Prioritizing Mutants to Guide Mutation Testing

Samuel J. Kaufman
kaufmans@cs.washington.edu
University of Washington
Seattle, Washington, USA

Ryan Featherman
feathr@cs.washington.edu
University of Washington
Seattle, Washington, USA

Justin Alvin
jalvin@umass.edu
University of Massachusetts
Amherst, Massachusetts, USA

Bob Kurtz
rkurtz22@gmu.edu
George Mason University
Fairfax, Virginia, USA

Paul Ammann
pammann@gmu.edu
George Mason University
Fairfax, Virginia, USA

René Just
rjust@cs.washington.edu
University of Washington
Seattle, Washington, USA

ABSTRACT

Mutation testing offers concrete test goals (mutants) and a rigorous test efficacy criterion, but it is expensive due to vast numbers of mutants, many of which are neither useful nor actionable. Prior work has focused on selecting representative and sufficient mutant subsets, measuring whether a test set that is mutation-adequate for the subset is equally adequate for the entire set. However, no known industrial application of mutation testing uses or even computes mutation adequacy, instead focusing on iteratively presenting very few mutants as concrete test goals for developers to write tests.

This paper (1) articulates important differences between mutation analysis, where measuring mutation adequacy is of interest, and mutation testing, where mutants are of interest insofar as they serve as concrete test goals to elict effective tests; (2) introduces a new measure of mutant usefulness, called test completeness advancement probability (TCAP); (3) introduces an approach to prioritizing mutants by incrementally selecting mutants based on their predicted TCAP; and (4) presents simulations showing that TCAP-based prioritization of mutants advances test completeness more rapidly than prioritization with the previous state-of-the-art.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**.

KEYWORDS

mutation testing, mutant selection, mutant utility, test completeness advancement probability, TCAP, machine learning

ACM Reference Format:
Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510187

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510187

1 INTRODUCTION

Mutation testing generates a set of program variants called *mutants* and challenges a developer to create tests that *detect* them—that is, distinguishes the variants from the original program. There is strong empirical evidence that mutants are coupled to real faults and that mutant detection is positively correlated with real fault detection [4, 10, 23, 46]. This correlation is stronger than is the case for code coverage criteria commonly used in practice (e.g., statement and branch coverage [19, 55]).

Mutation testing is expensive due to the large number of mutants that can be generated for a given software system. Recent research, focusing on the practicality of presenting mutants as test goals, identified the total number of mutants and the fact that most of them are not useful test goals as key challenges to adoption [7, 46, 48]. Further, equivalent and redundant mutants make it difficult for a developer to assess how close they are to achieving mutation adequacy [32]. *Equivalent* mutants are functionally identical to the original program, and therefore cannot be detected by any test. *Redundant* mutants are functionally identical to other mutants, and are therefore always detected by tests that detect the other mutants.

To make mutation testing feasible for practitioners, it is necessary to select just a few of the numerous mutants produced by current mutation systems. Given that many mutants are not useful, such a selection strategy must be biased toward useful mutants. Furthermore, neither achieving nor measuring mutation adequacy is among the reported desiderata of industrial mutation testing systems, which are instead concerned with iteratively presenting one or a small number of mutants as test goals to incrementally improve test quality over time [7, 45, 46, 48].

This paper proposes a new measure for mutant usefulness, which values mutants according to their likelihood of eliciting a test that advances test completeness, and evaluates the measure's ability to effectively prioritize mutants. Specifically, this paper contributes:

- An articulation of the differences between two mutation use cases—*mutation analysis* vs. *mutation testing*—and implications for empirically evaluating them (Section 3).
- A new measure, *test completeness advancement probability (TCAP)*, that quantifies mutant usefulness (Section 4).
- A mutation testing approach that prioritizes mutants for selection based on their predicted TCAP (Section 5).
- An evaluation showing that, for a variety of different initial test set coverages, prioritizing mutants according to TCAP improves test completeness more rapidly than the previous state-of-the-art (random selection) (Section 6).

2 BACKGROUND

A *mutation operator* is a program transformation rule that generates a *mutant* (i.e., program variant) of a given program based on the occurrence of a particular syntactic element. One example of a mutation operator is the replacement of an instance of the arithmetic operator "`*`" with "`/`". Specifically, if a program contains an expression "`a * b`", for arbitrary expressions "`a`" and "`b`", this mutation operator creates a mutant where "`a / b`" replaces this expression. The *mutation* is the syntactic change that a mutation operator introduces. A mutation operator is applied everywhere it is possible to do so. In the example above, if the arithmetic operator "`*`" occurs multiple times, the mutation operator will create a separate mutant for each occurrence. This paper considers first-order mutants, where each mutant contains exactly one mutation, as opposed to higher-order mutants, where each mutant is the product of multiple applications of mutation operators.

A *mutation operator group* is a set of related mutation operators. For example, the AOR (arithmetic operator replacement) mutation operator group contains all mutation operators that replace an arithmetic operator, including the example above.

A mutant may behave identically to the original program on all inputs. Such a mutant is called an *equivalent mutant* and cannot be detected by any test. As an example, consider the `if` statement in Figure 1a, which determines the smaller value of two integers: `numbers[i] < min`. Replacing the relational operator < with <= results in the equivalent mutant `numbers[i] <= min`—if `numbers[i]` and `min` are equal, assigning either value to `min` is correct, and hence both implementations are equivalent.

