The Journal of Systems & Software 175 (2021) 110905

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Fast and accurate incremental feedback for students' software tests using selective mutation analysis☆

Ayaan M. Kazerouni [a,*], James C. Davis [b], Arinjoy Basak [c], Clifford A. Shaffer [c], Francisco Servant [c], Stephen H. Edwards [c,*]

[a] Department of Computer Science and Software Engineering, California Polytechnic State University, United States of America
[b] Department of Electrical and Computer Engineering, Purdue University, United States of America
[c] Department of Computer Science, Virginia Tech, United States of America

ARTICLE INFO

Article history:
Received 29 June 2020
Received in revised form 23 November 2020
Accepted 4 January 2021
Available online 8 January 2021

Keywords:
Software testing
Mutation analysis
Software engineering education
Automated assessment tools

ABSTRACT

As incorporating software testing into programming assignments becomes routine, educators have begun to assess not only the correctness of students' software, but also the adequacy of their tests. In practice, educators rely on code coverage measures, though its shortcomings are widely known. Mutation analysis is a stronger measure of test adequacy, but it is too costly to be applied beyond the small programs developed in introductory programming courses. We demonstrate how to adapt mutation analysis to provide rapid automated feedback on software tests for complex projects in large programming courses. We study a dataset of 1389 student software projects ranging from trivial to complex. We begin by showing that although the state-of-the-art in mutation analysis is practical for providing rapid feedback on projects in introductory courses, it is prohibitively expensive for the more complex projects in subsequent courses. To reduce this cost, we use a statistical procedure to select a subset of mutation operators that maintains accuracy while minimizing cost. We show that with only 2 operators, costs can be reduced by a factor of 2–3 with negligible loss in accuracy. Finally, we evaluate our approach on open-source software and report that our findings may generalize beyond our educational context.

© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Software testing is the primary approach for evaluating the correctness of computer software in practice. It is thus critical that software engineering teams follow strong software testing practices. Unfortunately, many software engineers have inadequate training in testing (Lethbridge, 2000; Carver and Kraft, 2011; Radermacher and Walia, 2013). To address this shortcoming, educators have begun to incorporate software testing into the software engineering curriculum (Jones, 2000; Spacco and Pugh, 2006; Aniche et al., 2019), including introductory programming courses (Edwards, 2004). These educators provide students with feedback not only about their software, but also about their test suites.

To be effective, feedback on student test suites should meet three goals. First, it should provide a *reliable test adequacy criterion*, so that students are assessed in a meaningful way. Second, it should be *amenable to incremental feedback* in order to guide students throughout the development cycle. Third, it should be *fast to compute* to support student learning. Speediness also ensures that an educational institution's centralized Automated Assessment Tool (AAT) (Pettit and Prather, 2017) is not overloaded.

Many approaches have been proposed for evaluating student test suites (Goldwasser, 2002; Edwards, 2004; Aaltonen et al., 2010). *Code coverage* measures can provide incremental feedback and are quickly computed, but they set a low bar for adequacy: a test may cover code without ensuring its correctness (Myers et al., 2011; Edwards et al., 2009; Aaltonen et al., 2010; Inozemtseva and Holmes, 2014). The *all-pairs* approach involves running every student's tests against every other student's code (Goldwasser, 2002), improving test adequacy but requiring relatively complete implementations (non-incremental). These approaches do not currently meet our goals (see Section 2.2).

Mutation analysis, proposed by DeMillo et al. (1978), is a promising alternative feedback approach. Mutation analysis is a fault-based test assessment technique in which small changes (*mutations*) are made to the target program, creating incorrect variants known as *mutants*. The different kinds of mutations that can be applied are called *mutation operators*. The adequacy of the

☆ Editor: [W. ERIC WONG].

* Corresponding authors.
E-mail addresses: ayaank@calpoly.edu (A.M. Kazerouni), davisjam@purdue.edu (J.C. Davis), arinjoyb@vt.edu (A. Basak), shaffer@vt.edu (C.A. Shaffer), fservant@vt.edu (F. Servant), edwards@cs.vt.edu (S.H. Edwards).

https://doi.org/10.1016/j.jss.2021.110905
0164-1212/© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

test suite is measured as the percentage of mutants caught by the test suite.

Mutation analysis mitigates the limitations of code coverage approaches and all-pairs approaches. Unlike code coverage, mutation analysis is a reliable adequacy criterion for student-written software tests (Aaltonen et al., 2010; Shams, 2015). Unlike all-pairs approaches, mutation analysis can be used to provide incremental feedback. However, it is well-known that mutation analysis is a computationally expensive approach in general (Jia and Harman, 2011). Previous studies have found mutation analysis to be a reliable adequacy criterion for student-written software tests, but they have not studied whether mutation analysis can be performed cheaply enough to provide timely feedback in an AAT.

Our goal in this paper is to evaluate the cost of running mutation analysis on undergraduate programming projects and to reduce it to the point where it is feasible to deploy in an AAT to provide students with rapid incremental feedback about the quality of their tests. We evaluate the effectiveness of our approach as we vary the complexity of the student projects under consideration, determining whether a scheme for projects in introductory courses (with relatively small programs) will work for those in subsequent courses (with larger and more complex programs). We conduct three studies:

- **Motivational.** We investigate the cost and effectiveness of existing approaches to mutation analysis when applied to students' software projects (Section 6). We consider the state of the art in mutation analysis: *comprehensive* mutation (all available mutants) and two *selective* mutation strategies: *sufficient* and *deletion* mutation.
- **Core.** We propose a novel approach to mutation analysis that is appropriate for use in a rapid response automated feedback context (Section 7). We do this by selecting a further-reduced subset of mutation operators through a statistical procedure.
- **Validation.** We conduct an additional study to validate our findings by running similar analysis, but using a separate corpus of real-world software (Section 8). We evaluate the effectiveness along with two measures of cost – the number of mutants, and the proportion of equivalent mutants – for our chosen subsets of operators.

Summary of findings. Our proposed approach offers superior cost-effectiveness trade-offs compared to the state of the art. But in our context, we found that there exists a set of two mutation operators sufficient to predict the mutation coverage achieved under the full set of mutation operators with R^2 of up to 0.94, depending on the size of the program under test. In other words, in terms of *reliability*, our techniques perform nearly as effectively as comprehensive mutation analysis. And in terms of *computational cost*, we halve the cost of the state of the art (mutation by deletion), reducing the total cost of test adequacy assessment from roughly 630 mutants per thousand lines of code (KSLoC) for deletion mutation to between 230 and 330 mutants per KSLoC, depending on the program under test. In our validation study, we found that these findings may generalize to real-world projects as well, suggesting potential benefits for practitioners as well as educators.

Our results indicate that using mutation operator subsets is an effective approximation of much more costly mutation analysis strategies for student code, and reduces the run-time impact to a point where it is feasible to apply automatically as students check their work during solution development. This will allow educators to provide interactive feedback within a reasonable response time, allowing students to iterate more quickly on submission cycles, while being held to a higher standard of test adequacy. Since mutation analysis is a more reliable adequacy criterion than commonly-used code coverage measures, this will help students to produce stronger test suites (and therefore, we hope, more reliable software). This paper contributes to software engineering education research and practice by showing how mutation operators can be applied in a cost-effective way to better assess the quality of student-written tests. Our work is thus an important step toward improving the pedagogy of software testing.

This paper's outline is as follows. We describe prior work related to evaluating test quality and mutation analysis (Section 2); define goals for speedy mutation-based feedback (Section 3); describe our research questions (Section 4) and study context (Section 5); and describe the methods and results for each research question (Sections 6–8). We close with a discussion of our results (Section 9), an assessment of threats to validity (Section 10), and a summary of our conclusions (Section 11).

2. Background and related work

We first introduce the desirable properties of test assessment criteria, then review prior work on the evaluation of software test suites, and finally summarize the state of the art in mutation analysis in the educational context.

2.1. Desirable properties for student test assessment

Our work is conducted in the context of computing education, where a metric for student test assessment should meet three goals: it should enforce a *strong test adequacy criterion*, it should permit *incremental feedback*, and it should return a *fast response*.

The most desirable feature for such a metric is that it impose a **strong test adequacy criterion**. A test adequacy criterion is a predicate that defines "what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested program" (Goodenough and Gerhart, 1975). Note that the focus of such a criterion is not the program, but rather the tests. The stronger the test adequacy criterion, the higher the standard to which a test suite is held. Criteria of test adequacy vary in their strength. For example, successful compilation is a weak test adequacy criterion, statement-based code coverage is a stronger criterion, and condition-based coverage is stronger still. The stronger the test adequacy criterion, the better the alignment between a test suite's *actual* strength and its strength as measured by the criterion. So, satisfying a weak test adequacy criterion does not guarantee thorough testing. Spacco et al. (2006) and Shams (2015) report that students' high code coverage scores were not correlated with bug-free software—this has also been observed in non-educational settings (Inozemtseva and Holmes, 2014).

Incremental feedback has been found to improve student learning outcomes (Black and Wiliam, 1998; Edwards, 2004), and is therefore a desirable quality for an assessment metric. Rather than deferring feedback until their final submissions, incremental feedback permits students to gauge their progress and identify their errors along the way. It is thus becoming standard practice in computer science education (Pettit and Prather, 2017).

Providing a **fast response** is valuable for pedagogical and practical reasons. Pedagogically, fast feedback (seconds, not minutes) has been found to improve student learning (Azevedo and Bernard, 1995; der Kleij et al., 2015). Performance is also of practical concern in an educational context, because student submissions are typically assessed on a single centralized server shared among all CS courses at an institution. These servers are known as Automated Assessment Tools (AATs), and include

Table 1
Test evaluation techniques commonly used in CS education and their strengths and weaknesses.

Technique	Ref.	Strong adequacy criterion?	Supports incremental feedback?	Fast response?
Code coverage	Spacco and Pugh (2006) Edwards (2004)	×	✓	✓
All-pairs execution	Goldwasser (2002) Edwards et al. (2012) Wrenn et al. (2018) Buffardi et al. (2019)	✓	×	×
Mutation analysis	Aaltonen et al. (2010) Shams and Edwards (2013)	✓	✓	×[a]

[a]Addressed in this paper.

Web-CAT (Edwards, 2004), Athene (Pettit et al., 2015), and others (Jackson and Usher, 1997; Spacco and Pugh, 2006; Wang et al., 2011; Papancea et al., 2013). AATs are centralized to ensure a trusted computing base, making them a reliable source of student scores. At institutions that offer incremental feedback, each student may make many dozens of submissions per assignment to the institution's AAT, with bursty traffic near due dates (Edwards et al., 2009). Lower computational cost for the feedback reduces the risk that these AATs will be overloaded.

2.2. Existing measures of student test quality

Educators have explored three principal measures of student test quality: *code coverage*, *all-pairs comparisons*, and *mutation analysis*. None of these measures currently meets our goals for a student test assessment metric (Table 1).

Code coverage is *fast to compute* and *supports incremental feedback*. However, it is not a strong test adequacy criterion. It is satisfied simply when tests "cover" the code, whether or not they confirm that the code works correctly (Aaltonen et al., 2010; Edwards and Shams, 2014; Inozemtseva and Holmes, 2014). That is, even strong code coverage measures (like condition coverage or MC/DC) are insensitive to the *assertions* that appear in software tests. These measures are an effective tool for professional engineers (Ivanković et al., 2019), but like any tool they can be used incorrectly. For example, students often use "pathological" tests to achieve high coverage while only making assertions about (i.e., properly testing) small aspects of the desired functionality (Shams, 2015). Used this way, coverage measures do not give a student actionable feedback.

The **all-pairs approach**—in which a student's tests are run against every other student's code (Goldwasser, 2002)—is a *reliable test adequacy criterion* (Edwards and Shams, 2014), but it can be *slow to compute* (Shams, 2015) and is *not amenable to incremental feedback*. All-pairs testing requires several completed, compatible versions of each piece of a project. Compatibility is ensured when projects are scaffolded (e.g., lower-level courses), but in most upper-level courses students are given only the system-level I/O requirements and take responsibility for designing their own components and internal APIs. Since students' tests and solutions are typically not completed until the deadline (Kazerouni et al., 2017, 2019), there would be little opportunity for feedback during the development process.

Mutation analysis (DeMillo et al., 1978) is a provably *strong test adequacy criterion* (Wong and Mathur, 1995; Offutt and Voas, 1996), and like code coverage it *supports incremental feedback* with *relatively low human cost*. However, comprehensive mutation analysis is prohibitively expensive computationally. But while the shortcomings of code coverage and the all-pairs approach appear to be fundamental, the cost of mutation analysis may be reduced. Next, we discuss research to this end.

2.3. Reducing the cost of mutation analysis

The idea of mutation analysis is to inject micro-faults into the target program, and then determine whether the test suite can identify the change in behavior (DeMillo et al., 1978). Faults are injected using *mutation operators* to create (presumably) incorrect variants called *mutants*. Mutation frameworks use mutation operators to target different aspects of the program when it is represented as an AST. For example, frameworks provide mutation operators for arithmetic expressions, return values, and condition predicates, among others (King and Offutt, 1991). The resulting mutants have been found to be valid substitutes for "real" faults (Andrews et al., 2005; Just et al., 2014). After creating mutants, a test suite's adequacy can be measured in terms of its *mutation coverage* by calculating the proportion of mutants that it detects.

