Prioritizing Tests for Improved Runtime

Abdelrahman Baz
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
ambaz@utexas.edu

Minchao Huang
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
minchao@utexas.edu

August Shi
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
august@utexas.edu

ABSTRACT

Regression testing is important but costly due to the large number of tests to run over frequent changes. Techniques to speed up regression testing such as regression test selection run fewer tests, but they risk missing to run some key tests that detect true faults.

In this work, we investigate the effect of running tests in different test-orders on overall test runtime in Java projects. Variance in runtime across different test-orders can be due to various reasons, such as due to dependencies between tests. In our evaluation, we run tests in different, random test-orders, and we find on average that the slowest test-order per project can be slower than the fastest test-order by 31.17%. We also develop a technique for guiding a search for the fastest test-orders by clustering test-orders based on their runtimes and generating test-orders based on observed in-common relations between tests in the fastest test-orders.

ACM Reference Format:
Abdelrahman Baz, Minchao Huang, and August Shi. 2024. Prioritizing Tests for Improved Runtime. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Regression testing is an essential part of the software development lifecycle, where developers run tests after every change, ensuring their recent code changes do not break existing functionalities [49]. Despite its importance, regression testing is often time-consuming and resource-intensive, especially as software systems and their test suites grow in size and complexity [10, 28, 31].

Prior work proposed various regression testing techniques to address the high cost of regression testing, such as test-suite reduction (TSR), regression test selection (RTS), and test-case prioritization (TCP) [49]. TSR and RTS both aim to speed up regression testing by running fewer tests after every change. TSR reduces the test suite size by removing redundant tests based on some heuristics, such as code coverage [5, 7, 11, 13, 17, 19, 38, 41, 52]. RTS analyzes the code changes and selects to run only the subset of tests affected by those changes [9, 10, 12, 14, 15, 22, 28, 36, 37, 43, 45, 46, 50]. Since both TSR and RTS run a subset of the full test suite, there is risk of missing to run some key tests that would fail and detect newly-introduced faults [36, 42, 54].

Meanwhile, TCP aims to reorder tests to run in a different, better order, where techniques prioritize the tests to first run those that are more likely to detect faults, based on various metrics like code coverage or diversity between tests [6, 16, 18, 25–27, 30, 35, 40, 47, 51]. As soon as a test fails, developers can immediately start debugging even as other tests are still running. TCP still runs all tests, so there is no risk of missing to run any test that can detect newly-introduced faults, but all tests still need to be run, so even as developers can debug earlier upon a failure, there is still machine cost needed for running all tests. The general consensus is that the time needed to run all tests in any order remains the same.

We propose a different means of reordering tests to speed up testing even while running all tests. We intuit that there are other reasons for why a full test suite can have varying runtimes when run in different test-orders, such as due to dependencies between tests [53] or machine properties. For example, Stratis and Rajan previously studied how running tests in different test-orders can have an effect on code caching and cache locality in C programs, which in turn affects the overall runtime [48]. If there is variation in runtime between different test-orders, then it stands to reason that there are test-orders that run faster than others, and so a developer would prefer to run the tests in that test-order while still maintaining full fault-detection capability by running all tests.

In this work, we first demonstrate the variation in runtime between test-orders in Java projects through a preliminary study where we randomize test-orders across 12 Java projects' test suites, running each test-order five times to collect a distribution of runtimes for each test-order. We find that the runtime indeed varies, on average by 31.17% when comparing the fastest test-order's runtime to the slowest test-order's runtime. We then propose a technique for reordering tests to search for the fastest test-order by focusing on relative positioning between test classes in the test-order. Our technique is based on the intuition that a major reason for differences in test-order runtimes is due to dependencies between tests [20, 44, 53]. Our approach generates different test-orders by following the relative ordering between tests found in-common among already observed fastest test-orders. While this approach is effective at generating the fastest test-orders for some projects, likely due to there being dependencies between tests related to performance, there are still other reasons for runtime variances in different projects. Future work can focus on developing different techniques for reordering tests based on these other factors.