A *trivial mutant* is one that is detected due to an exception by every test case that executes the mutated code location. As an example, consider the `for` loop in Figure 1a, which includes a boundary check for an array index (`for int i=1; i<numbers.length; ++i`). If the index variable `i` is used to access the array `numbers` then a mutation `i<=numbers.length` always results in an exception, as the last value for `i` is guaranteed to index `numbers` out of bounds. Hence, this mutant is trivial, as any test that reaches the loop will terminate with an exception.

2.1 Dominator Mutants

Given a set of mutants M, a test set T is *mutation-adequate* with respect to M iff for every non-equivalent mutant m in M, there is some test t in T such that t detects m. However, mutation operators generate far more mutants than are necessary: the cardinality of the mutant set is much larger than the cardinality of the mutation-adequate test set. This redundancy among generated mutants was formally captured in the notion of *minimal mutation* [3]. Given any set of mutants M, a *dominator set* of mutants D is a minimal subset of M such that any test set that is mutation-adequate for D is also mutation-adequate for M.

Computing a dominator set is an undecidable problem, but it is possible to approximate it with respect to a test set [31]—the more comprehensive the test set, the better the approximation. This approximation is not useful to a developer interested in writing tests (they do not yet have a test set with which to approximate a dominator set). However, from a research and analysis perspective, a dominator set provides a precise measure for redundancy in a set

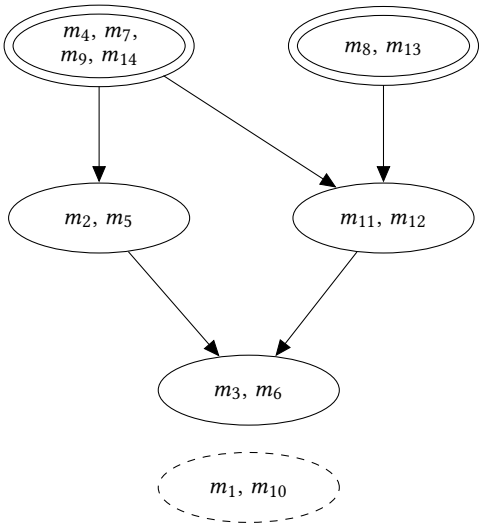
```
1 public int getMin(int[] numbers) {
2   int min = numbers[0];
3   for (int i=1; i < numbers.length; ++i) {
4     if (numbers[i] < min) {
5       min = numbers[i];
6     }
7   }
8   return min;
9 }
```

(a) Mutated relational operator in two different program contexts.

	Tests				Mutant properties					
	t_1	t_2	t_3	t_4	MutOp	Ctx	TCAP	Equi.	Triv.	Dom.
m_1	●	●	●	●	< ⟼ !=	for	0.0	**yes**	—	—
m_2	●	✸	●	✸	< ⟼ ==	for	0.5	—	—	—
m_3	✸	✸	✸	✸	< ⟼ <=	for	0.5	—	**yes**	—
m_4	●	✸	●	●	< ⟼ >	for	1.0	—	—	**yes**
m_5	●	✸	●	✸	< ⟼ >=	for	0.5	—	—	—
m_6	✸	✸	✸	✸	< ⟼ true	for	0.5	—	**yes**	—
m_7	●	✸	●	●	< ⟼ false	for	1.0	—	—	**yes**
m_8	✸	●	●	●	< ⟼ !=	if	1.0	—	—	**yes**
m_9	●	✸	●	●	< ⟼ ==	if	1.0	—	—	**yes**
m_{10}	●	●	●	●	< ⟼ <=	if	0.0	**yes**	—	—
m_{11}	✸	✸	●	●	< ⟼ >	if	1.0	—	—	—
m_{12}	✸	✸	●	●	< ⟼ >=	if	1.0	—	—	—
m_{13}	✸	●	●	●	< ⟼ true	if	1.0	—	—	**yes**
m_{14}	●	✸	●	●	< ⟼ false	if	1.0	—	—	**yes**

Symbols indicate test results: ● indicates that t_i passes on m_j; ✸ indicates that t_i fails on m_j with an assertion failure; ✸ indicates that t_i fails on m_j with an exception (i.e., the mutant crashes during execution).

(b) Test results and mutant properties.



(c) Dynamic mutant subsumption graph for the 14 mutants. Intuitively, mutants high in the graph are dominating other mutants, whereas mutants low in the graph are subsumed.

Figure 1: Motivating example for program context and test completeness advancement probability (TCAP).

of mutants, and hence the dynamic approximation approach is an important research tool for analyzing mutation testing techniques.

Given a finite set of mutants M and a finite set of tests T, mutant m_i *dynamically subsumes* mutant m_j if some test in T detects m_i and every test in T that detects m_i also detects m_j. If m_i dynamically subsumes m_j but the converse is not true, the subsumption is *strict*. If two mutants m_i and m_j in M are detected by exactly the same tests in T, m_i and m_j, the subsumption is not strict.

The *Dynamic Mutant Subsumption Graph* (DMSG) captures the subsumption relationship among mutants [31]. Each node in a DMSG represents a maximal set of redundant mutants and each edge represents the dynamic subsumption relationship between two sets of mutants. More specifically, if m_i strictly subsumes m_j, then there is an edge from the node containing m_i to the node containing m_j. If a test detects any arbitrary mutant in the DMSG, it is guaranteed to detect all the subsumed mutants [3], i.e., all mutants in the same node or below it in the graph.