Mutation analysis is costly. Applying a full set of mutation operators to a non-trivial program can yield thousands or millions of mutants, and running the test suite for each mutant is computationally intensive. This process can also exact considerable human cost, since a software tester must design tests that are able to detect all or most of the mutants that are produced. Exacerbating this situation is the possibility of producing an *equivalent mutant*, i.e., a mutant that is functionally identical to the original program and thus will not affect the test suite's results. Equivalent mutants represent wasted work; these "false positives" do not help assess or improve a test suite, and must in general be filtered out manually (Budd and Angluin, 1982).

Considerable effort has been devoted to reducing the cost of mutation analysis. Jia and Harman (2011) categorized these efforts into: (1) *"Do fewer"*, reducing the number of generated mutants, and (2) *"Do faster"*, reducing the execution cost in other ways (e.g., avoiding I/O). We are concerned with mutant reduction techniques, specifically the technique of *selective mutation* (Mathur, 1991), which reduces the number of mutation operators to trade completeness for lower costs.

Among selective mutation approaches, two are prominent: the *sufficient set* (Offutt et al., 1996) and the *deletion set* (Untch, 2009). The sufficient set was introduced by Offutt et al. (1996), who showed that a subset of the Mothra mutation operators (King and Offutt, 1991) would yield results comparable to comprehensive mutation, bringing with it considerable cost savings. While statement deletion was available in the first mutation systems (King and Offutt, 1991), using it as a sole mode of mutation was first proposed by Untch (2009). Delamaro et al. (2014) expanded this idea to include operators that delete different aspects of the target program (e.g., statements, conditions, and variables). Subsequent evaluations of the deletion set have been promising, showing that the deletion set yields substantially fewer mutants than the sufficient set with a minor loss in the accuracy of its test adequacy compared to comprehensive mutation (Deng et al., 2013; Delamaro et al., 2014). Mutation by deletion is a promising path toward practical and scalable mutation testing. However, neither

the sufficient set nor the deletion set have been evaluated in an educational context for large classes with complex projects, and we present data in Section 6.1 to show that neither is fast enough for our purposes.

2.4. Mutation analysis in education

In spite of its virtues, mutation analysis has seen little use or evaluation in the context of student assessment and feedback. Its high computational cost is a critical limiting factor for automated assessment, since it could delay feedback and thus degrade learning outcomes (Azevedo and Bernard, 1995; der Kleij et al., 2015). This cost hinders research into the potential pedagogical benefits of mutation analysis (e.g., its utility as a form of *feedback*, not just *assessment*). This paper enables such work by allowing for fast and accurate incremental feedback using mutation analysis.

There have been few efforts to apply mutation analysis in an educational setting. Aaltonen et al. (2010) reported that mutation analysis revealed deficiencies in students' testing that were not revealed by code coverage. However, their work only considered using mutation analysis non-incrementally (for grading purposes), and did not consider its deployment costs. These costs are of significant concern if an AAT provides students with feedback as they work. Shams and Edwards (2013) explored the use of mutation analysis in novice programming courses, and also compared the cost-effectiveness of various selective mutation approaches with other measures of test quality (Edwards and Shams, 2014). They found that statement deletion was the most cost-effective mutation-based test adequacy criterion, which partially influenced our experimental design. However, the cost of mutation by deletion has not been evaluated for the common educational context of automated assessment systems, and its accuracy as a test adequacy criterion has only been evaluated on software produced by novice CS students.

We build on these efforts in two ways. First, we examine the cost-effectiveness of various mutation approaches in the "production" educational context of an AAT, with its accompanying performance constraints. Second, we apply our analysis to a large corpus of student projects with a substantially wider range of size and complexity than has previously been considered. In this context, we find that existing techniques are inadequate. However, through statistical selection we can reduce the cost of mutation analysis by 50% compared to the deletion subset (and by 90% compared to comprehensive mutation), with only a minor degradation of the test adequacy criterion.

2.5. Mutation analysis tools for Java

Our institution uses Java as the primary programming language, so we sketch the landscape of Java mutation testing tools. Prominent among these are: μJava (Ma et al., 2005), Javalanche (Schuler and Zeller, 2009), MAJOR (Just et al., 2011), and PIT (Coles et al., 2016). Abstractly, they all follow the same two-stage process: (1) Generate mutants, and (2) Run test suites to see if the mutants are detected. Some of these tools target a particular language version (e.g., if they mutate source-level constructs).

Due to language version constraints, μJava and Javalanche are unsuitable in our context. μJava and Javalanche target Java 6, which is no longer support by either OpenJDK or Oracle. Our students use Java 8 or later in their projects. We attempted to use μJava on these projects, but it exhibited many errors. Javalanche has also exhibited issues on similar code-bases (Delahaye and du Bousquet, 2013; Gopinath et al., 2017). Upgrading μJava and Javalanche was out of scope for our work. Other, less prominent tools for Java mutation testing have also been considered by researchers (Delahaye and du Bousquet, 2013; Gopinath et al., 2017), and found to be unsuitable for larger or newer projects for various reasons. We thus focus our discussion on the newer mutation testing tools, PIT and MAJOR.

We compare PIT and MAJOR along two axes: *effectiveness* and *speed*. Effectiveness is determined by a tool's fault-revelation capability, dictated by which mutants it generates. Studies have found PIT more effective than MAJOR (Gopinath et al., 2017; Kintis et al., 2018).

Speed is determined by algorithmic and implementation decisions. There has been no empirical speed comparison between PIT and MAJOR. However, based on our study of their designs, we do not expect a substantial speed difference. To generate mutants, both tools manipulate the program representation in-memory within a single JVM instance. To detect mutants, both tools filter tests using line coverage and prioritize them using testing execution time. As a result, each tool detects a mutant using the fastest covering unit test.

3. Goals and constraints

Here we contextualize our study in modern AATs, and define what constitutes fast-enough test suite feedback in such an AAT. AATs tend to handle substantial throughput, particularly when they provide students with intermediate feedback on incremental submissions. It is imperative that any mutation analysis strategy used in an AAT supports a reasonable response time, both so that students can make appropriate use of intermediate feedback, and so as not to degrade the AAT during times of heavy load.

In order to provide real-world context for this study, we describe our AAT's hardware configuration, throughput, and the processing for a typical submission. At our institution, we use the AAT Web-CAT (Edwards, 2004) served using a machine running CentOS 7 with two 16-core 2.60 GHz Intel Xeon Gold 6142 CPUs and 256 GB RAM. The persistent storage was two 600 GB, 10,000 RPM Hitachi HDDs. Our experiments are memory and compute bound. Because PIT operates only on byte-code in memory, disk accesses only take place to access the compiled bytecode before analysis and to write out results after analysis. All analyses in this paper were run on a separate machine with identical specifications. Day-to-day usage of the AAT did not affect our experiments.

Our institution's AAT server (serving the AAT Web-CAT Edwards (2004)) is shared by many courses at our institution, and is also used by several other institutions. Our AAT has a steady-state load of hundreds of daily student users, peaking around 2000 daily users. These users drive a steady state throughput of 11 submissions per minute, or about 1 submission every 5 s. During submission bursts near assignment deadlines, our AAT receives 39 submissions per minute, or a submission every 1.5 s. The AAT requires a median of 13 s to generate full submission feedback for a submission, including compilation, static analysis, instructor-written reference tests, and test adequacy assessment—currently bytecode-level statement and condition coverage.

In light of this load, our AAT handles one submission per core, leaving some cores idle for the user interface and database functionality for interactive services. Though mutation analysis is an easily parallelizable problem, parallel processing for a single submission is unattractive: using up cores to process a single submission would only transfer the slow-down from time spent processing to time spent waiting to be processed.

While it is desirable to minimize any increase in processing time, using a more reliable test adequacy criterion makes some increase inevitable. In contrast to code coverage, which requires executing student-written software tests only once, mutation-based feedback involves running software tests once *per mutant*, substantially increasing the processing time for each submission.

Our experience with AAT usage and our understanding of student interactions with feedback suggest that increasing the delay in feedback response by minutes would be unacceptable to users. Such a delay would also substantially reduce throughput, to a degree prohibitive to address through additional hardware investments. However, we believe that adding a significantly smaller amount of time—perhaps 30 seconds or less—would approach the realm of feasibility. Additional measures (increased parallelism, faster hardware, more sophisticated cloud deployment, etc.) may still be needed to reach desirable peak throughput. In short, *we want mutation-based test adequacy feedback to add fewer than 30 seconds to the feedback generation time for a single student submission*, where smaller costs are definitely more desirable.

This goal of "fewer than 30 seconds" is specific to our institutional context—our hardware, software, and students. However, institutional needs vary. We explore a *continuum* of selective mutation approaches that could be tailored to institutional needs, budgets, and other factors. For example, an institution with slower hardware might opt for computationally cheaper but less reliable mutation approaches. And for institutions with the budget for faster hardware or managed cloud clusters, more reliable and costly approaches would be within reach. Our approach can help institutions make an informed choice appropriate to their context.

4. Research questions

We now describe the three empirical studies that we conducted. They were designed to provide a mechanism to evaluate student-written test suites using mutation analysis that is feasible for use in an AAT while remaining a reliable test adequacy criterion. For each study, we identify the research questions that it was meant to address.

4.1. Motivational study: Evaluating existing selective mutation approaches for use in an AAT

RQ1: How efficient is comprehensive mutation analysis at providing automated feedback on test suites? We study whether it is actually necessary to improve the efficiency of mutation analysis for student code. It may be that the smaller size of these projects allows mutation analysis to offer test suite assessment within our running time goal. We evaluate the efficiency of using mutation analysis for automated feedback in terms of the time taken for individual submissions to generate mutation analysis results. We interpret results in terms of running time on the AAT server at our institution.

RQ2: Are existing selective mutation approaches cost-effective alternatives to comprehensive mutation? We evaluate the *sufficient* (Offutt et al., 1996) and *deletion* (Delamaro et al., 2014) sets of operators for their feasibility in providing automated incremental feedback. Shams (2015) evaluated these subsets of operators on projects produced by novice programmers, and found statement deletion to be a cost-effective approach. We believe this result is promising, so we conduct an evaluation of the sufficient and deletion operators set on a more general corpus of student codebases, with submissions spanning a wider range of size and complexity.

4.2. Core study: Proposing new mutation approaches that are viable for use in an AAT

RQ3: Can the cost of mutation by deletion be reduced further while maintaining effectiveness? Even though mutation by deletion represents notable runtime savings over comprehensive and sufficient mutation, it may be possible to reduce this cost further without sacrificing too much in assessing test suite adequacy.

RQ4: How do the benefits of different mutation strategies vary by project size? Our analyses were conducted on a diverse set of programs, based on size and complexity (and therefore in terms of the mutants produced). We investigate whether our chosen selective mutation strategies vary in terms of cost-effectiveness based on the size of the projects under test. This would allow educators to make a more informed choice of operator subset to use for test suite evaluation.

4.3. Validation study: Evaluating proposed mutation approaches using an unrelated dataset

RQ5: How do our proposed mutation strategies perform in terms of cost-effectiveness against a separate validation dataset? Although our analyses were conducted on a large corpus of submissions to several assignments, it is possible that our results do not generalize beyond our specific educational context. To address this, we conducted an additional validation study to evaluate the cost-effectiveness of our proposed mutation operator subsets running on a separate dataset published by Kintis et al. (2016), including real-world projects and projects from the mutation testing literature.

5. Study context

5.1. Project corpuses under test

In our Motivational and Core studies, we examined Java projects developed by students enrolled in second-year (CS2) and third-year (CS3) Data Structures courses at a large public university in the US. These students have taken either 1 or 2 (depending on the corpus) prerequisite Java courses, each one of which has included JUnit testing in programming assignments. Therefore, students have the *declarative knowledge* needed to write JUnit tests (i.e., familiarity with the framework), but they may not have the *procedural knowledge* required to write strong test suites. Students were required to write unit tests for their projects, and part of their grade depended on the quality of their test suites (as measured by code coverage criteria).

We analyze a submission corpus that contains 1389 final submissions to seven programming projects. Descriptions of the assignments and the corpus are presented in Table 2, and code sizes in source lines of code are in Fig. 1. The **CS2 sub-corpus** (1019 submissions) consists of submissions to four programming assignments requiring students to implement and test a simple data structure, e.g., a stack or a queue. Students were given two to three weeks to work on each assignment. The **CS3 sub-corpus** (370 submissions) consists of submissions to three more-complex programming assignments. Students were given four weeks to work on each assignment.

This corpus is noteworthy for its scale and for the range of complexity within. This corpus contains around 2–3x the number of projects examined in other studies on mutation analysis for education, along a substantially wider range of size and complexity. Previous studies—e.g., Aaltonen et al. (2010), Shams and Edwards (2013), Edwards and Shams (2014)—have focused on smaller, simpler projects from introductory programming courses, comparable only to the first 2 projects out of our corpus of 7 (i.e., to the first 671 submissions out of our corpus of 1389 submissions).

In our Validation study we analyzed a separate dataset of 12 methods from 6 projects, published by Kintis et al. (2016). In addition to the codebases being tested, the dataset includes mutants and mutation-adequate test suites. More details may be found in Section 8.

Table 2
Programming tasks undertaken by students in our sample, and descriptions of their implementations. *# Mutants* indicates the number of mutants generated under the FULL set. Projects 1–4 are CS 2 projects, and 5–7 are CS 3 projects.

