Regression Test Case Prioritization – A Contribution-Based Approach

CISC 879 - Software Testing and Maintenance

12/9/2008

Abstract

Regression test case prioritization techniques have traditionally been studied as a function of objective metrics such as code coverage and fault proneness, which require much data analysis and computation from software release to release. Moreover, such techniques have traditionally been evaluated as a function of fault detection effectiveness. Consequently, not only applying the techniques is an expensive exercise, the results focus on fault detection rates rather than release-readiness, a more practical interpretation of software quality. We therefore propose to approach regression test case prioritization with a completely different tack: We propose a novel approach to prioritize based on subjective but inexpensive, one-off human assessment of test cases' contribution to predicting release-readiness, thereby eliminating the intensive data analysis and computation required to begin the prioritization process. Further, to align closer to industry software processes, we define the effectiveness of a prioritization as the time required to pronounce a software application release-ready. We believe these novel prioritization and evaluation methods will bring a new perspective to the state-of-the-art.

1 Introduction

Due to the finite amount of time and resources available to any tester, the goal of prioritizing test cases is to execute the test cases that best assess the quality of a software application earliest. Test case prioritization is particularly relevant to regression testing for a number of reasons. A tester's primary focus for a given software release is that release's new features, pushing regression testing to a secondary focus. At the same time, regression test cases are often reused or modified from previous releases. As the software migrates from one release to another, the number of regression test cases piles up, requiring a disproportionately large amount of the tester's attention. With prioritization, the tester can determine which regression test cases out of a large pool of candidates will best predict the software's quality, and mitigate risks by executing those test cases first.

Although the incentives for regression test case prioritization are high, the prioritization process is an equally expensive and complex task due to the myriad of factors that impact test case priority. Fault proneness, code coverage at different levels of granularity, data flow, perceived user importance – to name only a few – all affect how test cases can be prioritized within a test suite.

Much prior work has proposed various techniques to prioritize test cases [13, 6, 14, 3, 12]. Currently there is no single, most superior technique; each technique juggles between cost and effectiveness. There are three common threads amongst the state-of-the-art, however.

Firstly, before the prioritization task can begin, data relating to the test cases' power to predict software quality must be set up. To the authors' knowledge, apart from one technique [17], all such data setup requires significant code analysis, the tracking of fault detection and input data set histories, and model building. The cost of preparation alone is steep.

Secondly, recall that regression testing depends on materials (test cases and historical data) built in previous release cycles. Some prioritization techniques specific to regression testing [10, 12] require that the aforementioned data setup be re-calculated for every new release, incorporating the latest historical data from the last release. As the software program and the number of regression test cases become larger and larger, the cost of this repeated data setup also becomes larger.

Thirdly, to evaluate the effectiveness of test case prioritization techniques, researchers adopt the rate of fault detection as the weapon of choice. From a research point of view, this metric provides a measurable and deterministic medium for defining and comparing performance. From the point of view of a software project manager in the industry, however, the rate of fault detection is but one factor determining the quality of the software product. Most frequently, a software release is bound by both quality *and* time, in that the software must be released by a certain deadline as long as it meets minimum quality standards. In practice, the project manager must decide when the software is ready for release, thus negotiating between quality and time constraint. In fact, it is common practice to release a software with unresolved defects as long as those defects are not critical. Therefore we argue that to evaluate a prioritization technique more closely to industry practice, a metric based on time to release-readiness is more applicable than one based on fault detection alone.

The three observations above summarize three of the limitations in the state-of-the-art. In this proposal, we present a novel prioritization technique as well as a novel evaluation technique that address these three limitations. We hypothesize that a test case's power to predict software quality can be captured by *contributing factors* such as the amount of features covered by the test case and the perceived importance of those features. We propose to leverage human domain experts to provide quick, minimal-effort estimates of *contribution scores*, which can be combined into an overall *prioritization score*. Such human assessments need only take place once for each test case, eliminating the need to re-calculate the preparation data set for each release. We also hypothesize that time elapsed is a viable metric for evaluating a prioritization technique's effectiveness. The best prioritization is simply one that can verify earliest that a software has reached acceptable quality standards and is cleared for release.

1.1 Overview of research goals

To the authors' knowledge, our proposed prioritization technique is the first attempt that prioritizes purely based on subjective, human assessments. It is also the first attempt that completely foregos fault detection effectiveness and evaluate the technique purely based on time budget. The goals of our proposed research are therefore the following:

- Establish a prototype technique for prioritizing regression test cases not based on data or code analysis, but on one-off human assessments.
- Establish a list of test case contributing factors for which human domain experts can provide assessments with minimal effort. And from this list of contributing factors, develop a method to derive a prioritization score.
- Investigate that evaluating regression test case prioritization based on time budget alone produces valid results that accurately predict software quality.