Figure 1b shows an example detection matrix that indicates which test detects which mutants. In this example, the set M consists of 14 mutants and the set T consists of 4 tests. Every test that detects m_{12} also detects m_3, m_6, and m_{11}. Hence, m_{12} dynamically subsumes these mutants. In the case of the first two mutants, the dynamic subsumption is strict. However, m_{11} and m_{12} are detected by exactly the same tests, so the subsumption is not strict.

The DMSG shown in Figure 1c visualizes the subsumption relationships. Mutants m_1 and m_{10} are not detected by any of the tests in T—shown in the unconnected node with a dashed border. These mutants are equivalent with respect to T but they may be detectable by a test that is not an element of T. The DMSG is based on a finite test set and can only make claims about equivalence with respect to T.

Dominator mutants appear in the graph inside *dominator nodes*, stylized with double borders. Figure 1c has two dominator nodes; any combination of one mutant from each dominator node, such as $\{m_4, m_8\}$ or $\{m_9, m_{13}\}$, forms a dominator set. Figure 1c has $4 * 2 = 8$ distinct dominator sets. Ultimately, only two of the 14 mutants matter—if a test set detects the mutants in a dominator set, it is guaranteed to detect all non-equivalent mutants in the DMSG.

3 MUTATION ANALYSIS VS. MUTATION TESTING

This paper distinguishes between two mutation use cases: *mutation analysis*, which we define as a (research-based) use of mutation techniques to assess and compare the mutation adequacy of existing test sets, and *mutation testing*, which we define as a testing approach in which a developer interprets mutants as test goals and writes tests to satisfy those goals.

This distinction is blurred in the literature and, as a consequence, prior work usually applied the same methods for evaluating mutant selection techniques to both analysis and testing use cases. Specifically, evaluations usually compare mutant selection strategies to random mutant selection [1, 9, 16, 51, 54] by sampling a fraction (often less than 10%) of all mutants once, and then evaluating how effective a randomly sampled test set, which achieves full or x% mutation adequacy on the sample of mutants, is for the full set of mutants. In other words, these evaluations assess appropriate sampling thresholds and how representative the sampled set of mutants is of the entire set with respect to measuring mutation adequacy. Examples include E-selective [41, 42], N% random [1], and SDL [14, 49] approaches, all of which are evaluated with respect to mutation adequacy or, more recently, with respect to minimal mutation adequacy [33].

We argue that this evaluation methodology is appropriate for mutation analysis, but not mutation testing. Mutant selection approaches for mutation analysis and mutation testing differ both in their overall goals (selecting a representative subset of mutants to measure an existing test set's mutation adequacy vs. selecting a test goal that elicits an effective test) and presumed use cases (*a priori* vs. *incremental* mutant selection). This paper accounts for these differences in the design of a selection strategy for mutation testing and the methods used to evaluate its efficacy.

Goals The goal of mutation analysis is to assess and compare existing test sets or testing approaches by measuring mutation adequacy. In the context of mutation testing, however, achieving full mutation adequacy is neither realistic nor desirable [33, 45]. Developers do not write mutation- or even coverage-adequate test sets, and for good reasons [19, 34, 48]. The problems with achieving full coverage adequacy (e.g., test goals that are unsatisfiable or simply not worth satisfying) apply equally to mutation adequacy, with mutation adequacy having the additional burden of equivalent mutants, which pose an unrealistic workload [48]. Instead, industrial mutation testing systems present few mutants at a time and do not even compute mutation adequacy [7, 45, 46]. As a result, mutation testing requires selecting the most useful of all mutants according to some measure. (Section 4 proposes such a measure: TCAP.)

Evaluation Mutation analysis usually selects an entire subset of mutants just once, *a priori*, while mutation testing repeatedly and *incrementally* selects a few mutants. In the mutation testing use case, a developer is presented with one or a few mutants at a time, then resolves them by writing a test or labeling them as equivalent [7, 46]. If a mutant is presented as a test goal and resolved at a given time, then subsequent selections reflect the fact that the mutant, as well as all subsumed mutants, are detected.

A key difference between a-priori and incremental mutant selection is the effect of redundancy among mutants on the effort required to resolve those mutants. As an example, consider a-priori selecting three mutants with two selection strategies S_1 and S_2. Suppose the first mutant of S_1 elicits a test that detects all three mutants, whereas the first mutant of S_2 elicits a test that detects only that mutant, the second mutant is equivalent, and the third mutant elicits an additional test. The mutant sets of S_1 and S_2 contain the exact same number of mutants, but the effort to resolve those mutants differs—S_1 required resolving only one mutant whereas S_2 required resolving all three.

Prior evaluations of mutant selection techniques were largely based on a-priori mutant selection and considered two measures: (1) the number or ratio of selected mutants and (2) the mutation adequacy achieved by a corresponding test set. This is appropriate for the mutation analysis use case. For the mutation testing use case, however, an evaluation should be principally concerned with the effort required to resolve mutants and overall progress with respect to advancing test completeness. In other words, such an evaluation

should be agnostic to redundancy and measure test completeness over actual effort (as opposed to the number of selected mutants).