2 PRELIMINARY STUDY

For our preliminary study, we want to measure the variation in runtime across different test-orders. We evaluate on 12 open-source Maven Java projects from GitHub, sampled from prior research on software testing [23, 24, 33, 46]. If the project contains multiple modules, we randomly take one module from the project and evaluate on its test suite, as long as the test suite has more than five test classes and runs longer than 15 seconds. Table 1 shows our evaluation modules, where "ID" is an ID we give to each module for ease of future presentation, "Project" is the name of the project, "Module" is the name of the module in the project we use for evaluation, and "TC" is the number of test classes in that module.

For each module, we randomly shuffle the test classes in the test suite to form different test-orders. We generate 30 test-orders

Table 1: Project characteristics and evaluation results.

ID	Project	Module	# TC	# Clusters (Random)	# Clusters (Guided)	Fastest rank	Avg. gen. time (s)	# Rnd. gen. orders
M1	admiral	compute	74	13	4	5	1.54	0
M2	incubator-dubbo	dubbo-rpc/dubbo-rpc-dubbo	15	9	7	20	0.09	0
M3	commons-math	commons-math-legacy	315	2	2	1	156.92	0
M4	hazelcast-jet	hazelcast-jet-sql	90	2	4	1	2.00	1
M5	rocketmq	acl	9	2	6	10	0.06	2
M6	Achilles	integration-test-2_1	40	2	2	1	0.24	5
M7	languagetool	languagetool-language-modules/uk	27	2	2	23	0.20	0
M8	jitwatch	core	32	3	2	2	0.46	0
M9	flink	flink-table/flink-table-runtime	123	8	2	6	5.44	0
M10	Openfire	xmppserver	69	7	6	1	2.54	0
M11	commons-io	.	200	7	2	1	37.58	0
M12	elastic-job-lite	elasticjob-lite/elasticjob-lite-core	51	2	2	2	1.05	0

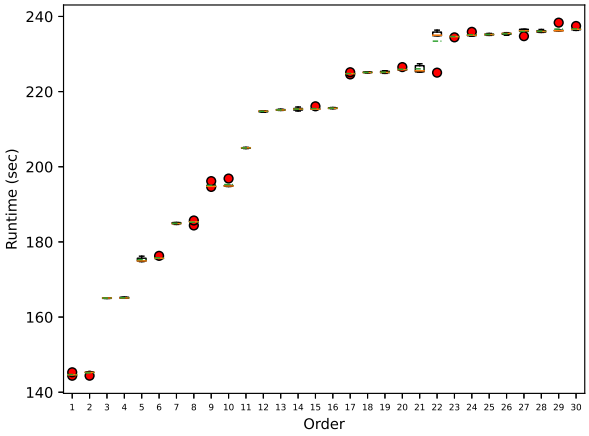

Figure 1: Runtimes of different randomly generated test-orders for project `apache/incubator-dubbo`.

per module, while ensuring each test-order to be unique and have the same test outcomes (different test outcomes can be due to test-order-dependencies [21, 53]; we generate a new test-order if tests have different outcomes. We then run each test-order five times to collect a spread of runtimes, obtaining a median runtime per test-order. We run all our experiments in a Docker container built from an Ubuntu 20.04 Docker image with JDK 17 and a modified version of the Maven Surefire plugin that allows us to control the test-order [3]. We run each module's tests in its own container, and we limit the container to use 2 CPUs and 8GB of RAM, similar to resources available in continuous integration services [2, 4].

Figure 1 shows the spread of runtimes across these different test-orders generated for tests in a module from project `apache/-incubator-dubbo`. Each box represents the variation in runtime (across five reruns) for a test-order. We see a wide range of runtimes, with the fastest test-order having a median runtime of 144.72 seconds, and the slowest having a median runtime of 236.70 seconds. We can essentially compute a similar plot for all modules.

