Using Test Suite Statistics for Test Prioritization

Michaela Williams†
Computer Science Department
University of Kentucky
Lexington, KY, USA
mrwi245@uky.edu

ABSTRACT

In continuous integration development environments, regression testing must be able to be conducted frequently and quickly. In order to make regression testing more effective, new test prioritization methods that do not require code instrumentation need to be developed. Although sorting algorithms are generally not desirable in time and computation-sensitive situations, particularly with large sets of data to sort, simple sorting algorithms are shown to be able to operate in a reasonable amount of time on collections of up to 30,000 test suites. It is possible that prioritizing smaller test suites through a sorting algorithm may be a slightly effective method of test prioritization, but the results are mixed and not entirely clear.

CCS CONCEPTS

• Software testing and debugging

KEYWORDS

Regression Testing, Continuous Integration, Software Testing, Test Prioritization

1 Introduction

In order to effectively perform regression testing in a continuous integration development environment, it is important to be able to prioritize test suites without code instrumentation [1], as the frequency of code changes and the need to thoroughly test all changes before they can be launched makes code instrumentation impractical. As a result, there is a need for the identification of potential test prioritization methods that work quickly, effectively, and without the need for code analysis.

The solution suggested by this paper is that instead of prioritizing test suites based on properties of the code, suites could be prioritized based on properties of the tests themselves, such as size and failure history. This paper identifies a reasonably light-weight way of selecting local "best" tests based on five different algorithms and aims to show that the use of sorting algorithms for test suite prioritization is both reasonable and potentially practical.

2 BACKGROUND

Regression testing is the process of ensuring that when previously working software is modified, it still functions properly [1]. Test prioritization is a method to refine the process of regression testing, such that tests that are believed to reveal more information are performed earlier than tests that are believed to reveal less information. For the purposes of this paper, "tests that reveal more information" indicates tests that are more likely to fail. When a test fails, that provides important feedback to the development team so that they can work on correcting the failure, so if a single test A in a series of ten is going to fail, it is better that A be run first or fifth rather than last.

3 ADVANTAGES OF A CONTINUOUS INTEGRATION DEVELOPMENT ENVIRONMENT

In continuous integration environments, code changes very frequently but still must be thoroughly tested before launch. The automated testing that should occur after each code change will result in statistics that are associated with each test suite. These statistics may include the number of times the suite has been run, the number of times it has failed, or how recently the suite was run. While there may not be sufficient time in the development cycle to perform code instrumentation, these test statistics are automatically generated through the normal process of testing and, in theory, require minimal additional time to be useful to the regression testing process.

4 APPROACH

To calculate the prioritized list of test suites, the entire list is sorted based on a modified form of selection sort that finds the "best" suite for a segment of all suites at a time, looping over the original list until only one suite remains. Unlike traditional selection sort, this approach returns a new list and does not operate in-place.

The original design involved an exhaustive selection sort, but this was not feasible in practice, particularly with the number of test suites used in experimentation (<=30,000). To reduce the amount of computation required for each selection, the decision was made to instead look for a local best at each iteration. A parameter *sortSize* was added to the sorting algorithms that would define how many tests would be viewed at a time. At sortSize=100, the best suite out of 100 would be selected and appended to the sorted list.

4.1 SORT BY SIZE

Size, as here used, is an approximate measure of the amount of time and computational resources needed to run the test suite to completion. For the purposes of experimentation, the possible options were "SMALL", MEDIUM", and "LARGE" [interpreted as 1, 2, and 3 respectively by the program], but a larger set of possible options might be useful in practice as the existence of only three categories will result in a large number of ties when sorting by size.

This sorting algorithm prefers smaller tests, the intuition being that prioritizing smaller tests would reduce testing time.

4.2 SORT BY FAILURE RATIO

It is possible that a test suite's historical performance may be indicative of future performance. Failure Ratio as defined here is

$$\frac{number\ of\ suite\ failures}{total\ number\ of\ times\ the\ suite\ has\ been\ run}$$

If a test has frequently failed in the past, that might indicate that the test covers some complicated or bug-prone portion of the software, which in turn may indicate that it is more likely to fail in the future.