Our evaluation (Section 6) is aligned with the mutation testing use case and adopts the model proposed by Kurtz et al. [33]. Specifically, it operationalizes effort as a sequence of *work* steps wherein a developer is presented a single mutant and then either writes a test to detect it or labels it as equivalent. The amount of work to resolve each mutant is presumed to be equal, and the total amount of work is therefore the sum of the number of tests written and the number of equivalent mutants. This model has two advantages. First, it is agnostic to redundancy: after a test is introduced, all detected mutants are removed from further consideration and never subsequently selected. Second, equivalent mutants have a cost in that they consume work but do not advance test completeness[1].

4 TEST COMPLETENESS ADVANCEMENT PROBABILITY (TCAP)

We propose a new measure of mutant usefulness: *test completeness advancement probability* (TCAP), which enables prioritizing mutants for incremental selection. For a given mutant, TCAP is the probability that a mutant, if presented as a test goal, will elicit a test that advances test completeness. The probability distribution of TCAP is generally unknowable, but it is possible to estimate it with respect to an existing set of developer-written tests. For simplicity, TCAP refers to its estimate in the remainder of this paper.

Kurtz et al. [33] defined *test completeness* in terms of work: what fraction of the expected number of tests necessary for mutation adequacy have been written? Kurtz et al. [33] further showed that *dominator score*, which is the fraction of dominator nodes detected by a test set, enjoys a linear relationship with test completeness. This is in contrast to the traditional mutation adequacy score, which rises rapidly with the first few tests due to redundant mutants, and hence is a poor measure of test completeness. Consequently, this paper uses dominator score as a proxy for test completeness.

Unlike prior work which defines mutant usefulness as a property solely of the mutant itself (e.g., [3, 25, 29, 36]), TCAP quantifies the usefulness of a mutant in terms of the value of its detecting tests. This captures a key idea: a mutant is useful as a test goal only insofar as it elicits useful tests, and a useful test is one that advances test completeness. Notice four properties of TCAP:

(1) Equivalent mutants have a TCAP of 0. This is desirable and captures the fact that equivalent mutants do not lead to tests.
(2) Dominator mutants have a TCAP of 1. This is in line with the definition of test completeness: selecting a test for a dominator mutant is guaranteed to advance test completeness.
(3) Subsumed mutants have a TCAP of strictly greater than 0, but less than or equal to 1. Since subsumed mutants are always detected by at least one test that also detects a dominator mutant, their TCAP is strictly greater than 0. Note that TCAP can be 1 for subsumed mutants, which is desirable and captures the idea that a mutant's usefulness is defined by the tests that it elicits.
(4) The TCAP of a subsumed mutant is smaller than or equal to that of its dominating mutant(s). This is desirable because subsumed mutants can impose overhead if a dominating mutant is later presented as a test goal: a developer might write a weak test to detect a subsumed mutant and later write a stronger test to detect a dominating mutant; it would have been more efficient to simply write the stronger test.

Recall the motivating example in Figure 1: only the equivalent mutants m_1 and m_{10} reach the minimum TCAP of 0; all six dominator mutants have a TCAP of 1; the subsumed mutants m_2 and m_5 as well as mutants m_3 and m_6 have a TCAP of 0.5—half of the tests that detect these mutants also detect a dominator node. In contrast, the subsumed mutants m_{11} and m_{12} have a TCAP of 1, despite not being dominators—both tests that detect these mutants also detect a dominator mutant.

We do not value mutants according to their ability to detect known faults (fault coupling) for three reasons. First, any benchmark that provides a set of known real faults is inevitably a subset of the faults that could have been introduced or that already exist but are yet to be detected. As a result, biasing mutant selection to a limited set of mutants may cause parts of the programs under test to not be mutated and tested at all. Second, the core idea of mutation testing is to systematically mutate a program to guard against fault classes, not just a few known faults. Finally, the set of dominator mutants subsumes all fault-coupled mutants.

5 PREDICTING TCAP

To evaluate the benefits of using TCAP to prioritize mutants, we trained a set of machine learning models—linear models and random forests— that predict TCAP from mutants' static program context.

Note that the purpose of training these machine learning models is to evaluate whether program context is predictive of TCAP and whether prediction performance translates to downstream improvements in mutation testing when incrementally selecting mutants based on said predictions. The goal is neither to maximize any particular metric of model performance, nor to comprehensively explore the design space of machine learning models to identify the model design that would be best suited for this task. We conjecture that more sophisticated machine learning models and a richer feature set will likely yield larger improvements, but we leave an in-depth exploration of modeling choices and feature importance as future work.

5.1 Dataset

To produce a dataset, we generated mutants for subjects drawn from Defects4J [22] and transformed them into feature vectors describing the mutation and program context. Additionally, we associated each mutant with a label for TCAP, derived from *detection matrices*.

We chose 9 subjects (projects) from the Defects4J benchmark [22] (v2.0.0), which provides a set of 17 open-source projects accompanied by thorough test suites. We selected the latest version of each project and generated mutants for the entire project. Out of 17 projects, we filtered out 5 because of technical limitations in the mutation framework (e.g., JVM limits on the size of an individual method) and 3 for which the computation of a full detection matrix was computationally too expensive.

[1] This model considers equivalent mutants as useless because they do not lead to tests. This is a simplification: equivalent mutants can be useful and, for example, expose code defects or lead to code refactorings [45, 48].

Table 1: Summary of subject classes and covered mutants. *Covering* gives the mean number of tests covering each mutant, and *detecting* gives the mean number of tests detecting each mutant.