#	Description	n	LoC		Cyc. Comp.		# Classes		# Mutants	
	(data structures implemented)		μ	σ	μ	σ	μ	σ	μ	σ
1	Bag	350	139.20	13.84	26.87	1.61	2.00	0.00	470.16	32.60
2	Linked stack	321	204.29	22.85	38.50	2.94	3.97	0.17	421.44	38.05
3	Array-based queue	259	448.00	39.71	104.00	7.58	6.00	0.10	1651.54	126.83
4	Linked list	89	718.02	221.53	147.38	53.78	8.52	2.99	2988.94	1491.91
5	Hash table, doubly-linked list, memory pool	128	724.26	142.33	152.38	32.40	7.82	1.97	3377.76	776.02
6	Hash table, sparse matrix	133	946.56	178.69	202.17	38.77	8.06	1.97	3244.17	785.65
7	Bintree, skip-list	109	1263.18	303.22	261.96	73.89	15.84	3.35	6095.79	1952.25
	Total	1389	650,515		136,763		8088		2,521,871	

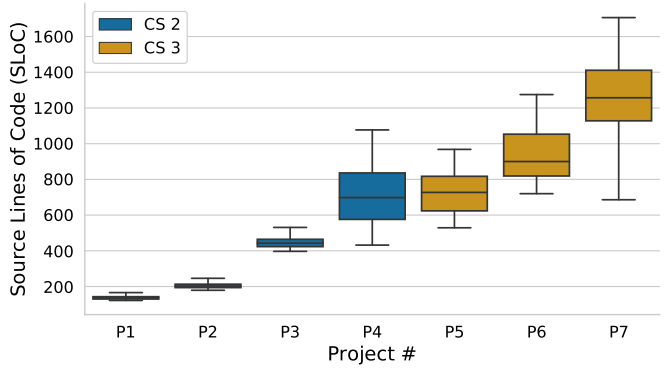

Fig. 1. Our corpus contains submissions to assignments of increasing sizes (source lines of code). Whiskers indicate the 5th and 95th percentiles.

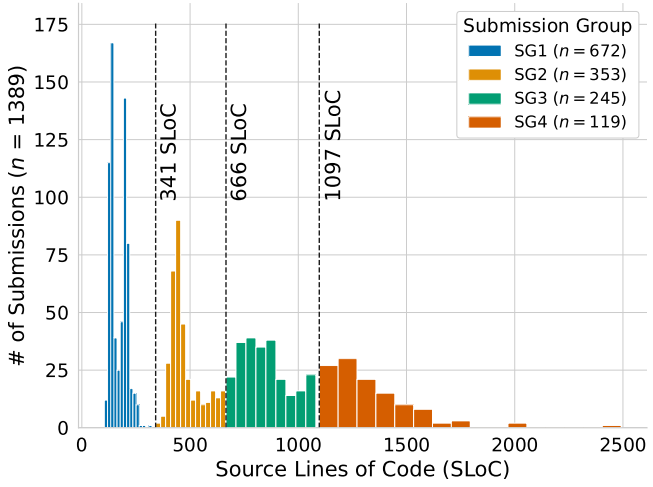

Fig. 2. Groups of submissions based on SLoC. Dashed lines indicate group boundaries.

5.2. Language and tooling

We focus on testing Java programs, since Java is widely used in introductory and advanced programming courses at the secondary and post-secondary levels. Furthermore, the Java ecosystem has good testing frameworks (e.g., JUnit) and many tools for assessing suites using various mutation operators. We used **PIT** (Coles et al., 2016), the state-of-the-art mutation testing system for the JVM, to conduct mutation analysis.

As noted in Section 2.5, multiple studies have found PIT to be the most effective in terms of fault-revelation capability (Gopinath et al., 2017; Kintis et al., 2018). The initial release of PIT was comparable to MAJOR and μJava (Kintis et al., 2016). Kintis then collaborated with Coles— the PIT author —and others to augment PIT with additional mutation operators. Kintis et al. (2018) showed that the updated PIT was more effective at fault revelation than μJava and MAJOR combined. Thus, the current version of PIT (v1.5.2) offers the strongest test adequacy criterion currently available for Java programs. We used this version in our experiments.

Building on the past performance of deletion operators (Section 2.2) in other languages and tools, and on PIT's current dominance of the Java mutation testing space in terms of cost and reliability (Section 2.5), we have analyzed selective mutation using PIT with the goal of further reducing the cost of mutation testing while maintaining performance on par with the comprehensive set of PIT operators.

5.3. Data preparation

We took several steps to prepare the corpus of students' submissions for analysis. Notice in Table 2 that the dataset is biased toward smaller, simpler projects. Nearly half of all submissions belong to Project 2 or 3 in the CS2 corpus. Submissions in the CS3 corpus—which are substantially larger and more complex—would not be well-represented by corpus-wide descriptive statistics, but they are critical to understanding the scalability of our approach.

Therefore, we split the corpus of submissions into groups based on program size. Splitting was performed using (Jenks, 1977) natural breaks optimization—a variation of K-Means clustering (Lloyd, 1982) simplified for 1-dimensional data. The main idea behind this splitting technique is to (1) maximize the variance between groups, and (2) minimize the variance within groups.

We used goodness of variance fit (GVF) to determine the appropriate number of splits. This measure is directly proportional to between-group variance, and inversely proportional to within-group variance (Jenks, 1967). Therefore, we would like to maximize it. To determine the appropriate number of splits k, we applied the Jenks algorithm for increasing values of k from 2 to 7 and plotted the GVF for each splitting. The diminishing improvements in GVF indicated $k = 4$ to be an appropriate number of splits for this dataset. The four submission groups SG1–SG4 and their intervals are depicted in Fig. 2.

The makeup of the submission groups generally follows the averages given in Table 2. We report the "major" occupants of each submission group (i.e., course projects that account for $\geq$ 10% of the submission group). SG1 consisted entirely of submissions to early projects in CS2 (#1 and #2 in Table 2), more or less evenly split. SG2 and SG3 included submissions from both courses. SG2 contained submissions to projects #3 (73%), #4 (10%), and #5 (14%). SG3 contained submissions to projects #4 (20%), #5 (31%), and #6 (42%). Finally, SG4 was entirely from CS3, consisting of submissions to projects #6 (23%) and #7 (72%).

5.4. Measuring the cost of a selective mutation approach

In the following sections, we evaluate the cost of several existing and proposed selective mutation analysis approaches. We use two measures of cost: the *computational cost* and the *running time cost*.

Computational cost was measured as the number of mutants produced per thousand source lines of code (KSLoC). This cost indicates the number of times a project's test suite needs to be run to conduct mutation analysis, giving an idea of the relative cost of a given subset of operators.

We also measured the **running time cost** of existing and proposed selective operator subsets on a server that is similar to a real-world AAT setup (see Section 3 for hardware specifications). This gives us an idea of the amount of time a student might wait between making a submission to the AAT and receiving feedback about their test suite.

We note that—since our corpus contains *final* submissions as opposed to intermediate, incomplete submissions—our cost measurements represent upper bounds on the cost of producing mutation-based incremental feedback. Final submissions are likely to contain more code and therefore to produce more mutants than intermediate submissions. However, we report that intermediate submissions are not far removed from final submissions in terms of size (and therefore in terms of the expected cost of mutation analysis). For example, the median student's median submission in the CS3 course contained 96% of the total LoC that would appear in their final submission. This number indicates that most submissions cost approximately what our measurements on final submissions imply. The large proportion may be explained by students' tendencies to make many submissions in quick succession near deadlines to check if small changes help them to pass all of the instructor-written tests.

6. Motivational study: Evaluating existing approaches

6.1. RQ1. How efficient is comprehensive mutation analysis at providing automated feedback on test suites?

In this section we evaluate the computational and running time cost of applying a comprehensive set of mutations to our corpus of target programs, in terms of the time taken to generate feedback on their test suites. We interpret results in terms of our desired performance goals.

6.1.1. Method

We define the FULL set of PIT operators to be all those used in the comparison of PIT with μJava and MAJOR by Kintis et al. (2018) (see Table 4 in the reference), with some minor optimizations. Specifically, we omitted operators that would, by their definitions, perform the same mutations that would be performed by other operators (*duplicate mutants*). For example, the *ROR* operator, which was added to PIT by Kintis et al. produces a superset of the mutants that the *ConditionalsBoundary* and *NegateConditionals* operators produce. *ROR* replaces occurrences of comparison operators with all other comparison operators. For example, the $<$ operator would be systematically replaced by $<=$, $>$, $>=$, $==$, and $!=$, for a total of 5 mutants. On the other hand, the *ConditionalsBoundary* operator would only replace it with $<=$, and *NegateConditionals* would only replace it with its negation ($>=$). Clearly, the mutants produced by these operators are duplicates of those created by *ROR*. We likewise omitted the *PrimitiveReturns* and *FalseReturns* operators (which are $\subseteq$ *NonVoidMethodCalls*), and the *InvertNegatives* operator ($\subseteq$ *ABS*). Finally, we omitted a subclass of the *AOR* operator (*AOR1*, according to PIT's nomenclature), which would duplicate mutants produced by the *Math* operator.

We measured the *computational cost* and *running time cost* of mutation analysis using the FULL set using the two measures of cost defined in Section 5.4. Cost was measured separately for each group of submissions.

We do not evaluate the FULL set of operators for its *accuracy* at measuring a test suite's adequacy (defect-detection capability). Comprehensive mutation analysis has been empirically shown to be a reliable measurement of test adequacy (Offutt, 1992; Andrews et al., 2005; Just et al., 2014). The FULL set as described here has been shown to be the strongest set of mutation operators available for Java programs (Kintis et al., 2018). Accordingly, a test suite's mutation coverage according to the FULL set of PIT operators is the best available proxy for its adequacy.

Mutation analysis was run on 1389 submissions, and results and running times were collected for each submission. Due to the size of the corpus, we did not attempt to manually exclude equivalent mutants from the corpus (see Section 10). Mutants were treated as *detected* if a test case failed or timed out when running on the mutant. Test timeouts were determined using PIT's default settings—a test was treated as timing out if the execution time exceeded $t * 1.25 + 4000$ ms, where t is the normal execution time of the test case, measured before running mutation analysis.

The percentage of timed-out mutants was not uniform across submission groups. Submissions in SG3 had a higher percentage of mutants that timed out (2.18%), relative to the other groups (0.24% in SG1, 0.91% in SG2, and 1.37% in SG4). The result is that running times for submissions SG3 were higher than one might expect given the number of mutants they produced. We discuss this further in Section 10.

6.1.2. Result

Mutation analysis using the FULL set is not efficient enough for incremental feedback on medium-to-large projects. Results are summarized in Fig. 3, which also summarizes results from **RQ2** (Section 6.2). For smaller projects (those in submission group SG1), analysis took a median of 16 s to run per submission. Unsurprisingly, running time was higher for the larger submissions. Mutation analysis on submissions groups SG2, SG3, and SG4 ran in 84 s, 283 s, and 325 s respectively. Such slow feedback times are less likely to help students to actively identify weaknesses in their test suites as they develop projects. Such running times would also impose an unacceptable additional load on an AAT. Recall that parallelizing mutation analysis for a single submission is precluded since multiple server cores are already in use to process multiple submissions at once (see Section 3).

6.2. RQ2. Are existing selective mutation approaches cost-effective alternatives to comprehensive mutation?

Having found that the FULL set of PIT operators is not efficient enough for incremental feedback in AATs, next we evaluate the cost-effectiveness of two operator subsets from the literature: the SUFFICIENT set and the DELETION set. Shams (2015) found both subsets to be reliable at measuring the adequacy of test suites produced by novices, and found *statement deletion* to be cost-effective. However, the costs of these subsets have not been evaluated for the common educational context of automated assessment systems, and their accuracy as test adequacy criteria has only been evaluated on software produced by novice CS students (i.e., like those in the CS2 corpus). We evaluate and compare the performance of sufficient and deletion operators with that of the FULL set of operators (Section 6.1), analyzing projects from a wider range of sizes, complexities, and opportunities for mutation.

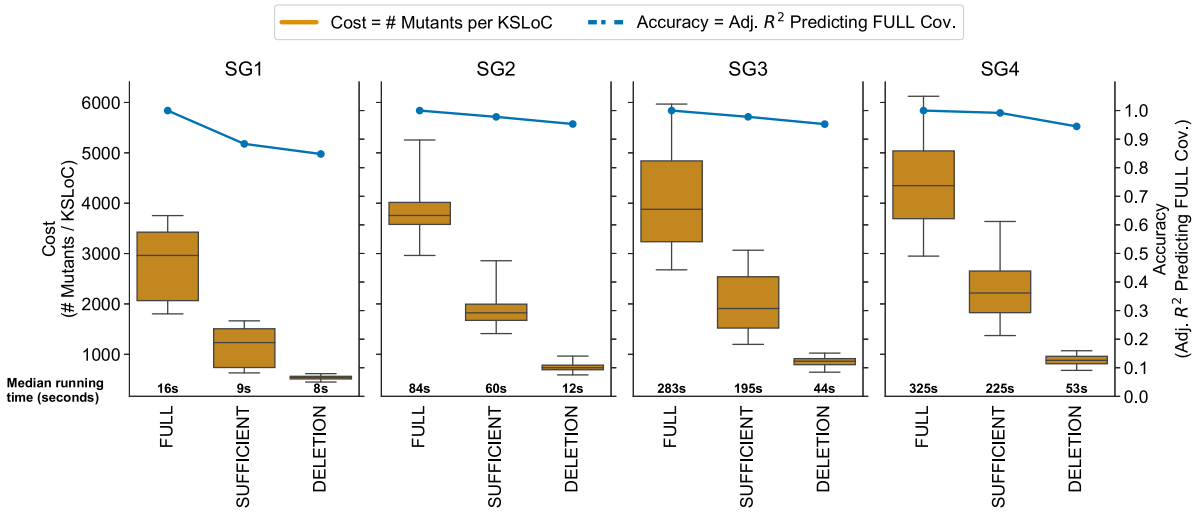

Fig. 3. Cost and accuracy of the FULL, SUFFICIENT, and DELETION subsets of mutation operators, for each of the submission groups. For each subplot, the left axis represents cost (# mutants per KSLoC) and the right axis represents accuracy (adjusted R^2 in a model predicting FULL coverage). The y-axes are shared across subplots. Inline text at the bottom of the charts indicates the median running time on our server.