Figure 2 shows a violin plot of the median runtimes for all the 30 test-orders, per module. The red plots represent the distribution of runtimes from the randomly generated test-orders. We observe that most of the modules show variation in runtimes, especially modules M1, M2, M3, and M4. We also observe that the average difference in runtime across all test-orders per module is 23.04 seconds, where on average the slowest test-order's runtime is 31.17% slower than the fastest test-order. Furthermore, we find all these differences between fastest and slowest test-orders' runtimes to be statistically significantly different based on the Mann-Whitney u-test [29].

Note that several test-orders may have similar runtimes as each other. We cluster the runtimes of different test-orders together and observe how many clusters there are. We use the K-means clustering algorithm from sklearn [34] while controlling for different sizes of K (from 2 to the number of test-orders), and we find which size K results in the highest silhouette score [39], an indicator of how good the clustering is. The column "# Clusters (Random)" in Table 1 shows the number of clusters computed this way across the randomly generated test-orders. We see that each module has at least two clusters, all with silhouette score higher than 0.5, indicating the clustering is decent. Moreover, there are five modules where the number of clusters is seven or more, indicating a wide variety of runtimes across different test-orders for these modules.

To better understand some reason for why there can be different runtimes per test-order, we looked into the relative ordering of test classes across the different test-orders for the module in project `apache/incubator-dubbo` (Figure 1), as the the results for this module (M2) show the greatest variance in runtimes between test-orders. We find that some test classes run longer when specific other test classes run first, despite having the same test outcomes. For example, `ChangeTelnetHandlerTest` takes about 1 second to run when it runs before `ExplicitCallbackTest`, but it takes about 40 seconds when it runs after. Upon further investigation, we found that tests in `ExplicitCallbackTest` add elements to a shared global `Map`. The tests in `ChangeTelnetHandlerTest` would use this shared `Map`, but they first clear the contents of this `Map` to ensure proper test independence. While this action ensures the outcome remains the same regardless of ordering between these two

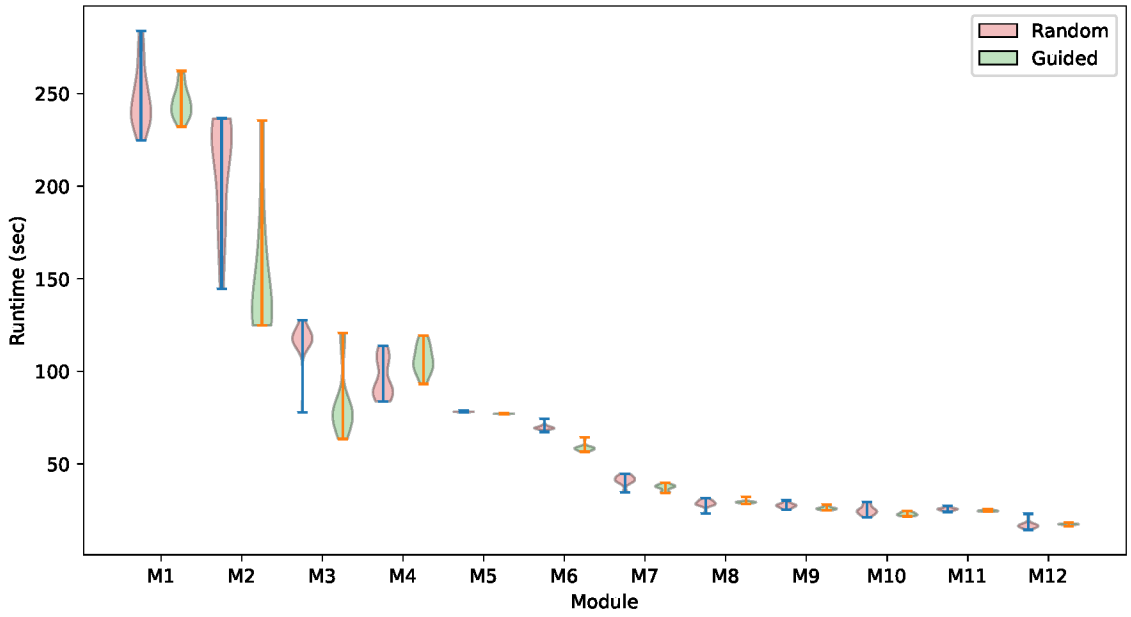

Figure 2: Variance in runtimes between test-orders.

test classes, the relative ordering has an effect on runtime, because the `ChangeTelnetHandlerTest` has to spend more time clearing out the `Map` if the `ExplicitCallbackTest` added those elements.

