submit phase. Both the proposed TCS and TCP algorithms are conceived for very large-scale contexts, such as Google CI environment, and hence utilize simple lightweight analyses relying on the time elapsed since a test has been executed or failed. In comparison, our approach leverages more fine-grained information, combining change-awareness and similarity criteria that have both individually proven to be effective in revealing possible regression faults. At the same time, as both Ekstazi and FAST were originally conceived with practicality in mind, we expect that Fastazi can scale up even beyond the size of subjects in which it has been evaluated.

Silva et al. [30] assume a Software Quality Function Deployment process and prioritize the test cases starting from the relevance of product features based on customers' needs. Test selection is performed using Ant Colony Optimization that refers to the criticality of test cases as calculated by a Fuzzy Inference System, and do not consider the software changes. Overall, while sharing a similar goal with us, this approach relies on a specific process and needs to weight classes relevance, hence applicability seems less general. Banias [1] proposes to apply a dynamic programming optimization algorithm that returns a ranked selection of test cases, after they have been assigned a cost that consider several project-specific importance criteria. In the context of an agile approach, Kandil et al. [16] combine the prioritization of test cases at each sprint, based on various process parameters, with the selection from clusters of test cases that exercise same functionalities of failed ones. In summary, the above works [1, 16, 30] combine TCS with prioritization based on some weighting criteria that require project specific information. In contrast, by merely relying on test code similarity that has proven effective for test prioritization, Fastazi can be applied in fully automated way without requiring any test annotations.

Some works explore the usage of learning techniques for TCP in combination with TCS heuristics, within a Continuous Integration environment. For instance, at each release Spieker et al. [32] prioritize test cases based on the failure history by reinforcement-learning algorithms, and then – if resources are not sufficient – apply selection based on execution constraints. More recently, Bertolino et al. [2] evaluate the effectiveness of several learning techniques (both supervised and reinforcement ones) for prioritization, considering further features in addition to failure history, after having performed a conservative test selection by static class-level dependency analysis of changes. In such approaches the greatest emphasis is on the ranking of test cases, which can be obviously addressed as a learning problem, whereas selection is left to simple heuristics applied before [2] or after [32] test suite reordering. Here we consider both TCS and TCP as equally important and aim at evaluating their combined application against each individual technique. Moreover, neither work considered a notion of test case similarity as we do.

Finally, a few works aim at reducing the number of test cases to be executed when the budget is constrained, similarly to our study in **RQ2**. Cibulski and Yehudai [4] reduce the problem of *bounded* TCS to test prioritization, in that the test cases are prioritized and then the top k are executed. However, the ranking is driven by the exposure of test cases to changes, as usually done in TCS. Shi et al. [29] instead evaluate the "selection of reduction" approach, in which TCS is performed after test suite reduction [18]. Their results show that the combined approach achieves a gain in size reduction but at the cost of potential loss in failure detection effectiveness. Unlike this work, Fastazi aims at explicitly combining a selection criterion and a prioritization criterion, for the sake of improving failure detection rate, both in the case of available time for affording a safe selection, and even in the case of constrained testing budget.

7 CONCLUSIONS AND FUTURE WORK

Software regression testing has undergone extensive research in the last several decades. The largest part of solutions, though, addressed separately one dimension of the problem at a time. While many TCS and TCP techniques have been proposed, they have not been directly compared, only few authors look into integrated approaches for combined selection and prioritization, and no work empirically assessed the advantages of using TCS and TCP in combination over their individual application. In contrast, we believe that, by merging differing criteria for selection and prioritization, we can achieve the most from the restricted subset of test cases that can be executed at each new release.

Towards this direction, we presented a study directly comparing two recent practical and effective approaches to TCS and TCP, namely file-based selection (by Ekstazi) and similarity-based prioritization (by FAST). Our results show that Ekstazi generally outperforms FAST, although the effect size is negligible or small; however, their orchestration by Fastazi outperforms both with a non-negligible effect. Moreover, considering a limited test budget, Fastazi exposed a higher effectiveness in consistent way. After assessing the overhead imposed by each of the studied approaches, we can conclude that Fastazi is quite practical: if we parallelize the preparation steps, the additional cost of similarity-based prioritization of the test cases selected by Ekstazi is negligible.

We aim at further improving the effectiveness and efficiency of Fastazi by refining several technical aspects. In particular, to make the approach more easily usable, it should be integrated into build systems as a plug-in as Ekstazi is now. In addition to that, we would also like to try orchestrating other TCS and TCP techniques from the literature to understand the resulting challenges and outcomes.