6.2.1. Method

The SUFFICIENT set of PIT operators, proposed by Offutt et al. (1996), is intended to produce significantly fewer mutants while maintaining effectiveness. Laurent et al. (2017) extended PIT with the SUFFICIENT operators. There is one exception: the Logical Connector Replacement (*LCR*) operator—which creates mutants by replacing logical `AND` and `OR` connectors—does not exist in PIT. The `&&` and `||` logical connectors in Java do not translate to single bytecode instructions that can be mutated. Instead, individual conditions translate to branching instructions, which are mutated by *ROR*. The SUFFICIENT set of mutation operators as implemented in PIT is in Table 3.

We define the DELETION set in PIT to be a subset of operators that approximates the mutation operators proposed by Delamaro et al. (2014), which deleted statements, operators, variables, and constants. Their evaluation showed the subset to be a cost effective selective mutation approach. Since PIT operates on Java bytecode, a precise replication of those deletion operators is not practical. For example, Java bytecode does not explicitly distinguish between local variable initializations and assignments, with the result being that local variable deletion as described by Delamaro et al. is not currently implemented in PIT. Additionally, constants are treated as either literal values or as local variables, depending on the compile-time optimizations that are applied, complicating Delamaro et al.'s constant deletion (CDL) mutation operator. We use six operators as the DELETION set (listed in Table 3).[1]

We evaluate each subset of operators for each group of submissions along two axes: *cost* and *accuracy*. Cost was measured as described in Section 5.4.

To evaluate a subset for *accuracy*, we measured, for each operator in the subset, the proportion of mutants that were detected (i.e., the submission's *mutation coverage* for the given operator). We also measured the submission's FULL mutation coverage. With these data in hand, we used linear regressions of the following form:

- **Independent variables**: For each operator in the subset, the % of mutants detected
- **Dependent variable**: Mutation coverage achieved under the FULL set of operators
- **Subjects**: Submissions in the given group

Accuracy was therefore measured as the proportion of variance in FULL coverage explained by coverage of individual operators in the subset being evaluated, i.e., the regression model's adjusted R^2. Accuracy is measured against the FULL set because it is the best available proxy for a test suite's defect-detection capability (see Section 6.1).

Using this approach, we measured the cost and accuracy for the FULL, SUFFICIENT, and DELETION subsets of operators, for each group of submissions SG1–SG4.

6.2.2. Result

The SUFFICIENT and DELETION sets are comparable in terms of their accuracy, and the DELETION set is much more cost-effective. However, its running time still presents challenges for larger projects. Results are summarized in Fig. 3. For each submission group SG1–SG4, the figure depicts the cost in two ways: *mutants/KSLoC* (boxplots) and *running time* (infixed text). The figure also depicts the *accuracy* (line charts) for each subset. We highlight two aspects of this figure.

First, although the DELETION set is slightly weaker than the FULL and SUFFICIENT sets, it still provides a reliable assessment of a test suite's adequacy for most submission groups. Fig. 3 indicates that its adjusted R^2 is high, ranging from 0.84–0.95 across submission groups.

Second, although the DELETION set shows notable cost savings over the FULL and SUFFICIENT sets, there is still need for improvement. The precipitous drop in cost from the FULL set to the DELETION set is visually apparent in Fig. 3 (see the boxplots). That said, offering automated feedback in an AAT using the DELETION set remains a costly proposition, especially for the more complex projects in submission groups SG3 and SG4. The DELETION set produces relatively fast mutation results for submissions in SG1 and SG2, taking a median 4 and 16 s per submission, respectively (see the infixed text in Fig. 3). For submissions in SG3 and SG4, the DELETION set took far longer: a running time of approximately 1 min per submission. This time is far greater than our target of approximately 30 s.

Briefly put, using the DELETION set is considerably cheaper than using the SUFFICIENT set, which in turn is considerably cheaper than the FULL set. These sizeable differences in cost,

[1] We did not use the *OBBN* mutation operator—variants of which mutate by deleting bitwise operators and operands—because only 5.7% of submissions in our corpus contained any bit operations.

Table 3
Selective mutation approaches evaluated for use in an AAT in this paper, including the incremental subsets evaluated in Section 7.2. The *Ref.* column refers to the first proposal of the specified subset.

Approach	PIT operators	Ref.	Evaluated
FULL	All in Table 4 in Kintis et al. (2018), **omitting** {*ConditionalsBoundary*, *NegateConditionals*, *PrimitiveReturns*, *FalseReturns*, *InvertNegatives*, *AOR1* }	DeMillo et al. (1988)	RQ1
SUFFICIENT	*AOR*, *ROR*, *ABS*, *UOI*	Offutt et al. (1996)	RQ2
DELETION	*RemoveConditionals*, *AOD*, *NonVoidMethodCalls*, *VoidMethodCalls*, *MemberVariable*, *ConstructorCalls*	Delamaro et al. (2014)	RQ2, RQ3
2-op subset of DELETION	*RemoveConditionals*, *AOD*	[a]	RQ4, RQ5
1-op subset of DELETION	*RemoveConditionals*	[a]	RQ4, RQ5

[a]Proposed in this paper.

coupled with relatively small differences in accuracy, suggest that the PIT DELETION set is a promising direction for cost-effective mutation analysis.

This analysis can be seen as a replication study that gathers more support for previous work evaluating deletion operators. We substantiated findings from Untch (2009), Deng et al. (2013), Delamaro et al. (2014), and Derezińska (2016) that reported mutation by deletion to be highly cost-effective. We also lent some generality to claims from Shams (2015), who found that statement deletion (SDL) represented a promising path toward the use of mutation testing for projects produced by novice programmers.

7. Core study: Proposing new approaches

Although the DELETION set is an improvement over other selective subsets, we found that it is still too expensive for larger student projects (Section 6.2). Therefore, we explore the possibility of reducing its cost further while maintaining its effectiveness.

We use the DELETION set as a starting point because it has two desirable properties. **First**, it is cost-effective. It already provides a good approximation of FULL mutation adequacy at a fraction of the cost of FULL mutation as well as that of other prominent operator subsets in the literature (Untch, 2009; Delamaro et al., 2014). Shams and Edwards (2013) found it to be a reliable measure of test adequacy in projects produced by novice programmers, and we have confirmed this property in our Motivational study (Section 6).

Second, and critically, DELETION operators tend to produce a significantly smaller proportion of equivalent mutants than other selective subsets like the sufficient sets from Offutt et al. (1996), Siami Namin et al. (2008), Untch (2009) and Delamaro et al. (2014). It is impossible to automatically discard or to avoid creating these mutants, because determining program equivalence is undecidable (Budd and Angluin, 1982). Therefore, creating a mutation-adequate test suite requires the tester to manually identify and ignore these mutants during testing. At best, this is unproductive, because this activity does not help the tester to strengthen their test suite, and may even reduce their reliance on feedback because the false positivity rate is too high. At worst, the tester (particularly a student) may mis-classify an equivalent mutant as detectable, and futilely try to devise a test case to do so. Reducing the incidence of these mutants would greatly increase the utility of mutation-based feedback to students.

While there do exist non-DELETION operators that produce few equivalent mutants, DELETION operators are more attractive since they tend to produce fewer mutants than other operators. As an example, consider the arithmetic operator replacement (*AOR*) mutator, which mutates arithmetic operations by replacing arithmetic operators in expressions. Empirical measurements from Yao et al. (2014) suggested that *AOR* produces few equivalent mutants. However, for a given expression—e.g., `a + b`—*AOR* would produce four mutants: `a - b`, `a * b`, `a/b` and `a % b`. The arithmetic operator deletion (*AOD*) mutator, on the other hand, would produce only two mutants for the same expression: `a` and `b`. As we have seen in the literature (Delamaro et al., 2014) and in Section 6.2, detecting a DELETION mutant—like those produced by *AOD*—often results in detecting other non-equivalent mutants. So, even though both *AOR* and *AOD* are likely to produce few equivalent mutants, the *AOD* mutator is more attractive since it also produces a smaller total number of mutants. Similar properties are observable for other DELETION operators.

In this section, we investigate whether a subset of the DELETION set performs comparably well at approximating mutation adequacy. First, we evaluate the predictive power added by individual DELETION operators to approximate FULL coverage (Section 7.1). We conduct this step on the entire corpus of 1389 submissions so as to not overfit to individual submission groups. This results in an ordering in which DELETION operators may be chosen (or omitted) to produce a cost-effective approximation of FULL coverage. We then use this ordering to incrementally evaluate subsets of the DELETION set on each submission group SG1–SG4 (Section 7.2).

7.1. RQ3: Can the cost of mutation by deletion be reduced further while maintaining effectiveness?

7.1.1. Method

To determine effectiveness, we formulate a new regression problem similar to the one described in Section 6.2. Instead of running it on individual submission groups, we perform this regression on the entire corpus of submissions.

We used a statistical procedure to select a subset of mutation operators out of an initial superset. We fit linear models in each step using the `statsmodels` Python package (Seabold and Perktold, 2010). The goal is to produce a subset of operators (selective mutation) that incur acceptable losses in effectiveness.

Forward selection (Bozdogan, 1987) is a statistical feature selection method. Starting with an empty model (i.e., with no features), we consider features one at a time, measuring how much each one improves the model. The best-performing feature is added and the procedure is repeated for all remaining features. This process repeats until the model stops improving, or until there are no more features.

Table 4
Forward selection on the entire corpus of submissions, choosing DELETION operators. White cells contain *cumulative* values for intermediate models, after adding each operator. For example, the *AOD* operator adds a median 140 − 102 = 38 mutants per submission. Gray cells contain values from the final model. Though feature selection was done on the basis of BIC, we report adjusted R^2 for the sake of interpretability.

	Operator added	# Mutants generated			Adj. R^2	Coeff.	Std. error
		Median	% of DELETION	% of FULL			
	(intercept)	–	–	–	–	0.03	0.008
1	*RemoveConditionals*	102	36.04%	7.04%	0.78	0.35	0.011
2	*AOD*	140	49.47%	9.67%	0.88	0.19	0.007
3	*NonVoidMethodCalls*	236	83.39%	16.30%	0.91	0.28	0.012
4	*VoidMethodCalls*	240	84.81%	16.57%	0.92	−0.04	0.005
5	*MemberVariable*	271	95.76%	18.72%	0.92	0.06	0.009
6	*ConstructorCalls*	283	100.00%	19.54%	0.92	0.04	0.007

Forward selection is generally used when one wishes to select a small subset from an initial pool of features. Our features are individual mutation operators. However, each feature carries with it some computational cost. Therefore, forward selection is an appropriate feature selection strategy since it will (theoretically) help reduce the number of operators while maintaining overall effectiveness.

We start with no operators, and at each step we add the operator that minimizes the Bayesian Information Criterion (BIC) (Schwarz, 1978). If two operators perform equally well when added to the model, we select the one with lower cost, i.e., the one that produces fewer mutants. BIC was chosen over R^2 since it is better at predicting model performance on future, unseen data. It was chosen over the closely related Akaike Information Criterion (AIC) because BIC penalizes additional features more heavily than AIC and might result in a simpler model (Bozdogan, 1987). This benefits our aim of reducing the number of mutation operators. The procedure stops when none of the remaining operators reduce BIC any further.

We used the procedure described above to incrementally choose operators in order of decreasing value added. Since our goal is to minimize cost, we chose operators from the cheapest known-good subset of mutation operators, the DELETION set. At each step, we add the next best operator that further improves the model according to BIC. This procedure therefore yields a sequence of DELETION operators ordered by the additional value they bring to the model. Further adjustments may be made to this sequence based on cost considerations, e.g., by omitting operators that add little value on top of previously chosen operators.

Note that although forward selection is a greedy approach, in this case it produced an optimal ordering of operators. That is, at no point was any single DELETION operator "incorrectly" chosen over two or more other operators that were cheaper and performed better. This was confirmed with a brute-force examination of the $2^6 - 1 = 63$ possible combinations of DELETION operators. As mentioned earlier, selecting mutation operators in this way gives a sequence of operators ordered by the additional value they bring to a test adequacy measurement. In **RQ4**, we use this ordering to build incremental cost-effective subsets of DELETION operators for each submission group (see Section 7.2).

Siami Namin et al. (2008) also used a feature selection procedure to select mutation operators in their analysis of C programs using Proteum. Instead of forward selection, they used least angle regression (LARS), which is appropriate for high-dimensional data, e.g., when the number of available features is much larger than the number of data points. Indeed, Namin et al. chose from 108 candidate Proteum operators using a dataset of 7 representative C programs. In contrast, in this paper we reduce from 6 candidate PIT operators (the DELETION set) using a dataset of 1389 programs, an experimental setup that is well-suited to forward selection.