3 GUIDED SEARCH

Based on our inspection, we intuit that we can guide a search for the fastest test-order based on which relative orderings between test classes may result in speeding up runtime. We propose a guided search process that relies on both clustering of test-orders with similar runtimes and constraint solving to enforce perceived helpful relative orderings between test classes.

Figure 3 shows pseudocode representing this clustering and constraint solving process. We start by randomly generating an initial set of S different test-orders (Line 11); for our initial experiments, we choose S to be 6. We run the test-orders five times each to collect a distribution of runtimes. Our intuition is that we can construct a better test-order than the ones we already observed by preserving any relative orderings of test classes among the fastest test-orders and reversing any relative orderings of test classes among the slowest test-orders. To determine which test-orders are the fastest and slowest, we use K-means clustering to cluster test-orders based on runtime (Line 19). We determine the best size K by iterating through all possible K values (from 2 to the number of test-orders) and choosing the one that results in a clustering with the highest silhouette score. If that cluster has a silhouette score of less than 0.5, we find that cluster to be not very good, and so we simply generate a new random test-order for this iteration (Line 20).

We take the cluster with the fastest test-orders and the cluster with the slowest test-orders, and we compare the relative orderings of test classes between the test-orders in both clusters. For the fastest test-orders, we observe which pairs of test classes have the same relative orderings among all test-orders. We do the same for the slowest test-orders in the other cluster. We construct a set of constraints where we preserve the same relative orderings of test classes in-common among the fastest test-orders and another set of constraints where we reverse the relative orderings of test classes in-common among the slowest test-orders. For example, if test class `T1` comes before `T2` in all the fastest test-orders, we output a constraint represented as a tuple `(T1,T2)`, indicating that in the test-order we want to generate, `T1` must come before `T2`. Conversely, if we observe in all of the slowest test-orders that `T3` comes before `T2`, we output a constraint represented as a tuple `(T2,T3)`, which is the reverse of the relative ordering we observe from the slowest test-orders, and we want to generate test-orders where `T2` comes before `T3`. We remove any contradictions, i.e., both the fastest and slowest test-orders have some relative ordering of test classes in-common, suggesting this relationship to be irrelevant to runtime differences between test-orders. If we get an empty set of constraints (Line 32), we simply generate a new random test-order. Otherwise, we apply a "mutation" operator (Line 29), inspired by evolutionary algorithms [32] to enable more diversity in results, by reversing each constraint with a set probability (the inverse of the size of the set of constraints).

We use these constraints to formulate a constraint solving problem, where the goal is to assign positions to all test classes within the test-order while following the constraints of which test class needed to be positioned before another. We use state-of-the-art constraint solver Z3 [1, 8] to solve for these constraints and generate a

```
1  # OGOrder: arbitrary test suite order
2  # S: size of initial set
3  # N: # of random orders to generate and run
4  def random_cluster_and_solve(OGOrder, N):
5    # Run OGOrder and save its test outcomes
6    OG_test_results = run_order(OGOrder)
7    order_to_runtimes = collections.OrderedDict()
8    order_to_runtimes[OGOrder] = OG_test_results
9
10   # Create and run initial set
11   while len(order_to_runtimes) < S:
12     new_order = generate_random_order(OGORder)
13     new_order_results = run_order(new_order)
14     if new_order_results == OG_test_results:
15       order_to_runtimes[new_order] = new_order_results
16
17   # Cluster, solve and run
18   while len(order_to_runtimes) < N:
19     best_n_clusters, score = find_kmeans_n_clusters(
           order_to_runtimes)
20     if score < 0.5: # If we cannot cluster
21       new_order = generate_random_order(OGORder)
22     else:
23       # Cluster using Kmeans
24       fast_set, slow_set = find_fast_and_slow_sets(
             order_to_runtimes, best_n_clusters)
25       # Extract constraints
26       constraints = extract_constraints(fast_set,
             slow_set)
27       if len(constraints) > 0:
28         # Mutate constraints
29         mutated_constraints = mutate(constraints)
30         # Generate a new order using Z3 solver
31         new_order = generate_using_constraints(OGOrder,
               mutated_constraints)
32       else:
33         new_order = generate_random_order(OGOrder)
34     new_order_results = run_order(new_order)
35     if new_order_results == OG_test_results:
36       order_to_runtimes[new_order] = new_order_results
37
38   return order_to_runtimes
```