2 Background

As a software application evolves from one version to the next, testing is performed to verify that existing as well as new features all function correctly. Regression testing is the portion of testing targeting existing features. Typically, regression test cases are recycled from old test cases targeting then-new features in the previous software version. As the software moves through multiple release cycles, the number of regression test cases compounds quickly. While testers' primary focus is the verification of new features, the cost of completing secondary regression testing is often disproportionately high. Moreover, testers have only a finite amount of time and resources to complete regression testing before the software must be released. There is thus a critical need to execute regression testing as effectively and efficiently as possible.

Most studies in the literature measure regression testing effectiveness in terms of fault detection, while others measure the amount of resources consumed. Regardless of the metric used, testers seek to organize regression test cases so as to optimize that metric. One technique is to *select* a subset of test cases to execute, intentionally opting out on those test cases that contribute least to the effectiveness metric. A second technique is to *prioritize* test cases in some order, intending to execute all the test cases thus prioritized. These two techniques may be deployed separately or in conjunction. The focus of this proposal, however, is on test case prioritization specifically for regression testing.

A well prioritized regression test suite strikes an optimal balance between prioritization cost, test execution cost, and test effectiveness. It identifies test cases that should be executed earlier so that faults are detected as early as possible and code is covered as early as possible.

2.1 State of the Art

Regression testing and test case prioritization are two areas that have been heavily studied, often as separate research problems and less commonly as a single, combined problem. To our knowledge, Srikanth and Williams' work [17] is the closest to our proposed research in that both techniques are based on subjective, estimated factors. More importantly, both techniques reflect industry software project practices more closely than other research-focused works. Their prioritization technique is based on priority of requirements, perceived code complexity, and customer satisfaction. Not surprisingly, the most influential ingredient in their technique is customer satisfaction. Their evaluation approach differs from our proposed one as they use the traditional fault detection effectiveness as the success metric. Our proposed evaluation method steps away from this mainstream metric to consider release readiness as the primary success metric. Although their work addresses both regular and regression test cases, we draw inspiration from their use of user-perceived contributing factors.

A number of other test case selection techniques specific to regression testing fall on the mainstream that is based on objective performance-oriented constraints and evaluate success by fault detection effectiveness. While these techniques and their evaluations may not align as well to industry software release processes, concrete, quantifiable metrics allow more objective and more measurable results. Walcott et. al [18] constrain the prioritization algorithm by time available, while Kim and Porter [7] further constrain by both time and resources. Park et. al [10] evaluate the performance of a prioritization by Average Percentage of Fault Detected per Cost (APFDc). Where one performance aspect of a prioritization is maximized, trade-offs often result in other aspects. For instance, achieving low time and resource consumption incurs a loss in fault detection effectiveness. Of all performance-oriented constraints targeted by these techniques, no single constraint is a clear leader that dominates a prioritization. To our knowledge, there is no single prioritization technique that can optimize *both* resource consumption *and* fault detection effectiveness.

Algorithms and models have also been applied to prioritize regression test cases. Since test case prioritization can be an arbitrarily complex problem spanning both human perception and objective metrics, some researchers opt to simplify the problem by abstracting it as a theoretical model. Data-flow model [15], Combinatorial Interaction Testing (CIT) [11, 12], Single Value Decomposition (SVD) [16], genetic algorithms [18, 8], greedy algorithms and metaheuristic algorithms [8] have all been shown to improve fault detection effectiveness. Mirarab and Tahvildari [9] propose to prioritize non-regression test cases via a Bayesian network. However, as with performance-oriented prioritization techniques, trade-offs are inevitable; achieving higher fault detection effectiveness comes at a significant cost of building the model. More importantly, abstracting the prioritization problem to algorithm- and model-levels thus far does not yield significantly more effective fault detection rates than constraint-based techniques.

Cross-cutting most test case prioritization techniques – for regression and other types of test cases – is the *basis* on which the techniques operate upon. The most popular approach prioritizes based on code coverage. Different levels of granularity have been studied by many researchers, from statement [6], branch [13, 5], block [1, 18], to function [4, 6, 1, 10]. Fault proneness of the code exercised in a test case can also predict the fault detection effectiveness of that test case [13, 4, 5, 9, 10]. Code modification history [7, 3, 10] and code distribution [2] can be tracked and fed to a prioritization technique. Other works prioritize test cases based on data-flow [15] and input grouping [14].