Project	Classes	Mutants				Tests		
		Total	Equivalent	Detectable				
				Dominator	*All*	*Total*	*Covering*	*Detecting*
Chart	446	65,300	47.5%	25.5%	52.5%	2,193	24.8	6.9
Cli	20	2,838	16.0%	25.1%	84.0%	355	39.9	18.3
Codec	50	24,281	30.5%	29.9%	69.5%	776	10.3	3.9
Collections	257	19,480	22.9%	29.0%	77.1%	16,069	26.9	14.5
Csv	9	1,986	17.5%	19.1%	82.5%	293	52.0	25.9
Gson	41	8,434	22.6%	24.0%	77.4%	1,035	98.2	41.0
JacksonXml	29	3,117	30.1%	26.1%	69.9%	183	39.2	15.7
JxPath	135	18,530	28.3%	20.1%	71.7%	386	76.6	28.0
Lang	97	38,532	17.8%	43.4%	82.2%	2,291	10.8	4.9
Overall	1,084	182,498	32.1%	29.6%	67.9%	23,581	29.6	11.1

We used the Major mutation framework [21] to generate all possible mutants for each of the 9 subjects and to compute the detection matrices. Of the 2,033,496 entries in the detection matrices, 3.2% were inconclusive due to a timeout of $16t_o + 1$ seconds, where t_o is a sample of the test runtime before mutation. Mutants that time out are considered detected.

We retained only mutants that are covered by at least one test for the evaluation and training sets. Covered but undetected mutants are deemed equivalent, which is a common approximation in mutation testing research. Since uncovered mutants cannot be detected but there are no tests that would increase confidence in the approximation of mutant equivalence, these are excluded. Table 1 provides a detailed summary of our final dataset, showing only retained (covered) mutants.

5.2 Program Context Features

Adopting the modeling approach and extending the model feature set of Just et al. [25], we modeled program context features using information available in a program's AST. Given a mutated AST node, we consider syntactic context (i.e., parent and child nodes in the AST as well as nesting information) and semantic context (i.e., the data type of expressions and operands or the data type of method parameters) of that node.

Note that prior work (e.g., [53]) used information derived from test executions (e.g., code coverage) when making predictions in a mutation analysis context. Because our goal is to identify useful mutants for tests that have not yet been written (mutation testing), we necessarily avoid features derived from tests.

Specifically, we chose the following set of features:

- *Mutation Operator.* The specific program transformation that generates a mutant (e.g., `<` ⟼ `!=` in Figure 1).
- *Mutation Operator Group.* One of AOR, COR, EVR, LOR, LVR, ORU, ROR, SOR, or STD.
- *Node Data Types.* The summary data type of the mutated node in the AST (e.g., `int`, `class`, or `boolean(int,int)`) encoding the return and parameter types of methods and operators, as well as a more abstract version which maps complex types to their result types (e.g., `boolean(int,int)` becomes `boolean`).
- *Node Kind.* The kind of the mutated AST node (e.g., 'binary operator', 'literal', 'method call').
- *AST Context.* Four features, each of which is a categorical variable: (1) the sequence of AST node kinds from the mutated node (inclusive) to the root node of the AST; (2) extends the first feature by including edge annotations describing node relationships; e.g., a `for` loop has child nodes `for:init`, `for:cond`, `for:inc`, or `for:body`; (3) and (4) correspond to the first two features, but provide a higher level of abstraction and include only those nodes that are top-level statements (as opposed to individual expressions).
- *Parent Context.* Versions of *AST context* features that consider only the immediate parent of the mutated node.
- *Children Context.* Three features that indicate the node kinds of the immediate child nodes of the mutated AST node: (1) an immediate child node is a *literal*; (2) an immediate child node is a *variable*; (3) an immediate child node is an *operator*.
- *Relative Position.* The relative line number of the mutated node inside its enclosing method, divided by the total number of lines in that method.
- *Nesting.* Seven features in total: (a) the maximum block nesting depth anywhere in the enclosing method; (b) the nesting depth (number of enclosing blocks) considering only loops, only conditionals, and any enclosing block (i.e., the sum of the previous two); and (c) an additional three features dividing the previous three by the maximum nesting depth.

5.3 Machine Learning Models

We evaluated a small number of model design choices and training settings against an intrinsic measure of model performance with the ultimate goal of choosing a single model for downstream evaluation. Each training setting was a kind of hold-one-out train-test split, using each held out project (or class in a given project) at a time as the evaluation set. We trained all models using Scikit-Learn [44] and evaluated every combination of the following choices:

(1) *Model Choice.* We compared ridge regression ($\alpha = 1.0$) to a random forest regression (max depth of 3 and 10 trees).
(2) *Feature Set.* We used a comprehensive set of features (*All* features, enumerated in Section 5.2) and a subset of these features (*Few* features, used in [25]: only the mutation operator and detailed parent statement context).
(3) *Training Setting.* We evaluated the following train-test splits:
    (a) *All Projects*: training on mutants from all projects, including those in non-held-out classes in the same project, and testing on mutants in the held-out class;
    (b) *Between Projects*: training on mutants strictly in other projects and testing on mutants in the held-out project;
    (c) *Project-Only*: training on mutants in all but one held-out class in a single project and testing on mutants in the held-out class.

    These training-test splits correspond to three realistic mutation testing settings, in which a developer intends to test a new class in an existing project, with or without access to other projects, or a completely new project.

We evaluated each combination of model choice, feature set, and training setting by training one model for each possible training-test split. For example, for a given model choice and feature set, the *All Projects* training-test splits resulted in a total of 1,084 models, one for each held-out class in the dataset, whereas the *Between Projects* splits resulted in just 9 models, one for each held-out project.

