Better Feedback Times Using Test Case Prioritization? Mining Data of Past Build Failures in an Automated Benchmark

Jakob Rott♣, Rainer Niedermayr♠, Elmar Jürgens♣

♣CQSE GmbH, München
{rott, juergens}@cqse.eu

♠Ergon Informatik AG, Zürich
rainer.niedermayr@ergon.ch

Abstract

In software projects with growing functionality, the number of tests increases fast which results in long execution times for the whole test suite. As a consequence, it is not possible to always execute the whole test suite after each commit so that feedback time to developers increases. With long test feedback times, the effort for an early fix rises and developers can be hindered in productive work.

One solution to reduce feedback times is test case prioritization. Although test prioritization strategies have been extensively studied, they are rarely used in practice and their benefits are widely unknown.

In this paper, we present a benchmark framework to evaluate the benefits of different test prioritization algorithms on open source projects and primarily use the time until the first failure (TUFF) as relevant metric. We conducted an empirical study with 31 open-source projects hosted on GitHub, using data of 437 builds that failed on the CI server. In 75% of the builds, the first test will fail within the first 18% of the total test suite's duration.

1 Introduction

Regression testing is well established to check that code changes do not break previously working functionality. A challenge in regression testing are more and more enlarging test suites. In practice, many development environments use **retest all** strategies that run all tests in no specific order. While this can be feasible in before-release testing, the problem of (too) long lasting test suites gets more severe in projects that want to test continously: It is not possible to execute all available tests after each commit or push in reasonable time. Test suites with a huge number of test cases are necessary for evolving systems but come with increasing costs and long execution times that make it hard to give fast feedback to developers about successful or failing tests. To conquer this problem, different approaches can be used:

- **Regression Test Selection** (RTS) techniques reduce testing time by running only a subset of tests of the complete test suite—Engström et al. present a survey on RTS techniques in [1].
- **Test Case Prioritization** (TCP) strategies that order tests in respect to certain goals, for example to reveal faults early—see [3] for common methods.

In the present study, we focus on benefits to developers in terms of a fast notification about a failing test. We therefore limit our investigated prioritization algorithms to TCP strategies, as longer lasting builds with all tests passing are accepted.

We designed a benchmarking system to assess TCP techniques in real-world software. It uses data from past builds and unveils how long a test suite would run until the first failure—that actually occurred on the build server—using different prioritization strategies. We used the **time until first failure** as metric in our benchmark. Having fast feedback from the CI system is not only beneficial for the developer (e.g., less or easier context-changes for rework) but also saves time and resources.

2 Approach

Build logs were collected from **TravisCI** and parsed to extract builds with test failures. Those builds should serve as study objects.

We evaluated seven different prioritization algorithms on those builds:

- An algorithm that takes into account recent changes in production and test code as well as per test execution times and code coverage. (Abbreviated as **F3D**.)
- Sorting test cases ascending by the duration of their last run. (**DUR**)
- Sorting test cases descending by their speed in lines per second. (**LPS**)
- Two algorithms that use the number of transitive method calls, one not counting already called methods (*additional* variant, **ANC**) and one counting them (*total* strategy, **TNC**).
- An algorithm, following [2], that sorts the test cases by their string distance among each other. (**DIS**)
- Random ordering of the test cases. (**RDM**)

Input data for the algorithms (i.e., code changes and coverage) was calculated based on the last preceding passing build. The result of a single algorithm was a sorted test list that represents the execution order in the test run. The **time until first failure** was determined by calculating how long it would take to reach any test in the set of failed tests.

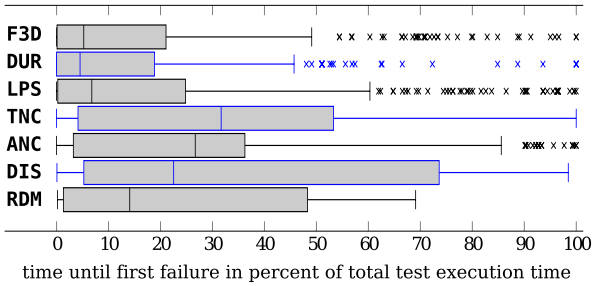

Figure 1: Statistics how fast a build would fail using the different prioritization strategies.