4.3 SORT BY MOST RECENT FAILURE

If a test suite has recently failed, it is possible that the current build being tested has attempted to fix the fault that caused the failure. However, the usefulness of this for post-submit testing is disputable, as these tests would perhaps be performed in pre-submit testing.

For the purposes of this paper, "most recently failed" was defined as the suite with the lowest "Time since last failure" value.

4.4 SORT BY LEAST RECENTLY RUN

If a test suite has not been run in a long time, there may be undiscovered faults that have been created since its last run. However, practically speaking, it is possible and perhaps even likely that all tests would be run during each testing cycle. In this situation, this metric would not only not be useful but potentially produce the same test order every time as tests that it originally prioritized would have been run earlier than tests with a low priority. This should result in them again receiving the same priority.

For the purposes of this paper, "least recent" was defined as the suite with the highest "Time Since Last Run" value.

4.5 SORT BY A COMBINATION OF STATISTICS

Even if other metrics were introduced, it is probable that no one statistic would be more informative than a combination of factors. Preliminary testing was performed during and after the writing of the preceding, singular metric algorithms. At the time, Size seemed to be the best performing metric. The performance for Failure Ratio on its own was extremely poor, but in theory it still seemed promising, so it was combined with Size to make a new sorting algorithm.

To find the "best" suite in the current subset of tests, the algorithm iterates over every test suite in its current subset of suites and stores the value and index of the best suite that it has seen so far. If a test suite of the same size as the current best suite is encountered, the current test suite and current best suite's Failure Ratio values are compared and the suite with the higher Failure Ratio will be selected as the new current best.

5 EXPERIMENTS

To test the feasibility and effectiveness of these sorting algorithms for test prioritization, a series of experiments were performed that measured the time and tests that it would have taken to detect some percentage of failing test suites if the tests were sorted using each algorithm. The original test order and a randomized test order were used for comparison.

5.1 GOOGLE DATASET

One of the datasets used consisted of real test data, "The Google Shared Dataset of Test Suite Results", hereafter referred to as "the Google Dataset" [2]. This dataset contains the results of around 3.5 billion test suites. Most relevantly to this paper, these results include the time it took to run each suite, the approximate size of the suite, and whether the suite passed or failed.

5.2 GENERATED DATASET

The other dataset used was generated using randomization functions in Microsoft Excel. In this section, the calculation of the data will be discussed in some detail.

About half of the data fields are dependent on some other field(s). The non-dependent fields are *Suite Name*, *Time to Run*, *Times Run*, and *Time Since Last Run*. Suite Name is simply Suite_x, where x begins at 1 and increments with each successive suite. Time to Run is the result of the RAND() function, optionally multiplied or divided by some constant to introduce more variety. Times Run is a random whole number between 1 and 30, assuming that each test available for prioritization has been run at least once and no more than thirty times. Time Since Last Run is generated in a similar way to Time to Run.

The simplest dependent field is *Times Failed* which measures the number of times a test suite has failed. It is simply a random whole number between 0 and the value of that suite's Times Run field. The *Time Since Last Failure* field is calculated based on the values of the Times Failed, Times Run, and Time Since Last Run fields. If Times Run is equal to Times Failed, then that means that the suite failed the last time it was run, so Time Since Last Failure is assigned the value of Time Since Last Run. If Times Failed is equal to 0 that means that it has never failed, Time Since Last Failure is assigned to a random whole number between the value of Time Since Last Run and 1000. The resulting value is assumed to be the time the test was created, but this data is not considered relevant to this paper unless the generated value is lower than some test that has failed before. It would have been safer to initialize the value to some value larger than could be generated normally for Time Since Last Failure. If Times Failed is neither equal to Times Run nor to 0, then it is assigned a random whole number between Time Since Last Run and 100.