Overall, virtually all techniques in the state-of-the-art prioritize test cases based on statistics such as code coverage and fault proneness, and evaluate based on fault detection effectiveness. The prevalent, mainstream approach that relies heavily on pre-prioritization data analysis and post-prioritization evaluation via fault detection spurs us to investigate an alternative approach.

3 Challenges and research goals

Upon reviewing the state-of-the-art in the previous section, we identify a few challenges that are unaddressed or as yet inadequately addressed by the testing research community. In this section, we describe these challenges and respond by introducing the goals in our proposed research that counter these challenges.

3.1 Heavy overhead

Test case prioritization, by definition, favors a test case over another based on some factor or factors. In our proposed technique, we refer to these factors as *contributing factors*. A contributing factor may be concrete, such as test case length, code coverage, data flow, and historical fault proneness, or abstract, such as perceived code complexity and fault severity. In general, concrete contributing factors are easily quantifiable but can only be quantified via significant analysis and bookkeeping. Abstract contributing factors, on the other hand, may be objective and less quantifiable but are usually relatively inexpensive to obtain.

The vast majority of the prioritization techniques in the state-of-the-art operate exclusively on concrete contributing factors. The corollary is that before prioritization can even begin, much pre-processing analysis must be done. During the prioritization process, this data-centric approach must also take all the data into account. Oftentimes, this significant time and resource overhead before and during the prioritization process has not been adequately emphasized. We argue that a large overhead eclipses the benefits of the resulting prioritization.

To counter this potentially benefit-diminishing overhead, we propose to take a direction opposite to the mainstream: prioritize exclusively by abstract contributing factors. Our proposed research leverages human domain knowledge to assess the objective, perceived *impact* of a test case, and prioritize the test case with the most impact first. With knowledgeable domain experts, such human assessment is relatively inexpensive to obtain as the test cases do not need to be correlated with any concrete data. If such assessment produces a similarly effective prioritization, our proposed research will significantly lower the overall cost of performing the prioritization process.

3.2 Repetitive data setup

Some previous works (for example [8]) that focus on regression test case prioritization perform the prioritization on only one version of the software. Other previous works apply a prioritization to multiple software versions (for example [10, 12]). However, for every new release, these techniques require that the setup of contributing factors be completely recomputed based on new data provided by the previous release. A new round of prioritization can then proceed with the renewed preparation data. As a result, every round of prioritization incurs the cost of repeating the entire data setup exercise.

We therefore propose to minimize data setup costs by investigating the alternative – a *one-off* data setup. In a one-off data setup, initial data needs only be established once and can be re-used in later software releases. To that end, we propose to gather rankings, or *contribution scores*, of abstract contributing factors such as feature coverage and test case complexity that are relatively independent from software changes from release to release. These contribution scores may need to be updated upon a major re-make of the software. In general, however, once initially assessed, contribution scores remain applicable in newer releases as the same test cases are used as regression test cases, thus achieving a one-off data setup.

3.3 Incomplete evaluation metrics

The *de facto* standard of evaluating a test case prioritization is fault detection effectiveness. Ever since Rothermel et. al introduced the rate of fault detection or Average Percentage of Fault Detected (APFD) in [13], most papers use this metric in their evaluations. Park et. al enhances this metric to take into account related cost and severity of the defects, and introduced Average Percentage of Fault Detected per Cost (APFDc) [10]. However, the success of a prioritization is still pinned on the very focus of fault detection.

Consider a real-life software project. In a typical release cycle, test cases are executed and re-executed as defects are discovered and fixed. Towards the end of the cycle, the number defects yet to be resolved diminishes. Moreover, the defects that remain unresolved at that point are generally minor defects. Industry best practice recommends that no software product be released until all critical defects or "show stoppers" are fixed. In other words, in a real-life project, test cases may be run for the express purpose of *not* finding defects. When the defect queue contains no critical defects and a small number of minor defects, the software is deemed ready for release. In our proposed research we say the software is *release-ready*. At different times in the software release cycle, fault detection effectiveness may or may not be the primary factor in predicting the success of a test case prioritization.

How should one evaluate the success of a prioritization then? Since evaluation by fault detection alone

does not completely reflect industry software release practice, we consider the amount of *time* that elapses until the software is release-ready. Specifically, this time span constitutes not only the time required to perform the prioritization, but also the time required to execute the prioritized test cases until the software becomes release-ready. Our research proposes to evaluate the success of a prioritization by tracking this amount of time required to produce a release-ready software.