7.1.2. Result

A small subset of DELETION operators is responsible for most of the DELETION set's value, indicating that its cost can be reduced further. Applying this process to the entire corpus of 1389 submissions yielded DELETION operators in the order described in Table 4. Highlighted cells indicate cost, accuracy (adjusted R^2), and errors from the final model, and other cells indicate cost and accuracy from intermediate models considered during forward selection. Notice that all the DELETION operators were included in the final model, suggesting that each of them brings some additional explanatory power to the model. In other words, in our experiment none of the DELETION operators is completely subsumed by a combination of the others.

The DELETION operators were able to explain 92% of the variance in mutation coverage achieved under the FULL set (see the highlighted R^2 value in Table 4), while doing just under 20% of the work. This is in keeping with previous findings that mutation by deletion is highly effective, and lends further support to our findings in Section 6.2.

Critically, *a small subset of DELETION operators is responsible for most of its effectiveness*. Model improvement tended to plateau after the first three operators were selected. The *RemoveConditionals* and *AOD* operators alone performed reasonably well at predicting coverage under the FULL set (adjusted $R^2 = 0.88$). *NonVoidMethodCalls* was selected next, bringing with it a slight increase in effectiveness: R^2 goes from 0.88 to 0.91. The addition of subsequent operators resulted in moderate successive increases in cost, and the model never improved beyond adjusted $R^2 = 0.92$ (rounded). These diminishing returns suggest that, after a certain point, additional DELETION operators are not worth the cost they incur.

7.2. RQ4: How do the benefits of different mutation strategies vary by project size?

Recall that the submission dataset is heterogeneous in size and complexity (Section 5). The models presented in Table 4 are based on the entire corpus of 1389 submissions. However, it is possible and plausible that different operator subsets perform better for submissions belonging to different groups, due to differences in the available opportunities for mutation. For example, submissions in SG1 overwhelmingly belong to early assignments in the CS 2 course. They are small and simple codebases that present comparatively fewer mutation opportunities, even when normalizing by program size.

Submissions in groups SG2–SG4 were larger not only in terms of KSLoC, but also in terms of the expected mutation opportunities available per line of code. For example, SG1 submissions contained an average 20.52 ($\sigma = 18.03$) math operations per KSLoC, while submissions in SG2, SG3, and SG4 contained between 69.20 ($\sigma = 27.12$) and 84.94 ($\sigma = 35.02$) math operations

per KSLoC. That is, submissions in SG2–SG4 provided many more opportunities for the *AOD* mutation operator to act *on each line of code* than did submissions in SG1. Similar per-LoC trends were observed for other program constructs like the number of method invocations, variables used, and parenthesized expressions.

It is therefore no surprise that submissions in SG1 produced significantly fewer mutants per KSLoC than projects in SG2–SG4. Submissions in SG1 produced an average of 2772 ($\sigma = 719$) mutants per KSLoC, while submissions in SG2, SG3, and SG4 produced an average of 3896 ($\sigma = 738$), 4076 ($\sigma = 1104$), and 4443 ($\sigma = 1034$) mutants per KSLoC, respectively. An analysis of variance followed by post-hoc analysis using Tukey's HSD test showed that the pairwise differences in mutants per KSLoC between groups SG2–SG4 is at least an order of magnitude less than the difference between SG1 and each of the other submission groups ($p < 0.05$ for all pairs).

As an example of how this might affect results, consider that the cyclomatic complexities for submissions to Projects 1 and 2 are substantially lower than those for Projects 3–7 (see Table 2). This translates into relatively fewer mutation opportunities for the *RemoveConditionals* operator in the smaller and simpler projects. It would be reasonable to expect this operator to perform worse on these projects than on Projects 3–7. Conversely, in the larger projects, it may mean more "impactful" mutants, i.e., *RemoveConditionals* mutants whose detecting tests would also detect mutants from other operators.

We conjecture that the choice of DELETION operators may differ based on the actual programs under test. Therefore, we investigated the differing cost-effectiveness of operator subsets based on the programs under test.

7.2.1. Method

We incrementally built "*n*-operator" subsets of DELETION operators for increasing values of *n*. Operators were selected one at a time in the order obtained through forward selection (Table 4), and each resulting subset was evaluated separately against each submission group using a linear regression of the form described in Section 6.2.

7.2.2. Result

Mutation adequacy on larger projects can be approximated with fewer mutation operators. Composite results (including subset accuracy, computational cost, and running time cost) are summarized in Fig. 4, which is a "zoomed in" version of Fig. 3. The 1-op, 2-op, and 3-op subsets have been included, and the FULL and SUFFICIENT subsets have been removed. We include the DELETION set to serve as a baseline for cost comparisons. Accuracy measures are made with respect to the FULL set (i.e., the strongest known test adequacy criterion for Java programs). Fig. 4 includes the incremental subsets (the 1-op and 2-op subsets) proposed in Table 3.

- **1-op Subset.** The first subset comprises only the *RemoveConditionals* operator, which removes conditionals by replacing them with boolean literals (`true` or `false`).
  The 1-op Subset shows poor performance for SG1, the group of small submissions, explaining a meager 47% of variance in FULL coverage (see the right most point in the first subplot in Fig. 4). For the groups in the middle, SG2 and SG3, *RemoveConditionals* is able to explain 88% and 86% of the variance in FULL coverage, respectively. It is able to explain 90% of the variance in FULL coverage for group SG4 (the group containing the largest submissions).
- **2-op Subset.** This subset contains the 1-op Subset plus the *AOD* operator, which eliminates arithmetic operators from statements by removing operands.
  This subset does better at predicting FULL coverage for all submission groups, with a natural increase in cost. The two operators—*RemoveConditionals* and *AOD*—are able to explain over 92% of the variance in FULL coverage for SG2–SG4. The subset still performs relatively poorly for SG1, with adjusted $R^2 = 0.80$.
- **3-op Subset.** This subset contains the 2-op Subset plus the *NonVoidMethodCalls* operator, which removes calls to non-void methods by replacing their return values with the given type's default value.
  The inclusion of *NonVoidMethodCalls* results in negligible improvements in model performance for all submission groups. The model continues to perform well for groups SG2–SG4 (adjusted $R^2 > 0.94$), and it continues to perform poorly for group SG1 (adjusted $R^2 = 0.84$). Note that adding the *NonVoidMethodCalls* operator *nearly doubles* the costs incurred by the previous subset for each submission group.
- **6-op Subset.** For the sake of brevity, we jump to results for the entire available set of DELETION operators, i.e., containing all 6 deletion operators listed in Table 3. With the entire DELETION set included, models are able to explain a high amount of variance in FULL coverage (94% or higher) for submission groups SG2–SG4.
  For group SG1, the model is only able to explain 85% of the variance in FULL coverage. For all groups, this represents a small improvement from the 3-op subset.

In addition to examining the number of mutants produced by each incremental subset, we also estimated their running times on our system, using the scheme described in Section 5.4. Normalizing by program size, we obtain a composite measure of running time cost for a given subset of operators, in *seconds per KSLoC*, facilitating comparisons across submission groups. The FULL and DELETION sets took an estimated median of 206 and 35 s per KSLoC, respectively. The 3-op subset offered little improvement over the DELETION set (median = 36 s per KSLoC). Improvements were more pronounced for the 2-op and 1-op subsets, with respective estimated running times of 24 and 18 s per KSLoC. It is evident that the estimated running time impact of the larger subsets of operators (including the DELETION and even the FULL set) are feasible for smaller projects (SG1), but not for larger projects (SG2–SG4).

8. Validation study

Our goal in Sections 6 and 7 was to develop a scalable approach to provide students with rapid mutation-based feedback on the quality of their test suites. Our findings suggested that only one or two DELETION operators can approximate the mutation coverage that would be achieved under the FULL set of operators. However, due to their scale and context, the studies suffer from threats to internal and external validity. We conducted an additional study to validate our findings by addressing the two most critical threats: (1) results that may be over-fitted to our educational context, and (2) the presence of equivalent mutants in our analysis. Discussion of the remaining and (we believe) less critical threats is deferred to Section 10.

We evaluated the cost-effectiveness of our chosen mutation operators using a dataset published by Kintis et al. (2016),[2] used in their comparison of various mutation testing tools for Java. The dataset comprises codebases, mutation-adequate test suites, and manually marked equivalent mutants for 12 methods in 6 Java projects. According to Kintis, 10 methods were randomly chosen

[2] http://pages.cs.aueb.gr/~kintism/papers/scam2016/. Accessed: April 25, 2020.

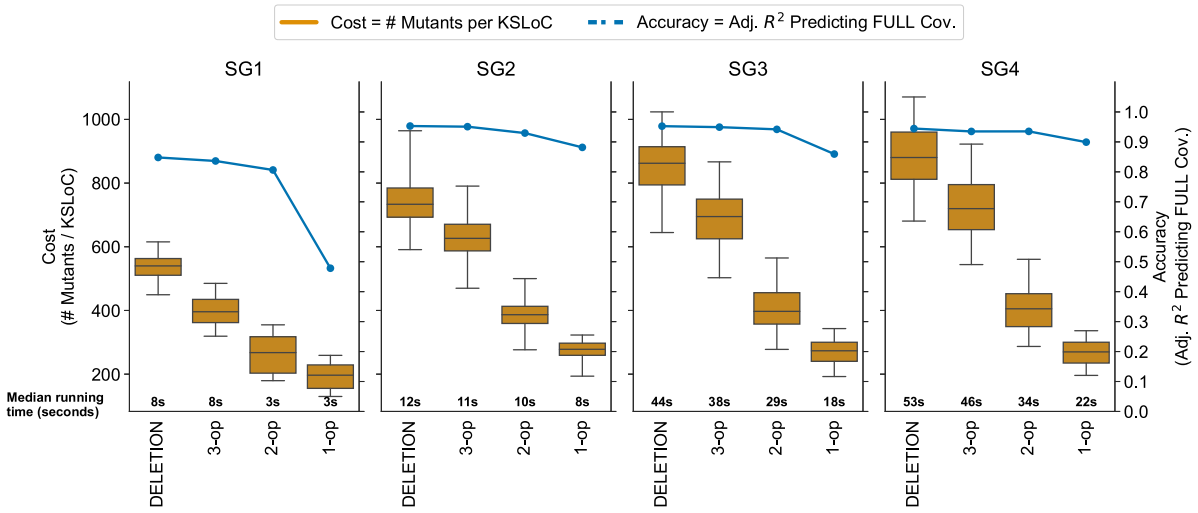

Fig. 4. The cost and accuracy of our proposed incremental subsets of operators. For each subplot, the left axis represents cost (# mutants per KSLoC) and the right axis represents accuracy (Pearson's r w.r.t. FULL coverage). Y-axes are shared across subplots. Inline text at the bottom of the charts indicates the median running time on our server.

from 4 real-world projects (Commons-Math, Commons, Pamvotis, and XStream). The projects ranged from 5505 LoC to 17,294 LoC, and the chosen methods ranged from 18 to 55 LoC. They chose two more methods (Bisect and Triangle), which are 23 and 39 LoC long, from an oft-cited software testing textbook (Ammann and Offutt, 2008). The methods and mutants they spawned are listed in Table 5.

Analyzing this dataset helped us to address two threats to the validity of our previous studies.

External validity. **Threat:** Though our corpus comprised 1389 programs of varying sizes and complexities, our findings may not generalize beyond the educational context, or even beyond our particular educational context. **Mitigation:** The validation dataset contains several real-world libraries and frameworks that are built for a range of purposes (i.e., String manipulation, network management, mathematics and statistics, and XML parsing) and used by thousands of users (according to Maven and SourceForge; see Table 5). If our Core results hold under analysis of this dataset, there is a better case for generalizability.

Internal validity. **Threat:** We did not exclude equivalent mutants from our Motivational and Core studies. The problem of automatically identifying these mutants is undecidable (Budd and Angluin, 1982), but can be done by manually inspecting programs. This was infeasible in our study due to the size of the corpus ($n_{projects} = 1389$, $n_{mutants} \approx 2.5$M), and would be impossible to operationalize in an automated assessment context. **Mitigation:** In the validation dataset, Kintis et al. have manually marked equivalent mutants, allowing us to exclude them from the analysis. If our chosen operator subsets from the Core study prove to be cost-effective using the validation dataset, our findings may be said to be free of this threat. Throughout this section, we measure the costs of operator subsets as their relative cost savings over the FULL set.

8.1. RQ5: How do our proposed mutation strategies perform in terms of cost-effectiveness against a separate validation dataset?

Having proposed and evaluated the cost and reliability for the DELETION set and incremental subsets on our submissions corpus, we seek to validate our findings along the same axes on a second dataset. In this section, we measure the DELETION, 3-op, 2-op, and 1-op operator subsets in terms of their *computational cost* and *reliability*. We interpret results in terms of the trade-off between cost savings and effectiveness as compared to the FULL set.

Here is the context for the validation study. Mutation analysis was run on the codebases studied by Kintis et al. using the FULL set of PIT operators described in Study 1 (see Section 6.1). This produced a dataset of 3037 mutants generated by 17 mutation operators. Of these, 355 (11.69%) were equivalent mutants. Note that the total number of mutants produced is higher than the number of PIT mutants reported by Kintis et al.: this is because PIT's available mutation operators have been extended since 2016, also by Kintis et al. (2018). We computed full mutation matrices for each project—for each detectable mutant, we obtained all of its detecting tests. In other words, for each test, we obtained the list of mutants it detected.