The calculation of *Size* is fairly complicated, but likely did not need to be. It is assumed that a "LARGE" test will take at least 10 seconds to run, a "SMALL" test less than 1 second, and a "MEDIUM" test any time between if the test passed, so in this case Size is based strictly on the value of Time to Run. If it failed, then one of the three sizes will be selected based on both Time to Run and the value of a call to RAND(), which returns a decimal value between 0 and 1. If Time to Run is less than 1 and RAND() is less than 0.38 , it is assigned "SMALL", if greater than or equal to 0.38 and less than 0.85, it is assigned "MEDIUM", and if 0.85 or greater assigned "LARGE". If Time to Run is between 1 and 10, Size is assigned "MEDIUM" if RAND() is less than 0.3 and "LARGE" otherwise. If Time to Run is 10 or greater, is assigned "LARGE".

The calculation of *Passed* measures whether or not a test suite passed. This calculation makes the impactful assumption that the higher the Failure Ratio of a suite the more likely it is to fail on the current run. This assumption is made primarily to see if sorting by Failure Ratio would be useful if this assumption were true. The randomization of the results means that no information can be learned here regarding whether or not the assumption is true. The value of Passed is false if the Failure Ratio multiplied by RAND() is greater 0.7 and true otherwise. 0.7 still appears to be too low of a threshold, as the total number of failing test suites is 614 out of 30,000. This seems to be too high when compared to the 100 failures out of 24,302 test suites from the Google dataset.

Procedure

To test the test prioritization methods on these datasets, three trials were run to test performance in detecting 90%, 95%, and 98% of failing test suites. For each trial, the original test order, a randomized order, and all applicable algorithmically sorted order were tested. For the Google Dataset, the only algorithm that could be used was Sort by Size, but the generated dataset used Sort by Size, Sort by Failure Ratio, Sort by Most Recent Failure, Sort by Least Recently Run, and the Multi-Factor Sort. As both datasets contain the results of test suites and not actual tests, the process of running test suites is simulated by iterating through the passed in list of tests and tracking the number of tests run, the time it would have taken to run those tests, and the number of failed tests seen. Once the percentage of failing suites is reached, the simulation ends, and the results are reported.

Results and Discussion

The results for the Google Dataset were mixed. There is a decent time improvement over random in detecting only 90% of failures, the best gain being about 20 hours when using a subset size of 1000. At subset sizes 1000 and 500, Sort by Size still performs reasonably well at detecting 95% of failures, but in general the effect becomes increasingly nominal as both the subset size and percentage of failures detected increases. It should also be noted that all subset sizes performed worse than the original order, although this is perhaps not representative of a typical situation.

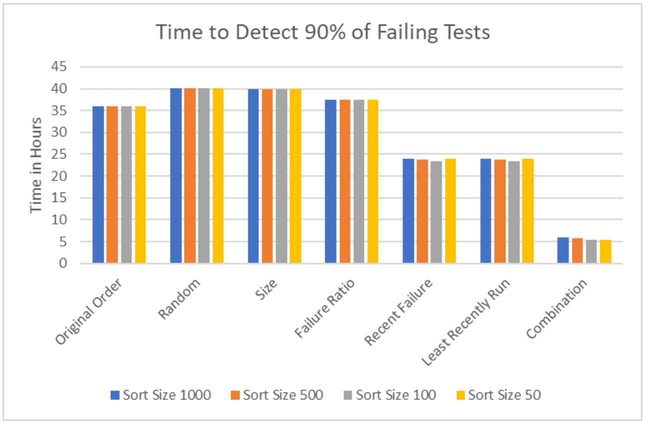
There was virtually no difference in the number of tests that had to be performed, but the sort time seems quite reasonable considering 24,302 test suites were sorted. The higher the subset size, the longer it takes to sort, but generally higher subset sizes perform better than lower sizes. The longest sort time was a little over 8.201 seconds at subset size 1000 and the shortest was 0.808 seconds at subset size 50.