We compared the combinations based on the Spearman rank correlation coefficient between the TCAP predictions and labels. The rank correlation coefficient is an appropriate intrinsic measure because we are principally interested in the order in which mutants will be selected. Figure 2 shows the distributions of the resulting coefficients across all models. We find that:

(1) *Model Choice.* A linear model is sufficient for the feature sets considered, an observation consistent with prior work [30].
(2) *Feature Set.* A larger feature set modestly improves the performance of linear models, but it reduces variation in correlation coefficients between splits, perhaps by inducing models that are less sensitive to noise in individual features. Many of these features are collinear (e.g., the various Nesting features). While this does not affect the suitability of our model designs to the downstream evaluation, collinearity means that linear model coefficients and permutation feature importances are not individually interpretable and the high cardinality of our categorical features mean the same for random forest impurities. Still, an ad-hoc analysis suggests that syntactic context features are the most important ones.
(3) *Training Setting.* The median performance of *Project-Only* is consistently the best, especially for the non-linear models, though the difference is less pronounced for linear models.

While many of the model configurations perform similarly, we choose the following configuration for the experiments in Section 6:

Model=Linear, Features=All, Training Setting=Project-Only

Project-Only is a common evaluation scenario and the chosen combination has the highest median correlation coefficient.

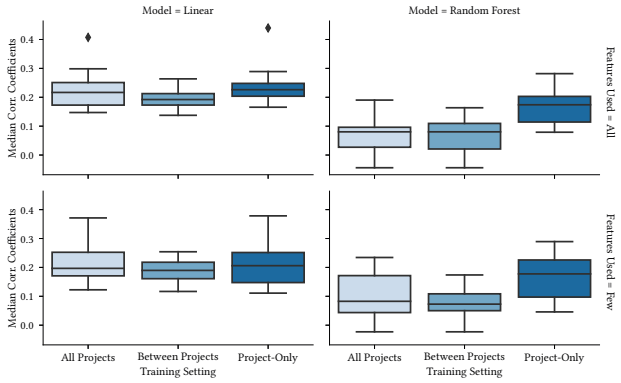


Figure 2: Intrinsic evaluation of model performance, broken down by model choice, feature set, and training setting. Each boxplot contains nine data points, showing the distribution of Spearman's ρ for all projects. Whiskers extend up to 1.5 times the interquartile range.

6 EVALUATION

Consider developing code for a large project for which a significant body of code, along with associated mutants and tests, already exists. A realistic mutation testing scenario employing a TCAP-based mutant selection approach is:

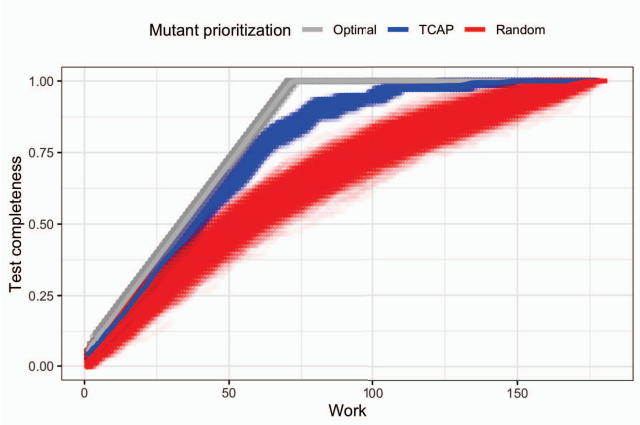
(1) Train a model that predicts TCAP on a project's existing mutants and tests.
(2) Generate new mutants for a developer's current source file.
(3) Predict TCAP of these new mutants, using the trained model.
(4) Provide the developer with the highest TCAP mutant as a test goal; if they write a test, then remove all mutants detected by that test from further consideration. Repeat until a stopping condition is met (e.g., some fixed amount of work or a predicted TCAP threshold).

As motivated in Section 3, this scenario corresponds to mutation testing, using TCAP to prioritize mutants for incremental mutant selection, and each iteration corresponds to a single work unit.

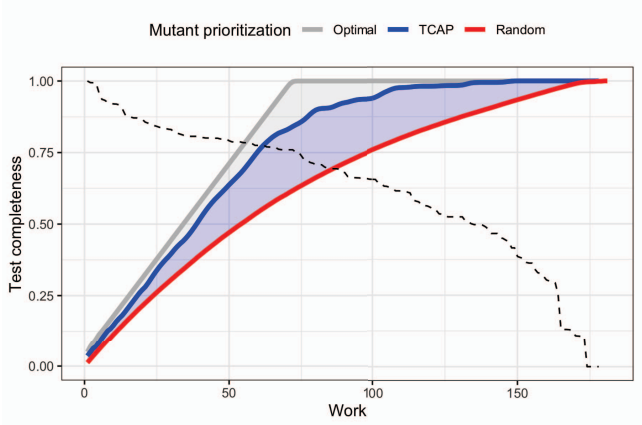
Work Simulation To evaluate TCAP-based mutant prioritization, we simulated the aforementioned scenario. A *work simulation* begins with some initial—possibly empty—test set and the set of mutants not detected by those tests, selects the highest-scoring mutant according to some prioritization strategy, adds a randomly selected test (without replacement) that detects that mutant to the test set, and repeats with the remaining set of undetected mutants.