Some additional work was needed to prepare the dataset for analysis. As mentioned in Section 5.2, we used an improved version of PIT, whose augmented set of mutation operators offers the strongest measurement of test adequacy currently available for Java programs (Kintis et al., 2018). As a result, this version of PIT produced more mutants than those published by Kintis et al. (2016), which resulted in there being previously unmarked equivalent mutants. These cases were few because Kintis et al. provided equivalent mutants from μJava and MAJOR in addition to (an older version of) PIT. So the only mutants that needed further checking were those that (1) were not produced by any of the three tools compared by Kintis et al. *and* (2) were not detected by the provided mutation-adequate test suites. Additionally, it was necessary to add two assertions to the Pamvotis test suite to detect two undetected mutants. These mutants affected only the private global state in the class under test (i.e., the value of an instance variable), but not the method's outcome (either with a single or repeated calls). However, they could affect calls to other methods, and therefore cannot be considered equivalent. Detecting these mutants involved: (1) modifying a private field to be publicly accessible, and (2) adding one assertion each to two existing test cases to check the value of the field. The addition of these assertions did not affect the impact of the modified tests other than to detect the targeted mutants. These were the only code changes made to the dataset.

Table 5 describes the subjects and the mutants they spawned under different operator subsets. Missing values indicate that the subject did not produce any new DELETION mutants in the current incremental subset. For example, the `Triangle#classify`

Table 5
Projects tested in the validation study, and the number of mutants and equivalent mutants produced by the FULL, DELETION, and incremental subsets of mutation operators. Projects are sorted by *Usage*: for the real-world projects, reports the number of Maven artifacts that depend on them (*since Pamvotis is not on Maven, we report the # downloads on SourceForge) as of May 6, 2020.

Usage	Project	Method	Subset									
			FULL		DELETION		3-op		2-op		1-op	
			#	% Eq.	#	% Eq.	#	% Eq.	#	% Eq.	#	% Eq.
–	Bisect	sqrt	212	10%	24	4%	23	4%	22	5%	6	17%
–	Triangle	classify	463	9%	–	–	–	–	52	4%	34	0%
1.7k	XStream	decodeName	305	17%	54	28%	52	27%	34	32%	24	38%
1.6k	Commons-Math	gcd	367	19%	–	–	42	10%	40	10%	22	9%
		orthogonal	370	3%	56	0%	52	0%	48	0%	10	0%
4.4k*	Pamvotis	addNode	424	13%	52	8%	46	9%	44	9%	6	0%
		removeNode	108	11%	13	0%	10	0%	8	0%	6	0%
16.3k	Commons-Lang	lastIndexOf	132	8%	–	–	19	5%	18	6%	16	6%
		subarray	93	10%	15	0%	14	0%	10	0%	8	0%
		toMap	96	18%	19	11%	16	13%	12	17%	10	0%
		capitalize	137	18%	25	4%	24	4%	–	–	14	7%
		wrap	330	11%	52	6%	51	6%	32	9%	16	0%
Total			3037	12%	424	8%	401	8%	334	9%	172	8%

subject only produced DELETION mutants belonging to the *AOD* and *RemoveConditionals* operators; we do not report numbers for subsequent incremental subsets (3-op subset and the DELETION set), since they would contain no additional mutants.

8.2. Method

We evaluated the cost and reliability[3] of the DELETION set and the incremental subsets proposed in Section 7. We had the following experimental design:

- **Independent variables**: Mutation operator subsets: the DELETION, 3-op, 2-op, and 1-op subsets
- **Dependent variables**: Cost and reliability of the subsets being evaluated
- **Subjects**: 12 methods from 6 Java projects

We measured each subset's *computational cost* as the number of mutants it would produce, as a proportion of the number that would have been produced by the FULL set. This relative cost measures allow comparisons of cost savings across subject programs, which are of different sizes.

Evaluating a subset's *reliability* entails measuring its strength as a test adequacy criterion. In other words, if a tester were to stop testing after satisfying the given subset of mutation operators, how good would their tests be? Analysis was carried out on each project as follows:

1. We generated a complete set of mutants M using the FULL set of operators, and discarded all equivalent mutants that were identified by Kintis et al. and ourselves.
2. We produced a matrix of detectable mutants and detecting tests by testing each mutant in M using a mutation-adequate test suite T (provided by Kintis et al.).
3. For each operator subset S being evaluated, we constructed a subset-adequate test suite $T_S \subseteq T$ and measured its mutation coverage, i.e., the proportion of mutants in M that were detected by T_S. This measurement is the subset's *reliability*.
   We did this by choosing the smallest subset of tests that detected all mutants generated by S. The detecting tests were chosen from the mutation matrix. Note that there can exist multiple such smallest test sets, and results are dependent on the order in which tests are selected and the power of individual tests. Following Delamaro et al. (2014), we shuffled the available test set to minimize order-related bias, and we repeated the test selection process 3 times per project to minimize power-related bias, selecting the test set with the median mutation coverage.
4. The mutation coverage of subset S was measured as the percentage of mutants in M (the complete set of mutants) that were detected by the subset-adequate test suite T_S.

[3] Note the change in terminology from "accuracy" to "reliability". Instead of statistically predicting FULL mutation adequacy, we concretely measure how far toward it we get when we satisfy a given operator subset.

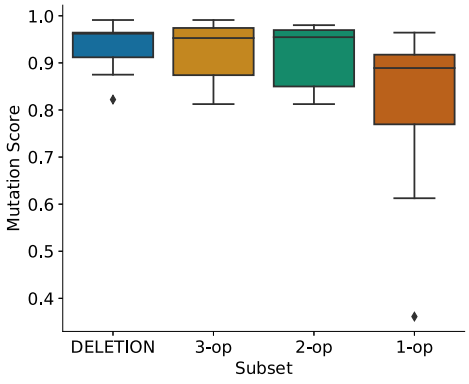

Fig. 5. Mutation coverage: Proportion of FULL mutants detected by the subset-adequate test suite.

8.3. Result

Subset performance is in agreement with the results obtained in the Core study (Section 7). Results are summarized in Figs. 5 and 6, as distributions across subjects. Fig. 5 depicts the mutation coverages for each subset. That is, for each subset S, it shows the proportion of FULL mutants that were detected by the subset-adequate test suite T_S. For example, the DELETION-adequate test suite detected a median 96% of mutants from the FULL set of operators. The 2-op and 1-op subsets are nearly as effective at achieving mutation coverage as the DELETION set, detecting a median 95% and 89% of mutants, respectively. The 3-op subset brings little to no improvement over the previous subset, i.e., it appears that the *NonVoidMethodCalls* operator brings little additional value over the *RemoveConditionals* and *AOD* operators. This is in agreement with observations from **RQ4** (Section 7.2).

In addition to the mutation coverage, we consider the *computational cost* of the subsets under study, i.e., the number of mutants that will be produced and tested. Fig. 6 depicts the

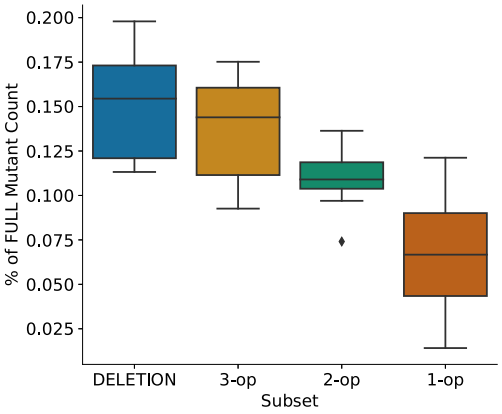

Fig. 6. Computational cost: Number of mutants produced by each subset, expressed as a proportion of the FULL number of mutants.

number of mutants produced by each subset as a proportion of the number that would have been produced by the FULL set. We observe cost decreases in the expected order, with subsets having the following median costs: DELETION (15%) > 3-op (14%) > 2-op (11%) > 1-op (6%). We can see that the 3-op subset brings little cost reduction over the DELETION set, while the 2-op subset and 1-op subset bring better cost savings over their respective preceding incremental subsets. These results are in keeping with observations from Sections 6 and 7. Observe that proportions are close to those seen in Table 4 (see the column titled "*% of FULL*").

Finally, we measured the proportion of all equivalent mutants that were produced by the DELETION set and each incremental subset. Percentages for individual subjects can be seen in Table 5. The DELETION set produced a median of 6% of all equivalent mutants (i.e., it avoided producing 94% of the equivalent mutants that would otherwise have been produced). This is in keeping with previous findings (Untch, 2009; Delamaro et al., 2014). This performance can only improve when we eliminate operators from the DELETION set. The 3-op and 2-op subsets also produced a median of 6% of all equivalent mutants. The 1-op subset was most impressive in this regard, producing a median 0 equivalent mutants. We can see in Table 5 that the 1-op subset produced 0 equivalent mutants for 7 out of 12 projects. The reduced propensity of the DELETION set and its incremental subsets for producing equivalent mutants bodes well for its potential utility as a feedback mechanism for student-written software tests.

9. Discussion

We discuss the implications of our findings.

9.1. Choosing a subset of operators

What subset of DELETION operators is the most cost-effective in general? We have seen that the DELETION set, though cheaper than the FULL and SUFFICIENT sets (Fig. 3), still includes a component of unproductive cost. The DELETION operators' ability to approximate FULL coverage improves and then tapers off after an appropriate subset of mutation operators has been chosen. Based on the changing R^2 values in Table 4, one might conclude that the critical point is after the second (*AOD*) or third (*NonVoidMethodCalls*) operator is added to the model. However, including *NonVoidMethodCalls* increases the total cost of the previous two operators by nearly 50%, but only explains an additional 3% of the variance, which is a relatively small improvement over the previous subset. We believe that this large additional cost is not worth the value added to the model. Selective mutation with *NonVoidMethodCalls* takes a median 38 s and 46 s for SG3 and SG4, respectively. These running times are far beyond our target time of 30 s as specified in Section 3. These diminishing returns were also observed in the Validation study.

Fig. 4 is a "zoomed in and panned right" version of Fig. 3, with the FULL and SUFFICIENT sets excluded, and the 1-op and 2-op subsets included. For submissions in SG2–SG4, subsets of the DELETION set are able to bring huge cost savings with small losses in accuracy. Similar results were seen in the Validation study—in Figs. 5 and 6, we see that the 2-op subset achieved a median mutation coverage of 95%, while consistently producing under 15% of the total number of mutants. Inclusion of the *NonVoidMethodCalls* operator substantially increases the cost with no improvements in effectiveness. Taking cost and effectiveness into account, we conclude: **In the educational context, the 2-operator subset is the most practical set for fast and effective mutation analysis**.

Why does RemoveConditionals perform so effectively by itself? We found that *RemoveConditionals* alone was effective at approximating FULL coverage for the groups of larger submissions SG2–SG4 (Section 7.2). This operator replaces conditionals with Boolean literals, effectively excluding (or ensuring the execution of) all statements guarded by a condition. Mutation analysis using this operator has strong ties to *object branch coverage* (OBC), one of the strongest forms of code coverage for Java programs. OBC requires students to write tests that exercise every Boolean condition generated in their solution's compiled bytecode. *RemoveConditionals* can be seen as a stronger form of this measure, since it is sensitive not only to the *execution* of conditions, but also to the *propagation* of program state or output from those conditions to the tests.

This finding may be clearer in light of the kinds of programs we investigated. Our corpus included submissions from an upper-level Data Structures & Algorithms (CS3) course, nearly all of which were clustered in submission groups SG2–SG4. These projects require significant control flow components to implement complex behaviors, so it is plausible that the conditions in the control flow logic would be the most critical aspects of quality testing. Similarly, in the Validation study, *RemoveConditionals* was a highly effective lone operator, achieving a median mutation coverage of 89% (Fig. 5). That the operator tends to produce few equivalent mutants (median = 0) only serves to increase its attractiveness as an option for selective mutation. We recommend: **AATs should use the 1-op subset for larger and more complex projects** (SLoC > 666).

How does one evaluate tests for smaller submissions (group SG1)? In Section 7.2, notice that the 3-op subset—or indeed, the entire DELETION set—is unable to achieve a good approximation of coverage under the FULL set for smaller submissions. This throws into question whether selective mutation is an effective approach for these projects. The time to run mutation analysis on these submissions is so low ($\mu = 22.2$ s, $\sigma = 16.8$ s) that a cheaper approximation of the FULL set is unnecessary. In light of these differences in the effectiveness and cost of mutation testing on our data set, we report: **AATs may use the FULL set of mutation operators for small and simple** submissions (SLoC $\leq$ 341).

However, it is worth considering whether mutation testing is an over-engineered test evaluation strategy for smaller programs of such minimal complexity. The large number of mutants produced could potentially overwhelm beginning CS students. Where possible, instructors might opt for all-pairs methods, or they could curate a set of faulty implementations for students to detect with their tests (Politz et al., 2014; Wrenn and Krishnamurthi, 2019).