Figure 3: Clustering and constrains solving pseudocode.

new test-order (Line 31). If at any point when we generate a new test-order and the test outcomes are not consistent with previous outcomes, we throw the test-order away and generate a new one. We rerun test-orders five times to obtain a distribution of runtimes.

Figure 2 shows the spread of runtimes across different test-orders generated by this guided search, in the green-colored plots. We observe for some modules that the distribution of most of the runtimes of test-orders generated using the guided search bunch up at the bottom of the violin plot, namely for M1, M2, and M3. This characteristic means that most of the test-orders are fast, which suggests that the search is being guided towards the faster test-orders. We also observe that for the modules M5, M7, M8, M9, M10, M11, and M12, the distribution of the runtimes from the guided search mostly overlaps with the distribution from random generation. As such, both random test-orders and guided search test-orders have similar runtime results for these modules.

In Table 1, we show under column "# Clusters (Guided)" the number of clusters across the guided search test-orders, similar to how we compute the number of clusters across the randomly generated test-orders (Section 3). We observe that 5 modules have the same number of clusters and 3 modules have more clusters.

We would also like to see whether the guided search can find test-orders that have faster runtimes sooner, i.e., trying out fewer test-orders to find the fastest test-order. If guided search is faster at finding the fastest test-order, we could run that search fewer iterations. The column "Fastest rank" in Table 1 shows at which iteration does the guided search find a test-order that is in the same cluster as the fastest cluster of test-orders. We notice that 3 modules have their fastest test-order found later after the randomly generated initial set of test-orders (rank > 6). This result suggests that the guidance from the constraints are useful in helping construct faster test-orders. We also find that the guided search finds faster test-orders than random generation (both the guided search and random generation generate the same number of test-orders), for modules M3 and M6. These results suggest that their tests' runtimes are likely affected by dependencies between tests, so our approach at leveraging orderings between tests helps find faster test-orders. Unfortunately, the approach is not as effective for other modules.

In Table 1, we also show under column "Avg. gen. time (s)" the average time in seconds the guided search takes to generate a test-order, with overall average time of 17.34 seconds. Modules with a high number of test classes generally take more time to generate test-orders due to the large number of constraints. The "# Rnd. gen. orders" column shows the number of random test-orders the guided search generated, excluding the 6 initial ones. In most cases, guided search does not resort to generating a random test-order.

4 CONCLUSIONS AND FUTURE WORK

In this work, we investigate the effects of running tests in different test-orders on the runtime. Our study on 12 open-source Java projects' test suites show significant variance in runtime between test-orders, where the slowest test-order is on average 31.17% slower than the fastest test-order. We develop a new approach that generates faster test-orders by clustering existing test-orders based on runtime and constructing new test-orders based on in-common relative orderings between test classes among the test-orders in the clusters. This approach was effective at generating faster test-orders than random generation for two projects.

In the future, we plan on developing an improved approach to search for the fastest test-orders, with the goal to find the fastest test-orders more efficiently than simply randomly generating test-orders, as well as to generate test-orders that are even faster. We plan on investigating further the various reasons for why tests run faster in certain test-orders, such as due to JIT optimizations or memory usage during testing, beyond the dependencies between tests reason that was the focus of our proposed approach.