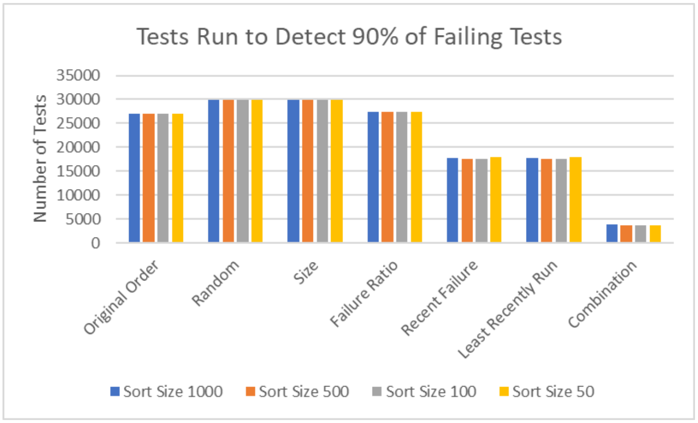
Part of the purpose of this paper was to compare the results obtained for the Google Dataset to the results found by the authors of "Techniques for Improving Regression Testing in Continuous Integration Development Environments" who also used this dataset [1]. However, due to technical constraints, only 24,302 tests results were used out over 3.5 billion which were available, so it is difficult to compare our results. There were also only 100 failures to detect in the shortened version of the dataset used for these experiments, as opposed to the 5000 found in the full dataset, so an attempted comparison would likely not be fair.

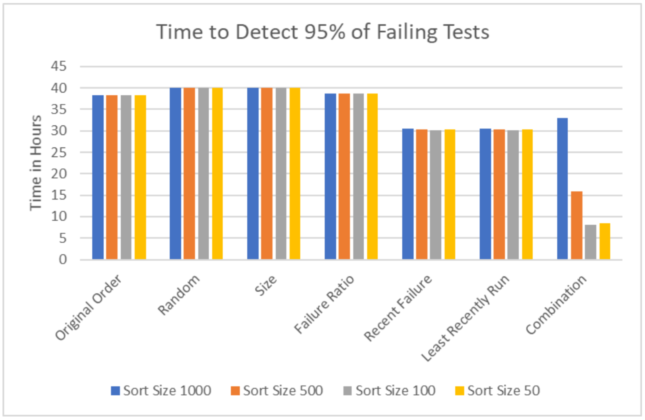
Before discussing the results for the generated dataset, it should be stated that its uses are limited. As the data was not real, but largely randomly generated, its applicability to the real world is unknown and the primary benefit is its measurement of sorting time for the proposed algorithms. However, it is possible that Sort by Size still has some validity due to the relationship between Time to Run and Size in generating the data. Additionally, in an environment where a high Failure Ratio is positively correlated with test suite failure, the results for Sort by Failure Ratio and Multi-Factor Sort may be helpful as well. The results for Sort by Most Recent Failure and Sort by Least Recently Run will not be discussed here, with the exception of their sort times, as the random nature of the values for these fields means that there is no reason to expect applicable results.

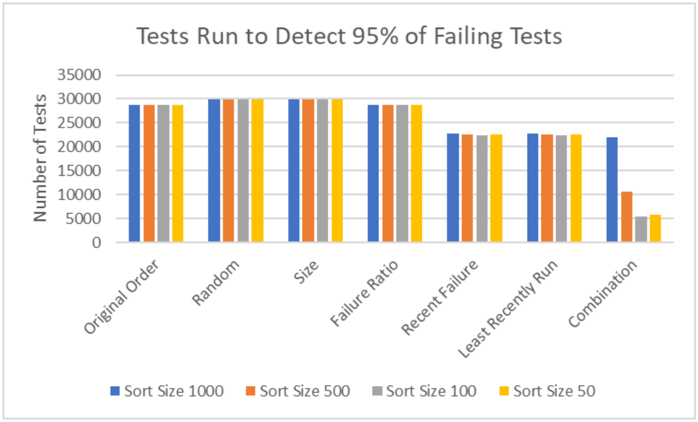
However, both Sort by Size and Sort by Failure Ratio performed extremely similarly to random. The reason for the discrepancy between the results for the Google Dataset and the generated dataset is unclear, but it may be related to a higher percentage smaller test suites or suites with shorter run times in the Google Dataset. Perhaps the generated dataset was improperly generated or perhaps the Google Dataset is not necessarily representative in this area. The original order for the Google Dataset performed very well, implying that either failing suites, short suites, or both occurred earlier in the order than they perhaps otherwise would have. When a tie occurs, the Sort by Size algorithm keeps the suite it saw first which may have given inadvertent priority to test suites that appeared earlier in the original order. There were also only three possible size options which may have affected the broader applicability of the test results.