More generally, with this work we have paved the way to exploring a full range of potential strategies of combining differing criteria for selection and prioritization. It can be worthwhile to also expand the study to the orchestration of techniques along other dimensions of regression testing, e.g., also test reduction or test amplification. Overall, we consider that for maximized efficacy under restricted budgets the problem of regression testing should be addressed in a holistic strategy that we called regression test orchestration.

DATA AVAILABILITY

All material required for the replication of this study can be found in our replication package [11], including: (1) experiment source code, (2) further details about the experiment subjects, (3) instructions for running the experiments, (4) raw and absolute values of the results seen in the paper, (5) data used as source for tables and figures, and (6) supplementary results of experiments not shown in the paper.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their feedback. This research was partially supported by a FACEPE grant (APQ-0783-1.03/21) and the US National Science Foundation under Grant No. CCF-1652517.

REFERENCES

[1] Ovidiu Banias. 2019. Test case selection-prioritization approach based on memoization dynamic programming algorithm. *Information and Software Technology* 115 (2019), 119–130.

[2] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12.

[3] Nauman Bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the search for industry-relevant regression testing research. *Empirical Software Engineering* 24, 4 (Aug. 2019), 2020–2055. https://doi.org/10.1007/s10664-018-9670-1 ISBN: 1066401896 Publisher: Springer New York LLC.

[4] Hagai Cibulski and Amiram Yehudai. 2011. Regression test selection techniques for test-driven development. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 115–124.

[5] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.

[6] 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*. 235–245.

[7] Michael G Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 234–245.

[8] Vahid Garousi, Ramazan Özkan, and Aysu Betin-Can. 2018. Multi-objective regression test selection in practice: An empirical study in the defense software industry. *Information and Software Technology* 103 (2018), 40–54.

[9] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/2771783.2771784

[10] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/2642937.2643019

[11] Renan Greca, Breno Miranda, Milos Gligoric, and Antonia Bertolino. 2022. *Comparing and Combining File-based Selection and Similarity-based Prioritization towards Regression Test Orchestration (Replication Package) (1.2)*. https://doi.org/10.5281/zenodo.6402708

[12] Mark Harman. 2011. Making the case for MORTO: Multi objective regression test optimization. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 111–114.

[13] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation (SCAM 18)*. IEEE, 1–23.

[14] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-Box and Black-Box Test Prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 523–534. https://doi.org/10.1145/2884781.2884791

[15] 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*. 437–440.

[16] Passant Kandil, Sherin Moussa, and Nagwa Badr. 2017. Cluster-based test cases prioritization and selection technique for agile regression testing. *Journal of Software: Evolution and Process* 29, 6 (2017), e1794.

[17] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *Comput. Surveys* 50, 2 (June 2017), 1–32. https://doi.org/10.1145/3057269

[18] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. 2018. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. *IEEE Access* 6 (2018), 11816–11841.

[19] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (Jan. 2018), 74–93. https://doi.org/10.1016/j.infsof.2017.08.014

[20] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. 2015. Supporting continuous integration by code-churn based test selection. In *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*. IEEE, 19–25.

[21] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 583–594.

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

[23] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.

[24] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. 2019. How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1054–1080. https://doi.org/10.1109/TSE.2018.2822270

[25] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST Approaches to Scalable Similarity-Based Test Case Prioritization. In *International Conference on Software Engineering*. 222–232.

[26] Armin Najafi, Weiyi Shang, and Peter C Rigby. 2019. Improving test effectiveness using test executions history: An industrial experience report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 213–222.

[27] R H Rosero, O S Gómez, and G Rodríguez. 2016. 15 Years of Software Regression Testing Techniques - A Survey. *International Journal of Software Engineering and Knowledge Engineering* 26, 5 (2016), 675–689. https://doi.org/10.1142/S0218194016300013

[28] Gregg Rothermel and Mary Jean Harrold. 1994. A Framework for Evaluating Regression Test Selection Techniques. In *International Conference on Software Engineering*. 201–210.

[29] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 237–247.

[30] Dennis Silva, Ricardo Rabelo, Matheus Campanha, Pedro Santos Neto, Pedro Almir Oliveira, and Ricardo Britto. 2016. A hybrid approach for test case prioritization and selection. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 4508–4515.

[31] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. 2016. Change-based test selection: an empirical evaluation. *Empirical software engineering* 21, 5 (2016), 1990–2032.

[32] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 12–22.

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

[34] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. Module-level Regression Test Selection for .NET. In *Symposium on the Foundations of Software Engineering, Industrial Track*. 848–853.

[35] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[36] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*.

[37] Lingming Zhang. 2018. Hybrid regression test selection. In *International Conference on Software Engineering*. 199–209.