We evaluated TCAP-based mutant prioritization against two baselines: *Optimal* and *Random*. Optimal prioritizes mutants based on the TCAP labels in the dataset, and Random simply produces a randomized order. *Optimal* establishes an upper bound on the performance of any mutant prioritization strategy and *Random*, perhaps surprisingly, corresponds to the state-of-the-art [18, 33].

All mutant prioritization strategies in our evaluation are stochastic: ties for TCAP are broken randomly, and the test introduced at each work step is chosen uniformly at random (without replacement) from those that detect the selected mutant. Our evaluation repeats each work simulation 1,000 times per class, for all strategies.



(a) A work simulation for a single class, plotting all individual 1,000 runs for each strategy.



(b) The mean test completeness (over the 1,000 individual runs) per unit of work. The dashed line indicates the predicted TCAP.

Figure 3: An example work simulation for the CollectionUtils class in Apache Collections.

Figure 3 shows an example work simulation for a single class, illustrating how test completeness increases over the course of the simulation. The optimal strategy increases smoothly and rapidly to a test completeness of 100% since each unit of work detects one—or occasionally more than one—dominator node. TCAP-based selection does not increase as rapidly, despite making constant progress, since the choices are imperfect. However, it increases substantially faster than the random strategy. Consider the units of work required by each strategy to reach 0.75 test completeness. The optimal strategy requires about 50 units of work, and TCAP-based selection requires no more than 60. The random selection strategy, in comparison, requires about 100 units of work. In this simulation, a developer interested in reaching a test completeness of 0.75 needed to resolve about 20% more mutants with the TCAP-based selection, compared to the optimal strategy, whereas randomly selecting mutants as test goals would have required resolving 100% more mutants.

Efficiency In order to summarize a work simulation and quantify the efficacy of a prioritization strategy, we define *efficiency* to be a measurement of a strategy's improvement over prioritizing mutants randomly, normalized by the performance of the optimal strategy. Specifically, the efficiency of the TCAP-based strategy t over a sequence of work units u is:

$$\sum_u (c_t^u - c_r^u) / \sum_u (c_o^u - c_r^u) \quad (1)$$

where c_o^u, c_r^u, and c_t^u are test completeness at work unit u for the optimal, random, and TCAP strategies respectively. An optimal strategy has an efficiency of 1, a strategy no better than random has an efficiency of 0, and a strategy that performs worse than random has a negative efficiency. Intuitively, a positive efficiency indicates how much of the gap (*wasted work*) between random and optimal the strategy closes. In contrast, a negative efficiency indicates an underperforming strategy, but it is not normalized because we do not include the worst possible strategy. The subsequent sections show that TCAP-based prioritization is more efficient than random, and they also investigate outliers of negative efficiency results.

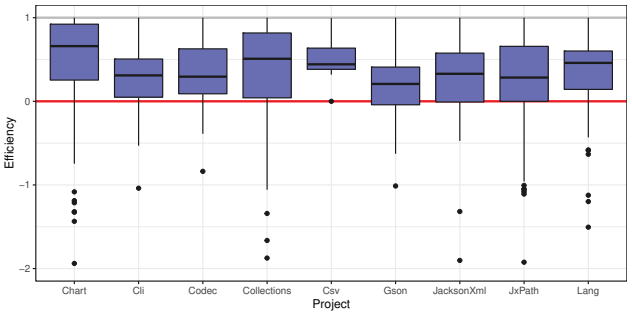


Figure 4: Efficiency per class, grouped by project (7 extreme outliers with efficiency < -2 are removed from the plot for clarity; see Section 6.1.2 for details.)

Note that 46 out of 1,084 classes had either exactly one generated mutant or very few, yet only equivalent, mutants. In either case, all three prioritization strategies are trivially equivalent, and hence we discarded these classes, leaving 1,038 classes for analysis.

6.1 Simulating Work

6.1.1 Measuring Average Efficiency. Our goal is to evaluate whether TCAP-based mutant prioritization is likely to save work for a developer testing an arbitrary class with an arbitrary test budget. To that end, we simulated the efficiency of TCAP-based prioritization for all 1,038 retained classes in our data set. We derived TCAP predictions from the model described in Section 5.

Figure 4 shows the distribution of efficiency per class, broken down by project. The results show that TCAP-based prioritization consistently and meaningfully outperforms the random strategy across all projects, reducing median wasted work by between a third and a half.

6.1.2 Inspecting Extreme Outliers. We encountered seven extreme outliers, with efficiencies ranging from -2.04 to -6.49, which are

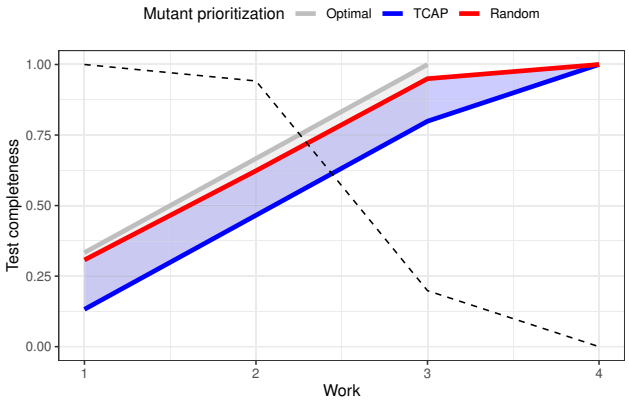


Figure 5: Work simulation for an extreme outlier class (ConstantFactory), with a negative efficiency of -4.04.