4 Proposed research

4.1 Contributing factors

The central component of our proposed research is the use and selection of contributing factors. Loosely defined, a contributing factor predicts how well a test case will contribute to making the software release-ready as soon as possible. As mentioned above, it is critical that we select contributing factors that are:

- abstract so as to minimize data setup overhead, and
- independent of software changes from release to release so as to facilitate one-off data setup.

To that end, we propose to investigate the following contributing factors:

- Feature coverage: Does the test case involve one, few, or many features?
- Importance of feature: Is the feature being tested important? Do users actually use that feature?
- Criticality of feature: How critical is it that the feature being tested passes the test case? If the feature fails the test case, is the software still release-ready?
- Fault proneness: How likely, possibly based on experience from a previous regression test cycle, will this test case reveal faults? Notice that this question asks for a domain expert's "gut feel," unlike other existing techniques where fault proneness is a concrete contributing factor that requires detailed defect analysis.
- Test case variation: How many different input data sets are applicable to this test case? Intuitively, the more data sets applicable, the more likely the test case will cover corner cases and thus reveal faults and subtract from release-readiness.
- Test case dependence: How early in the software usage process will a user require this feature? Intuitively, if a test case tests a feature early in the usage process but fails, other test cases dependent on the failed feature cannot begin testing.
- Test case complexity: How complex is the test case? Intuitively, the more complex the test case, the more likely it is to reveal faults.
- Test case length: How long is a test case? In other words, how many steps does it take a tester to execute this test case? Intuitively, longer test cases will touch upon more features.
- Execution time: How much time does it take to execute a test case? Notice this question is related to but not the same as test case length.

In practice, every software project has some domain experts who are well versed in the architecture, functionality and limitations of the software. When such an expert is presented with a test case, the expert can quickly nominate a 1-to-5 contribution score to most contributing factors.

To demonstrate the feasibility of requesting minimal-effort human input, consider the following example. Given a test case that generates a monthly sales report, the expert can quickly estimate that the test case only covers one feature (monthly sales report), has limited user impact (only the sales reporting team needs monthly sales reports), has little dependence on other test cases (the monthly report is a background batch job that depends solely on correct data), but is very complex (due to all the data aggregation and tax calculations).

4.2 Contribution mix and prioritization score

After collecting contribution scores of various contributing factors, the data needs to be condensed into a single, normalized number, or *prioritization score*. When each test case is assigned a prioritization score, the prioritization process simply sorts the test cases according to their prioritization scores.

To derive the prioritization score, we propose to use a statistical regression model to fit the collection of contributing factors, or *contribution mix*, to known results of a test suite as follows. Regression testing, by definition, takes place in the second, third, and further releases of a software. Regression testing is not performed in the original release of the software. However, regression test cases are, in general, taken from or modified from the test cases used in the orig-

inal release. As long as we know the defects found in the first release, their severity, and which test case found them, we can experiment with contribution mixes until we find that one mix that minimizes the time to release-readiness.

We expect that our research may reveal that not every contributing factor proposed delivers sufficient impact to be included in the contribution mix.

Notice also that our proposed technique implies that every software project needs to generate its own contribution mix. We hypothesize that, given the proposed contributing factors are generic across domains and software projects, a contribution mix from one project may well apply to other projects. However, we leave the investigation of this follow-up problem to future research.

4.3 Time tracking

As mentioned before, the overall goal of our research is to produce a prioritization technique capable of minimizing the time required to bring a software to release-readiness. In practice, this overarching time span includes:

1. Time to set up preparation data prior to prioritization
2. Time to perform the actual prioritization
3. Time to execute the prioritized test cases
4. Time to fix defects found
5. Time to re-execute any failed test cases

To isolate the time spent relevant to our proposed research while keeping as close as possible to industry software release practice, we plan to include only items 1, 2, and 3. In effect, we assume that each test case needs to be executed only once. This assumption is far from realistic. However, we believe as a first step in this novel evaluation approach, the time span thus tracked will still provide an indicative result of how effective our prioritization technique is.