9.2. Operationalizing feedback

What might feedback based on mutation analysis look like? Ultimately, the goal of our research is to improve the quality of student-written test suites. Mutation analysis only furthers this goal if the students get feedback about the process in some way. Similar to code coverage, it is easy to generate feedback for students by highlighting the lines of code that contain undetected mutations. Consider the code snippet in Listing 1. The *AOD* mutation operator was applied to the highlighted line (line 2), changing it to `return i`. The highlight indicates that *all tests passed* even with the specified mutation in place. In other words, no test behaves differently whether the output is `i` or `i * i`. A combination of information—the highlighted line and the exact mutation that was applied—gives the student an explicit strategy for improving the test suite based on the provided feedback, i.e., write a test that makes an assertion about the function's return value. Similar feedback may be devised for other mutation operators.

```
1 public int probeSquare(int i) {
2   return i * i;
3   // Tests did not check the use of this arithmetic
4   // expression.
5 }
```

Listing 1: A snippet highlighting a line that contained a surviving mutant (similar to reports emitted by PIT).

In addition to the empirically validated benefits of DELETION mutation—namely, its cost-effectiveness and reduced propensity for producing equivalent mutants—we believe that DELETION mutation offers a third potential benefit over existing mutation approaches: *simplicity of feedback*. When a student is faced with an undetected DELETION mutant, there are two possibilities: the deleted code is unchecked, or it is redundant. For other mutation operators, the situation is more complex. An undetected mutant may mean that the test suite has failed to cover a subtle, potentially obscure, possibly unreachable edge case (e.g., see Yao et al.'s (2014) work on equivalent and "stubborn" mutants). Constructing a test case to detect an undetected mutant could require knowledge about the specific mutation operator used, which further implies a basic understanding of mutation analysis. DELETION mutation avoids this added complexity by producing fewer mutants, which are more obvious and more likely to be actionable.

How does this compare with condition coverage? Offutt and Voas showed that condition coverage is subsumed by mutation coverage, i.e., if a test set satisfies mutation coverage, it also satisfies condition coverage. Condition coverage requires that all conditions—including individual conditions that are joined with conjunctions or disjunctions to form compound decisions—be made to evaluate to `true` or `false` at least once (Myers et al., 2011). This criterion is subsumed by the *RemoveConditionals* mutation operator. That is, detecting all *RemoveConditionals* operators means that all conditions must have been executed at least once by the test suite.

Condition coverage is satisfied when when students execute the code, regardless of whether or not they check that the behavior is correct. In stark contrast, mutation testing requires the test suite to recognize that the mutant has failed to be detected by the test suite. Specific to our corpus of submissions, consider Listing 1 again. It depicts a function from a project in our corpus that achieved complete condition coverage but zero mutation coverage. It is a simple probing function that helps determine a record's position in a hash table. The student's tests only verified that records existed in the hash table after insertion, but never that records were inserted at the right positions. Code coverage measures were unable to detect this deficiency, since the `probeSquare` function was executed ("covered") during the insertion process.

Indeed, this discrepancy was reflected in submissions across our entire corpus. Condition coverage scores tended to cluster close to the 100% mark ($\mu = 0.98, \sigma = 0.03$). Notice that students almost universally had good condition coverage scores. This may be because they were graded in part on their coverage scores. In contrast, mutation coverage using only the 2-op Subset (*RemoveConditionals* and *AOD*) tended to be far lower ($\mu = 0.81, \sigma = 0.18$). These outcomes are worse than they seem, since in our experience condition coverage of at least 80% is fairly trivial to reach (sometimes through pathological or bad-faith tests).

9.3. Future directions

This paper sets the stage for educators to offer students incremental feedback based on mutation analysis. As our particular goal, we reduced the cost of the analysis such that it can produce reliable feedback for students' test suites in under 30 s on typical institutional AAT hardware. But our approach covers contexts beyond this experimental setting. The subsets of DELETION operators evaluated in Section 7 provide incrementally cheaper approaches with which to provide mutation-based incremental feedback. These approaches can be selectively applied to a diverse set of contexts, governed by institutional needs, budgets, or other factors.

In addition to its cost and effectiveness, it is important to evaluate the educational value of mutation analysis. In particular, how useful is mutation analysis to undergraduate CS students? How does the type or number of mutation operators affect students' ability to react to feedback? What level of programming expertise do students need to benefit from mutation-based feedback? As a preliminary step toward this effort, we held 9 interviews with third-year undergraduate CS students, who indicated that they found feedback based on mutation analysis to be useful and actionable. They were able to construct specific test cases that would detect mutants generated from their own code. Further work is needed to determine the degree to which mutation feedback is useful to CS students and the best way in which to present feedback.

10. Threats to validity

10.1. Internal validity

Running time distributions may have been affected by differences in the proportion of timed-out mutants between submission groups. In particular, the increase in median running times over SG1, SG2, and SG3 (8 s, 12 s, and 44 s, respectively) were more pronounced than the increase in running times from SG3 to SG4 (44 s to 53 s). Further investigation revealed that submissions in SG3 (specifically, those submitted to Project 5 in the CS 3 corpus) contained an atypically high percentage of mutants that timed out. This drove up the median running time for mutation analysis for the submission group. We could not identify a systematic cause. However, we do not believe this to be a serious threat to the overall validity of our findings; we used two cost measures for corroboration (mutant count, running time) as recommended by Guizzo et al. (2020), and our choice and ordering of operators are not affected by the higher-than-expected median running time for SG3.

10.2. External validity

As with any research on software, our results are only as general as the students and programs we study. We have striven to mitigate this threat, by studying (1) a corpus of 1389 programs of various sizes and complexities, implementing 7 requirement specifications, and (2) a smaller corpus of randomly chosen codebases from real-world projects and the mutation testing literature. That our findings from both datasets are largely in agreement suggests that these results may be free of this particular threat. That said, for reasons described in Section 5, we studied only Java programs. It is possible that our findings do not generalize to other programming languages or mutation tools. For example, in Java bytecode, characters (`char`) are represented as integers, while in other languages (like Python), they are not. As a result, for similar operations in the two languages (e.g., assigning a character to a variable), different mutation operators would be used.

10.3. Construct validity

We studied PIT, a mature mutation testing tool available for Java. As described in Section 5, it is currently the most robust, easy-to-use, and practical mutation testing tool for the JVM, making it the most practical choice for fast feedback based on mutation analysis. Nevertheless, we do not use any direct measures of test adequacy here, such as defect-detection capability. Instead, we use coverage on the `FULL` set of PIT operators as a proxy for measuring test quality, relying on existing theoretical (DeMillo et al., 1978; Offutt and Voas, 1996) and empirical (Offutt, 1992; Andrews et al., 2005; Just et al., 2011) results on the validity and strength of mutation analysis. We are encouraged by the fact that Shams performed an assessment of deletion mutators in terms of measuring test suite bug detection ability and found them to be more effective than code coverage measures, but these results still depend on the validity of the relationship between mutation analysis and test quality.

11. Conclusion

The pedagogy of software testing is hindered by widespread reliance on weak test adequacy criteria for the purposes of assessment and feedback. Mutation analysis has been proposed as an alternative solution, but its computational cost is a significant limiting factor. We have devised a cost-effective mutation strategy to produce fast, accurate, and incremental feedback on the quality of student-written software tests. This approach provides a better assessment of how well software tests check expected behaviors, and can be used to generate feedback for students. We improved upon the most efficient mutation operator set previously proposed in the literature, the `DELETION` set. For the projects we studied, the *RemoveConditionals* and *AOD* operators produced results comparable to the most stringent set of operators at *1/10*th the cost of comprehensive mutation, and *less than half* the cost of deletion mutation. These cost savings will enable the use of mutation analysis in the automated assessment tools frequently used in computer science courses. Future work will evaluate the efficacy of mutation analysis as a feedback mechanism in the classroom.

CRediT authorship contribution statement

Ayaan M. Kazerouni: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **James C. Davis:** Conceptualization, Methodology, Formal analysis, Writing - original draft, Writing - review & editing, Visualization. **Arinjoy Basak:** Data curation, Software, Investigation, Writing- review & editing. **Clifford A. Shaffer:** Conceptualization, Methodology, Resources, Writing - original draft, Writing - review & editing, Supervision, Funding acquisition. **Francisco Servant:** Conceptualization, Writing - original draft, Writing - review & editing, Visualization, Supervision. **Stephen H. Edwards:** Conceptualization, Methodology, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to the National Science Foundation, United States of America for their support under grants DUE-1625425 and DLR-1740765. We thank the anonymous peer reviewers whose feedback improved this paper.

References

Aaltonen, K., Ihantola, P., Seppälä, O., 2010. Mutation analysis vs. Code coverage in automated assessment of students' testing skills. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. In: OOPSLA '10, ACM, New York, NY, USA, pp. 153–160. http://dx.doi.org/10.1145/1869542.1869567, URL: http://doi.acm.org/10.1145/1869542.1869567.

Ammann, P., Offutt, J., 2008. Introduction to Software Testing, first ed. Cambridge University Press, USA.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments?. In: Proceedings of the 27th International Conference on Software Engineering. In: ICSE '05, ACM, New York, NY, USA, pp. 402–411. http://dx.doi.org/10.1145/1062455.1062530, URL: http://doi.acm.org/10.1145/1062455.1062530.

Aniche, M., Hermans, F., van Deursen, A., 2019. Pragmatic software testing education. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. In: SIGCSE '19, ACM, New York, NY, USA, pp. 414–420. http://dx.doi.org/10.1145/3287324.3287461, URL: http://doi.acm.org/10.1145/3287324.3287461.

Azevedo, R., Bernard, R.M., 1995. A meta-analysis of the effects of feedback in computer-based instruction. J. Educ. Comput. Res. 13 (2), 111–127. http://dx.doi.org/10.2190/9LMD-3U28-3A0G-FTQT, URL: http://journals.sagepub.com/doi/10.2190/9LMD-3U28-3A0G-FTQT.

Black, P., Wiliam, D., 1998. Assessment and classroom learning. Assess. Educ.: Princ. Policy Pract. 5 (1), 7–74. http://dx.doi.org/10.1080/0969595980050102.

Bozdogan, H., 1987. Model selection and akaike's information criterion (AIC): The general theory and its analytical extensions. Psychometrika 52 (3), 345–370.

Budd, T.A., Angluin, D., 1982. Two notions of correctness and their relation to testing. Acta Inform. 18 (1), 31–45. http://dx.doi.org/10.1007/BF00625279.

Buffardi, K., Valdivia, P., Rogers, D., 2019. Measuring unit test accuracy. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. In: SIGCSE '19, ACM, New York, NY, USA, pp. 578–584. http://dx.doi.org/10.1145/3287324.3287351, URL: http://doi.acm.org/10.1145/3287324.3287351.

Carver, J., Kraft, N.A., 2011. Evaluating the testing ability of senior-level computer science students. In: 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T). pp. 169–178. http://dx.doi.org/10.1109/CSEET.2011.5876084.

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. PIT: A practical mutation testing tool for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis. In: ISSTA 2016, ACM, New York, NY, USA, pp. 449–452. http://dx.doi.org/10.1145/2931037.2948707, URL: http://doi.acm.org/10.1145/2931037.2948707.

Delahaye, M., du Bousquet, L., 2013. A comparison of mutation analysis tools for java. In: 2013 13th International Conference on Quality Software. IEEE, pp. 187–195. http://dx.doi.org/10.1109/QSIC.2013.47.

Delamaro, M., Offutt, J., Ammann, P., 2014. Designing deletion mutation operators. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE, pp. 11–20. http://dx.doi.org/10.1109/ICST.2014.12.

DeMillo, R., Guindi, D.S., McCracken, W.M., Offutt, A.J., King, K.N., 1988. An extended overview of the mothra software testing environment. In: [1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis. IEEE, pp. 142–151. http://dx.doi.org/10.1109/WST.1988.5369.

DeMillo, R., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41. http://dx.doi.org/10.1109/C-M.1978.218136.

Deng, L., Offutt, J., Li, N., 2013. Empirical evaluation of the statement deletion mutation operator. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, pp. 84–93. http://dx.doi.org/10.1109/ICST.2013.20.

Derezińska, A., 2016. Evaluation of deletion mutation operators in mutation testing of c# programs. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (Eds.), Dependability Engineering and Complex Systems. Springer International Publishing, Cham, pp. 97–108.

Edwards, S.H., 2004. Using software testing to move students from trial-and-error to reflection-in-action. SIGCSE Bull. 36 (1), 26–30. http://dx.doi.org/10.1145/1028174.971312.

Edwards, S.H., Shams, Z., 2014. Comparing test quality measures for assessing student-written tests. In: Companion Proceedings of the 36th International Conference on Software Engineering. In: ICSE Companion 2014, ACM, New York, NY, USA, pp. 354–363. http://dx.doi.org/10.1145/2591062.2591164, URL: http://doi.acm.org/10.1145/2591062.2591164.

Edwards, S.H., Shams, Z., Cogswell, M., Senkbeil, R.C., 2012. Running students' software tests against each others' code: New life for an old "gimmick". In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education. In: SIGCSE '12, ACM, New York, NY, USA, pp. 221–226. http://dx.doi.org/10.1145/2157136.2157202, URL: http://doi.acm.org/10.1145/2157136.2157202.

Edwards, S.H., Snyder, J., Pérez-Quiñones, M.A., Allevato, A., Kim, D., Tretola, B., 2009. Comparing effective and ineffective behaviors of student programmers. In: Proceedings of the Fifth International Workshop on Computing Education Research Workshop. In: ICER '09, ACM, New York, NY, USA, pp. 3–14. http://dx.doi.org/10.1145/1584322.1584325, URL: http://doi.acm.org/10.1145/1584322.1584325.