Table 1: Shows significant (sig.) differences in the average times to failure over the different algorithms in pairwise comparison (t-test). A checkmark is drawn if the algorithm in the row is sig. faster than the algorithm in the column.

	F3D	DUR	LPS	TNC	ANC	DIS	RDM
F3D		✗	✓	✓	✓	✓	✓
DUR	✗		✓	✓	✓	✓	✓
LPS	✗	✗		✓	✓	✓	✗
TNC	✗	✗	✗		✗	✗	✗
ANC	✗	✗	✗	✓		✓	✗
DIS	✗	✗	✗	✗	✗		✗
RDM	✗	✗	✗	✓	✓	✓	

3 Evaluation

The evaluation should answer which TCP algorithm shortens the **time until first failure** in a test run most.

Methodology For each failing build, the prioritization algorithms were applied. Using the duration of the last test execution, we calculated how long it would take to execute the ordered test cases until the first failure of the build occurs.

To statistically evaluate whether the prioritization algorithms differ in these times, we calculated an analysis of variance (**ANOVA**). It reveals if the algorithm has a significant effect on the time to first failure. Following, we used Student's t-tests to pairwise compare the times. Hence, the results express whether, in a pair of algorithms, one performs significantly better—meaning it reaches a failing test earlier. We used the significance level $\alpha = 5\%$ for the ANOVA and t-tests.

Results Figure 1 shows the distribution of the times to first failure for the algorithms visualized as boxplots[1]. Besides, the results of the Student's t-tests are presented in Table 1. It shows that there are relevant differences in pairwise comparison.

Discussion F3D and DUR are both faster than the other 5 algorithms. F3D and DUR show no significant performance difference among each other. Each of the algorithms beside F3D and DUR is significantly slower than at least two other algorithms and faster than at most one algorithm. **In terms of finding failing tests early, F3D and DUR are the most promising algorithms.** Regarding the t-tests, they are similarly fast and also the boxplots in Figure 1 show alike distributions.

The speed-based algorithm LPS works slightly worse. A possible reason can be fast large but free of failures tests that are executed early before the failing test—that has a lower speed in lines per second.

TNC and DIS perform clearly worse. They do not differ in pairwise comparison. Slightly better is the performance of ANC that takes into account only prior uncalled methods.

Noticeable is the measured performance of RDM: In 50% of the builds the first failure was found in below 15% of the total execution time. We investigated this issue and revealed that in two projects often a huge number of test cases failed which increases the expectation value of the RDM's performance.

Threats A threat to the internal validity of the results is that 327 (75%) of the investigated 437 builds were taken from 6 (19%) of the 31 projects. Project-specific effects in those 6 projects have a larger impact on overall results.

Another threat is that we did not execute the tests in the prioritized order. Instead we calculated hypothetical execution times. Due to capabilities of the testing system and interdependencies of tests, it might not always be possible to run tests in the proposed order. However, further feasible execution orders of tests can be enabled by enhanced testrunner implementations in the future.

[1]The whiskers follow the definition of John Tukey, have a maximum length of 1.5 times the interquartile range (IQR) but can be shorter if the maximum value is nearer to the box.

4 Conclusion

In this paper we presented a benchmarking system for test prioritization algorithms and an exemplary study with projects hosted on **GitHub** and using past build data from **TravisCI**. This offers a new view on the performance of TCP algorithms.

The results show that there are differences in the performances of TCP algorithms in various aspects. The proposed benchmark enables a quick and comparatively easy uncovering of them.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "Sofie, 01IS18012A". The responsibility for this article lies with the authors.

References

[1] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[2] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[3] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.