ACKNOWLEDGMENTS

This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2145774 and CCF-2217696, as well as the Jarmon Innovation Fund.

REFERENCES

[1] 2017. Z3 Theorem Prover. https://z3.codeplex.com.
[2] 2024. GitHub Actions. https://github.com/features/actions.

[3] 2024. maven-surefire. https://github.com/TestingResearchIllinois/maven-surefire.

[4] 2024. Travis-CI. https://travis-ci.org.

[5] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *International Conference on Software Engineering*. 106–115.

[6] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667.

[7] T. Y. Chen and M. F. Lau. 1998. A new heuristic for test suite reduction. *Journal of Information and Software Technology* 40, 5-6 (1998), 347–354.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[9] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.

[10] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.

[11] Arnaud Gotlieb and Dusica Marijan. 2014. FLOWER: Optimal test suite reduction as a network maximum flow. In *International Symposium on Software Testing and Analysis*. 171–180.

[12] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *International Symposium on Software Reliability Engineering*. 112–122.

[13] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-demand test suite reduction. In *International Conference on Software Engineering*. 738–748.

[14] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 312–326.

[15] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. 2001. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering* 27, 3 (2001), 248–263.

[16] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *International Conference on Software Engineering*. 523–534.

[17] Dennis Jeffrey and Neelam Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *IEEE Transactions on Software Engineering* 33, 2 (2007), 108–123.

[18] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *International Conference on Automated Software Engineering*. 233–244.

[19] James A. Jones and Mary Jean Harrold. 2001. Test-suite reduction and prioritization for modified condition/decision coverage. In *International Conference on Software Maintenance*. 92–102.

[20] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *International Symposium on Software Testing and Analysis*. 101–111.

[21] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*. 312–322.

[22] 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 *International Symposium on Foundations of Software Engineering*. 583–594.

[23] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *International Conference on Automated Software Engineering*. IEEE, 949–954.

[24] Chengpeng Li, M Mahdi Khosravi, Wing Lam, and August Shi. 2023. Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests. In *International Symposium on Software Testing and Analysis*. 627–638.

[25] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 33, 4 (2007), 225–237.

[26] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *International Conference on Software Engineering*. 535–546.

[27] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *International Symposium on Foundations of Software Engineering*. 559–570.

[28] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *International Conference on Software Engineering, Software Engineering in Practice*. 91–100.

[29] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[30] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *Programming Journal* 4 (2020), 12:1–12:32.

[31] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*. 233–242.

[32] Melanie Mitchell. 1998. *An Introduction to Genetic Algorithms*. The MIT Press.

[33] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. In *International Symposium on Software Testing and Analysis*. 249–259.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[35] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *International Symposium on Software Testing and Analysis*. 324–336.

[36] Gregg Rothermel and Mary Jean Harrold. 1994. A framework for evaluating regression test selection techniques. In *International Conference on Software Engineering*. 201–210.

[37] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology* 6, 2 (1997), 173–210.

[38] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.

[39] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[40] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *International Conference on Software Engineering*. 268–279.

[41] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *International Symposium on Foundations of Software Engineering*. 246–256.

[42] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *International Symposium on Software Testing and Analysis*. 84–94.

[43] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 187:1–187:29.

[44] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555.

[45] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjørner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *International Conference on Software Engineering*. 689–699.

[46] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *International Symposium on Software Reliability Engineering*. 228–238.

[47] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *International Symposium on Software Testing and Analysis*. 97–106.

[48] Panagiotis Stratis and Ajitha Rajan. 2018. Speeding up test execution with increased cache locality. *Journal of Software Testing, Verification and Reliability* 28, 5 (2018), 1–17.

[49] 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.

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

[51] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *International Conference on Software Engineering*. 192–201.

[52] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An empirical study of JUnit test-suite reduction. In *International Symposium on Software Reliability Engineering*. 170–179.

[53] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*. 385–396.

[54] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *International Conference on Software Engineering*. 430–441.