Goldwasser, M.H., 2002. A gimmick to integrate software testing throughout the curriculum. In: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education. In: SIGCSE '02, ACM, New York, NY, USA, pp. 271–275. http://dx.doi.org/10.1145/563340.563446, URL: http://doi.acm.org/10.1145/563340.563446.

Goodenough, J., Gerhart, S.L., 1975. Toward a theory of test data selection. IEEE Trans. Softw. Eng. SE-1 (2), 156–173. http://dx.doi.org/10.1109/TSE.1975.6312836.

Gopinath, R., Ahmed, I., Alipour, M.A., Jensen, C., Groce, A., 2017. Does choice of mutation tool matter?. Softw. Qual. J. 25 (3), 871–920. http://dx.doi.org/10.1007/s11219-016-9317-7, URL: http://link.springer.com/10.1007/s11219-016-9317-7.

Guizzo, G., Sarro, F., Harman, M., 2020. Cost measures matter for mutation testing study validity. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 1127–1139. http://dx.doi.org/10.1145/3368089.3409742.

Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, ACM, New York, NY, USA, pp. 435–445. http://dx.doi.org/10.1145/2568225.2568271, URL: http://doi.acm.org/10.1145/2568225.2568271.

Ivanković, M., Petrović, G., Just, R., Fraser, G., 2019. Code coverage at google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, ACM, New York, NY, USA, pp. 955–963. http://dx.doi.org/10.1145/3338906.3340459, URL: http://doi.acm.org/10.1145/3338906.3340459.

Jackson, D., Usher, M., 1997. Grading student programs using ASSYST. In: Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education. In: SIGCSE '97, ACM, New York, NY, USA, pp. 335–339. http://dx.doi.org/10.1145/268084.268210, URL: http://doi.acm.org/10.1145/268084.268210.

Jenks, G.F., 1967. The data model concept in statistical mapping. Int. Yearb. Cartogr. 7, 186–190.

Jenks, G., 1977. Optimal Data Classification for Choropleth Maps Occasional Paper No 2. University of Kansas, Department of Geography.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678. http://dx.doi.org/10.1109/TSE.2010.62.

Jones, E.L., 2000. Software testing in the computer science curriculum – a holistic approach. In: Proceedings of the Australasian Conference on Computing Education - ACSE '00. ACM Press, New York, New York, USA, pp. 153–157. http://dx.doi.org/10.1145/359369.359392, URL: http://portal.acm.org/citation.cfm?doid=359369.359392.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing?. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2014, ACM, New York, NY, USA, pp. 654–665. http://dx.doi.org/10.1145/2635868.2635929, URL: http://doi.acm.org/10.1145/2635868.2635929.

Just, R., Schweiggert, F., Kapfhammer, G.M., 2011. MAJOR: An efficient and extensible tool for mutation analysis in a java compiler. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). pp. 612–615. http://dx.doi.org/10.1109/ASE.2011.6100138.

Kazerouni, A.M., Edwards, S.H., Shaffer, C.A., 2017. Quantifying incremental development practices and their relationship to procrastination. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. In: ICER '17, Association for Computing Machinery, New York, NY, USA, pp. 191–199. http://dx.doi.org/10.1145/3105726.3106180.

Kazerouni, A.M., Shaffer, C.A., Edwards, S.H., Servant, F., 2019. Assessing incremental testing practices and their impact on project outcomes. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. In: SIGCSE '19, Association for Computing Machinery, New York, NY, USA, pp. 407–413. http://dx.doi.org/10.1145/3287324.3287366.

King, K.N., Offutt, A.J., 1991. A fortran language system for mutation-based software testing. Softw. Pract. Exper. 21 (7), 685–718. http://dx.doi.org/10.1002/spe.4380210704.

Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., 2016. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 147–156. http://dx.doi.org/10.1109/SCAM.2016.28.

Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Le Traon, Y., 2018. How effective are mutation testing tools? An empirical analysis of java mutation testing tools with manual analysis and real faults. Empir. Softw. Eng. 23 (4), 2426–2463. http://dx.doi.org/10.1007/s10664-017-9582-5.

der Kleij, F.M.V., Feskens, R.C.W., Eggen, T.J.H.M., 2015. Effects of feedback in a computer-based learning environment on students' learning outcomes: A meta-analysis. Rev. Educ. Res. 85 (4), 475–511. http://dx.doi.org/10.3102/0034654314564881.

Laurent, T., Papadakis, M., Kintis, M., Henard, C., Traon, Y.L., Ventresque, A., 2017. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 430–435. http://dx.doi.org/10.1109/ICST.2017.47.

Lethbridge, T.C., 2000. Priorities for the education and training of software engineers. J. Syst. Softw. 53 (1), 53–71. http://dx.doi.org/10.1016/S0164-1212(00)00009-1, URL: http://www.sciencedirect.com/science/article/pii/S0164121200000091.

Lloyd, S., 1982. Least squares quantization in PCM. IEEE Trans. Inform. Theory 28 (2), 129–137. http://dx.doi.org/10.1109/TIT.1982.1056489.

Ma, Y.-S., Offutt, J., Kwon, Y.R., 2005. Mujava: an automated class mutation system. Softw. Test. Verif. Reliab. 15 (2), 97–133. http://dx.doi.org/10.1002/stvr.308.

Mathur, A., 1991. Performance, effectiveness, and reliability issues in software testing. In: [1991] Proceedings the Fifteenth Annual International Computer Software Applications Conference. pp. 604–605. http://dx.doi.org/10.1109/CMPSAC.1991.170248.

Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.

Offutt, A.J., 1992. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. 1 (1), 5–20. http://dx.doi.org/10.1145/125489.125473.

Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. 5 (2), 99–118. http://dx.doi.org/10.1145/227607.227610, URL: http://doi.acm.org/10.1145/227607.227610.

Offutt, A.J., Voas, J.M., 1996. Subsumption of Condition Coverage Techniques by Mutation Testing. Tech. Rep. ISSE-TR-96-100, Department of Information and Software Systems Engineering, George Mason University.

Papancea, A., Spacco, J., Hovemeyer, D., 2013. An open platform for managing short programming exercises. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research. In: ICER '13, ACM, New York, NY, USA, pp. 47–52. http://dx.doi.org/10.1145/2493394.2493401, URL: http://doi.acm.org/10.1145/2493394.2493401.

Pettit, R., Homer, J., Gee, R., Mengel, S., Starbuck, A., 2015. An empirical study of iterative improvement in programming assignments. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education. In: SIGCSE '15, Association for Computing Machinery, New York, NY, USA, pp. 410–415. http://dx.doi.org/10.1145/2676723.2677279.

Pettit, R., Prather, J., 2017. Automated assessment tools: Too many cooks, not enough collaboration. J. Comput. Sci. Coll. 32 (4), 113–121, URL: http://dl.acm.org/citation.cfm?id=3055338.3079060.

Politz, J.G., Krishnamurthi, S., Fisler, K., 2014. In-flow peer-review of tests in test-first programming. In: Proceedings of the Tenth Annual Conference on International Computing Education Research. In: ICER '14, Association for Computing Machinery, New York, NY, USA, pp. 11–18. http://dx.doi.org/10.1145/2632320.2632347.

Radermacher, A., Walia, G., 2013. Gaps between industry expectations and the abilities of graduates. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. In: SIGCSE '13, Association for Computing Machinery, New York, NY, USA, pp. 525–530. http://dx.doi.org/10.1145/2445196.2445351.

Schuler, D., Zeller, A., 2009. Javalanche: Efficient mutation testing for java. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. In: ESEC/FSE '09, ACM, New York, NY, USA, pp. 297–298. http://dx.doi.org/10.1145/1595696.1595750, URL: http://doi.acm.org/10.1145/1595696.1595750.

Schwarz, G., 1978. Estimating the dimension of a model. Ann. Statist. 6 (2), 461–464. http://dx.doi.org/10.1214/aos/1176344136.

Seabold, S., Perktold, J., 2010. Statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference.

Shams, Z., 2015. Automated Assessment of Student-written Tests Based on Defect-detection Capability (Ph.D. thesis). Virginia Tech, Blacksburg, VA, URL: http://hdl.handle.net/10919/52024.

Shams, Z., Edwards, S.H., 2013. Toward practical mutation analysis for evaluating the quality of student-written software tests. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research. In: ICER '13, ACM, New York, NY, USA, pp. 53–58. http://dx.doi.org/10.1145/2493394.2493402, URL: http://doi.acm.org/10.1145/2493394.2493402.

Siami Namin, A., Andrews, J.H., Murdoch, D.J., 2008. Sufficient mutation operators for measuring test effectiveness. In: Proceedings of the 30th International Conference on Software Engineering. In: ICSE '08, ACM, New York, NY, USA, pp. 351–360. http://dx.doi.org/10.1145/1368088.1368136, URL: http://doi.acm.org/10.1145/1368088.1368136.

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., Padua-Perez, N., 2006. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. In: ITICSE '06, ACM, New York, NY, USA, pp. 13–17. http://dx.doi.org/10.1145/1140124.1140131, URL: http://doi.acm.org/10.1145/1140124.1140131.

Spacco, J., Pugh, W., 2006. Helping students appreciate test-driven development (TDD). In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications. In: OOPSLA '06, ACM, New York, NY, USA, pp. 907–913. http://dx.doi.org/10.1145/1176617.1176743.

Untch, R.H., 2009. On reduced neighborhood mutation analysis using a single mutagenic operator. In: Proceedings of the 47th Annual Southeast Regional Conference. In: ACM-SE 47, ACM, New York, NY, USA, pp. 71:1–71:4. http://dx.doi.org/10.1145/1566445.1566540.

Wang, T., Su, X., Ma, P., Wang, Y., Wang, K., 2011. Ability-training-oriented automated assessment in introductory programming course. Comput. Educ. 56 (1), 220–226. http://dx.doi.org/10.1016/j.compedu.2010.08.003.

Wong, W.E., Mathur, A.P., 1995. Fault detection effectiveness of mutation and data flow testing. Softw. Qual. J. 4 (1), 69–83. http://dx.doi.org/10.1007/BF00404650, URL: http://link.springer.com/10.1007/BF00404650.

Wrenn, J., Krishnamurthi, S., 2019. Executable examples for programming problem comprehension. In: Proceedings of the 2019 ACM Conference on International Computing Education Research. In: ICER '19, ACM, New York, NY, USA, pp. 131–139. http://dx.doi.org/10.1145/3291279.3339416, URL: http://doi.acm.org/10.1145/3291279.3339416.

Wrenn, J., Krishnamurthi, S., Fisler, K., 2018. Who tests the testers?. In: Proceedings of the 2018 ACM Conference on International Computing Education Research. In: ICER '18, ACM, New York, NY, USA, pp. 51–59. http://dx.doi.org/10.1145/3230977.3230999, URL: http://doi.acm.org/10.1145/3230977.3230999.

Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. ACM Press, Hyderabad, India, pp. 919–930. http://dx.doi.org/10.1145/2568225.2568265, URL: http://dl.acm.org/citation.cfm?doid=2568225.2568265.

Ayaan M. Kazerouni is an Assistant Professor in the Department of Computer Science and Software Engineering at California Polytechnic State University. He is interested in computing education and software engineering with a focus in self-regulated software development and software testing. Ayaan obtained a Ph.D. in Computer Science at Virginia Tech in 2020, and a BS in Computer Science at the University of West Georgia in 2015. Find him on the web at https://ayaankazerouni.github.io/.

James C. Davis is an Assistant Professor in the department of Electrical and Computer Engineering at Purdue University. He received his Ph.D. from Virginia Tech. His research focuses on improving the correctness, security, and overall quality of computer systems. His work identifies best practices and common errors through qualitative and quantitative empirical methods. Find him on the web at https://davisjam.github.io/.

Arinjoy Basak is a Ph.D. student working with Dr. Clifford A. Shaffer at the Department of Computer Science, Virginia Tech. He earned his B.E. in Computer Science from IIEST Shibpur in Howrah, India in 2016, and his M.S. in Computer Science from Virginia Tech in 2019. His interests include educational data mining and analyzing student behaviors. His thesis focuses on building interactive tutorial systems for understanding mathematical problem-solving behaviors of students in undergraduate engineering mechanics courses. Find him on the web at https://arinjoy-basak.github.io/.

Clifford A. Shaffer is Professor and Associate Department Head in the Department of Computer Science at Virginia Tech. He received his Ph.D. from the University of Maryland, and is an ACM Distinguished Educator. His current research promotes online technologies for learning. He directs the OpenDSA project, an open-source, online collection of materials and infrastructure for creating eTextbooks on Computer Science topics. He also studies integration of online educational tools, and developing tools for analyzing learner analytics. Find him on the web at https://people.cs.vt.edu/shaffer.

Francisco Servant is Assistant Professor in the Department of Computer Science at Virginia Tech. He received a Ph.D. in Software Engineering from the University of California, Irvine, and a B.S. in Computer Science from the University of Granada, Spain. His research focuses on software development productivity and software quality. Find him on the web at https://fservant.com/.

Stephen H. Edwards is a Professor and the Associate Department Head for Undergraduate Studies in the Department of Computer Science at Virginia Tech, where he has been teaching since 1996. He received his B.S. in electrical engineering from Caltech, and M.S. and Ph.D. degrees in computer and information science from The Ohio State University. His research interests are in computer science education, software engineering, automated testing, the use of formal methods in programming languages, and component-based approaches to software engineering and reuse. Find him on the web at https://people.cs.vt.edu/edwards.