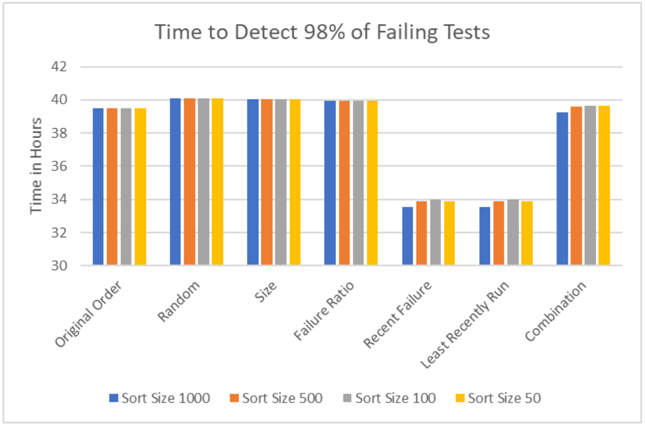
Figure 1: Results for the Generated Dataset

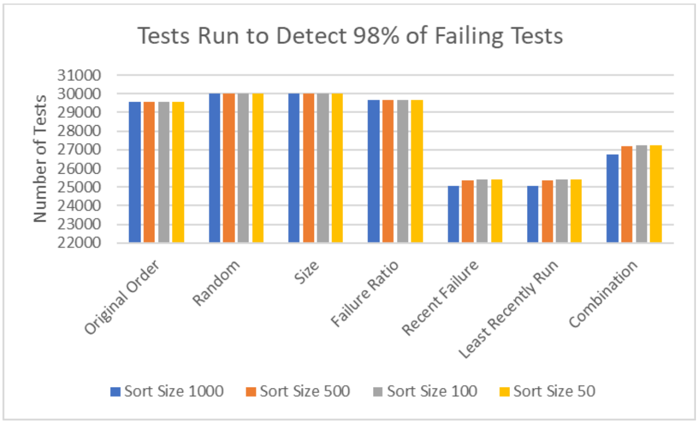


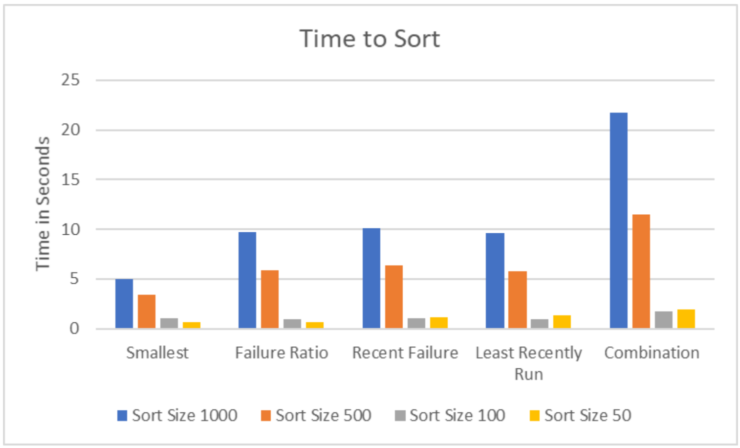


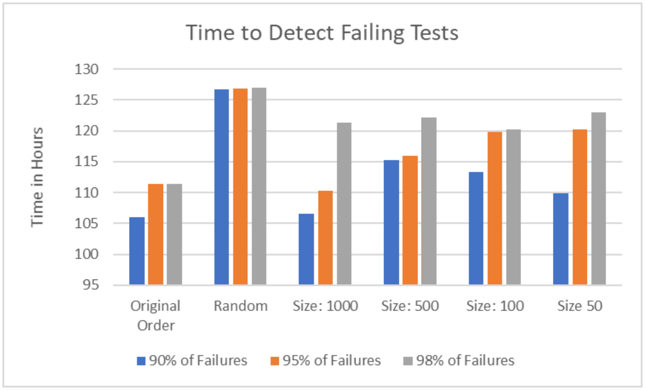


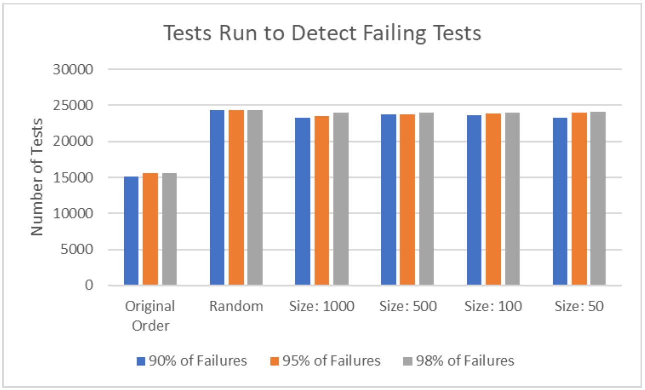












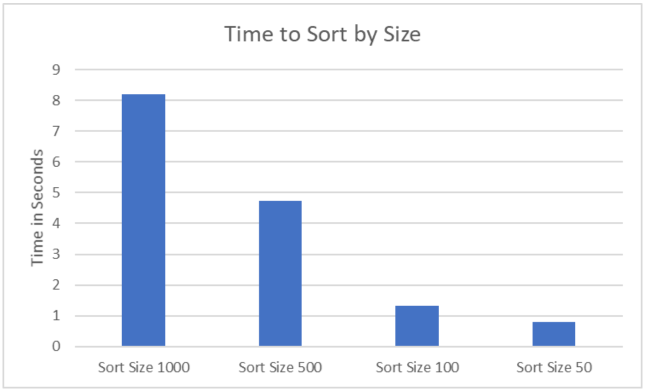


Figure 2: Results for the Google Dataset

Due to the poor performance of Sort by Size and Sort by Failure Ratio individually, the performance of Multi-Factor Sort, which combines the two, is confusing. As the generation of this dataset made the assumption that a higher failure ratio necessarily corresponds to a higher chance of a test suite failing, the algorithm essentially knew directly which tests were most likely to fail than others. When it found two tests that were the same size, it could choose the one more likely to fail while still attempting to optimize for test time. However, if this is the case, it is not clear why Sort by Failure Ratio performed so poorly on its own.

The results for sort times were largely consistent with the results for the Google Dataset. The exception to this is, not surprisingly, the Multi-Factor Sort which takes 21.72 seconds to sort at subset size 1000. If the results for Multi-Factor found in this paper were more broadly applicable, however, the sort time would seem to be worth it. However, due to the observation that performance is worse for higher subset sizes and due to the poor performances when sorting by Size and Failure Ratio individually, this is likely not the case. When using real data, higher subset sizes are positively correlated with performance, making the results for Multi-Factor Sort suspect.

CONCLUSION

The most promising and potentially more widely applicable result of this study is the relatively low cost of sort algorithms for the purposes of test case prioritization, but it is also possible that prioritizing smaller test suites may be applicable as well based on the results of the Google Dataset tests. Regrettably, sorting by Failure Ratio does not appear to be effective even in environments where it is positively correlated with test suite failures. In a real environment, it is possible the behavior would be different, but the results from the generated dataset are discouraging.

REFERENCES

[1] *Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, 235–245. DOI:https://doi.org/10.1145/2635868.2635910*

[2] *Sebastian Elbaum, Andrew Mclaughlin, and John Penix, "The Google Dataset of Testing Results", https://code.google.com/p/google-shared-dataset-of-test-suite-results/ 2014*