5 Evaluation plan

We plan to conduct a multi-staged experiment to evaluate the efficiency and effectiveness of the proposed technique. The materials of the experiment will be derived from an open source project with at least two released versions. Known test cases, times required to execute the test cases, defects and their associated severities, and correlations between defects and test cases related to the first release will become the training materials for developing the best-performing contribution mix and thus prioritization score calculation. This prioritization score calculation will then be applied to the second release to simulate a regression test scenario. The stages of the experiment are as follows:

Stage 1: Gather contribution scores. We intend to recruit software testers who tested the selected software application to become domain experts for the experiment. Since the proposed prioritization technique relies heavily on the accuracy of human domain knowledge, it is crucial that the experiment harvest such knowledge from actual members of the selected software application. These experts will be supplied with the set of known test cases, and asked to estimate contribution scores for all aforementioned contributing factors for every test case according to their subjective judgment. We expect that most of these test cases had been executed by one or more of the recruited experts, further bolstering the accuracy of their nominated contribution scores. Using human subjects close to the test cases should not invalidate the contribution scores, since in industry test case prioritization are almost exclusively carried out by testers themselves.

Stage 2: Train for best-fit method to calculate best-performing prioritization scores. In this stage, we plan to derive a method to calculate prioritization scores from a contribution mix that produces a test case prioritization capable of bringing the software to release-readiness earliest. To arrive at the best-fit method, we intend to apply statistical regression model to the contribution mix so as to extract the subset of contributing factors and their weights. Using these prioritization scores, test cases from the first release are sorted.

This stage of the experiment will not involve actual execution of the test cases. Since the defects, their severities, and their association with test cases are known, we can simulate the execution of the prioritized test cases one by one until all critical defects are revealed.

Stage 3: Test prioritization scores against unseen test suite. Apply the prioritization scores calculated in stage 2 to the second version of the software application. Since the execution times for the test cases are known, similar to the previous stage, we can simulate the testing process and record the virtual times required to execute the test cases, effectively simulating a regression test cycle. After the

virtual execution of each test case, we will examine the defects and their severities associated with that test case, as well as the overall list of defects associated with the software application. By tracking the test cases and their associated defects, we will be able to conclude whether all critical defects have been revealed so that the software is release-ready, or if further (simulated) testing is required.

Stage 4: Compare test results with baseline results. This stage of the experiment will address two research questions.

Firstly, does prioritization help at all in verifying sooner that the software is release-ready? To answer this question, we plan to simulate the virtual testing process using random prioritization.

Secondly, do all the contribution scores and contribution mix provide a better test case prioritization? Are all the human input and domain knowledge necessary? To answer this question, we plan to simulate the virtual testing process using a naive prioritization based on test case length. Intuitively, the longer a test case, the more code and functionality the test case touches. Therefore obtain a prioritization where the longest test case is prioritized earliest.

By comparing the time required for all three techniques to verify the release-readiness of the software application, we will be able to evaluate the merits of the proposed contribution-based technique.

6 Summary of foreseen contributions

We anticipate that our proposed research will contribute to the state-of-the-art in the following areas:

- Deliver a practical, contribution-based prototype that prioritizes regression test cases.
- Confirm the hypothesis that domain knowledge based on human judgment alone can provide crucial directions on how to best prioritize regression test cases.
- Develop a one-off data setup alternative to current prioritization techniques that require repeated data setup for each software release.
- Develop a list of domain knowledge-based contribution factors that, in combination, can accurately predict a test case's power to bring a software to release-readiness.
- Present a novel evaluation metric based on the time spent to bring a software to release-readiness.

References

[1] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 113–124, Nov. 2004.

[2] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, pp. 33–70, March 2006.

[3] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 733–752, Sept. 2006.

[4] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, (New York, NY, USA), pp. 102–112, ACM, 2000.

[5] S. Elbaum, D. Gable, and G. Rothermel, "Understanding and measuring the sources of variation in the prioritization of regression test suites," *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pp. 169–179, 2001.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 159–182, Feb 2002.

[7] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, (New York, NY, USA), pp. 119–129, ACM, 2002.

[8] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *Software Engineering, IEEE Transactions on*, vol. 33, pp. 225–237, April 2007.

[9] S. Mirarab and L. Tahvildari, "An empirical study on bayesian network-based approach for test case prioritization," *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 278–287, April 2008.

[10] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*, pp. 39–46, July 2008.

[11] X. Qu, M. Cohen, and K. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 255–264, Oct. 2007.

[12] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, (New York, NY, USA), pp. 75–86, ACM, 2008.

[13] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 179–188, 1999.

[14] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 3, pp. 277–331, 2004.

[15] M. J. Rummel, G. M. Kapfhammer, and A. Thall, "Towards the prioritization of regression test suites with data flow information," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1499–1504, ACM, 2005.

[16] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, pp. 81–90, Nov. 2007.

[17] H. Srikanth and L. Williams, "On the economics of requirements-based test case prioritization," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–3, 2005.

[18] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, (New York, NY, USA), pp. 1–12, ACM, 2006.