Table 2: Efficiencies calculated as the sum of every work unit across all classes, per project. *Total* refers to all classes across all projects. These efficiencies account for class size and corresponding total number of work units. This is in contrast to Figure 4, which plots distributions over class-level efficiencies irrespective of the number of work units required to achieve 100% test completeness.

Chart	Cli	Codec	Collections	Csv	Gson	JacksonXml	JxPath	Lang	**Total**
0.41	0.19	0.19	0.35	0.45	0.15	0.22	0.11	0.32	**0.33**

removed from Figure 4 for clarity. The asymmetry in the efficiency measure motivates us to understand whether these extreme outliers indicate a potential for meaningful costs in practice over the random approach. One possible explanation for these values lies in features that can make a mutant set well tailored to the random approach. For example, all extreme outliers have five or fewer dominator nodes, with four outliers only having one. The percentage of covered mutants that are dominators is over 30% for all outliers. Consequently, when most mutant choices increase completion and problem size is small, an early suboptimal choice made by TCAP-based selection can result in a substantial negative efficiency score since random exhibits steady progress. Figure 5 demonstrates this by showing an example for an extreme outlier (ConstantFactory from Apache Collections) with a computed efficiency of -4.04.

To introduce some perspective, we can quantify the potential real-world cost of these outliers for a developer generating the handful of tests needed to reach full test completeness on classes of this size. For 5 out of 7 outliers, the median time to completion for TCAP-based selection is only 1-2 steps behind random (4-5 steps for the remaining 2 outliers). However, this measure ignores the high variance present in the random approach, compared to TCAP-based selection. When considering the mean completion percentage, TCAP-based selection finishes with or before random for 6 outliers, with the final outlier taking only one additional step. In summary, none of these results suggest the potential for significant performance concerns arising from these outlier values.

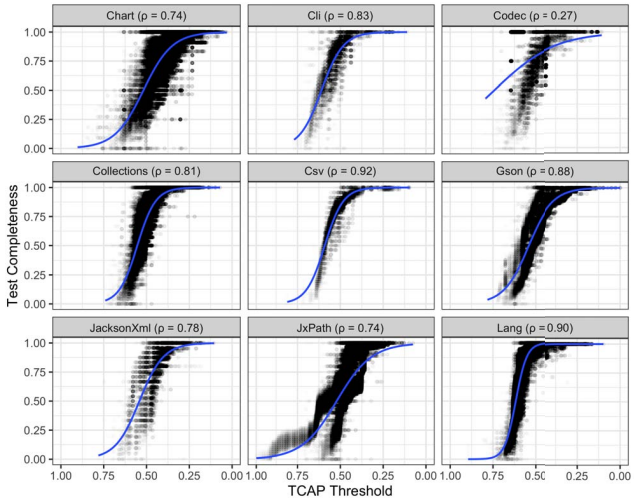


Figure 6: TCAP threshold as a predictor of test completeness (of the tests elicited by all mutants whose predicted TCAP is above that threshold). Note the inverted TCAP Threshold axis. All Spearman's rank correletion coefficients (ρ) are significant at $p < 0.01$.

6.1.3 Accounting for Class Size. A potential weakness of the efficiency analysis in Section 6.1.1 is that it treats all classes as being equal, even though they differ in size and therefore perhaps in importance or testing difficulty. To account for this, we computed the overall efficiency per project and across all projects—that is, the efficiency over all mutants per project and over all mutants in the entire data set. This means that larger classes, with more mutants and work steps, have a higher weight. Recall Figure 3b, which visualized efficiency as the ratio of two areas. Intuitively, overall efficiency is the ratio of the sums of these two areas across all classes, as opposed to the average ratio per class. Table 2 shows the results. While the efficiency varies between projects, it is consistently positive. Furthermore, the variation in overall efficiency aligns with the variation of median efficiency across projects. In conclusion, TCAP-based mutation testing reduces the total amount of wasted work by a third.

6.2 Deciding When to Stop Testing

Selecting mutants in order of descending predicted TCAP allows a developer to focus time and effort on the most important test goals. However, the ranking alone does not answer the fundamental question of when resolving additional mutants provides only marginal benefits. In other words, when should a developer stop testing and how well tested is a software system after writing tests for all mutants with a TCAP above a given threshold.

In order to understand whether TCAP thresholds are predictive of expected test completeness, we correlated the two for all classes in our data set. Figure 6 shows the results, indicating a strong association between TCAP thresholds and test completeness. At the same time, the variance of TCAP thresholds is quite high for some projects. For example, a developer interested in reaching at least 50%

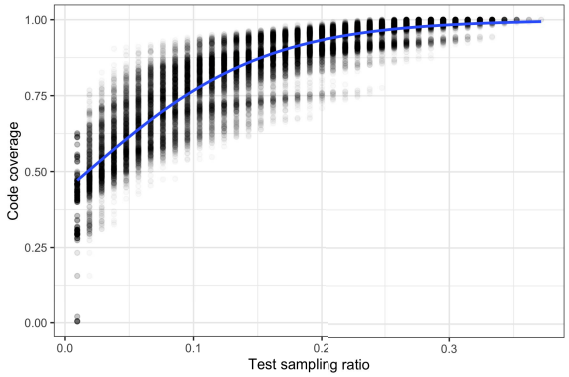


Figure 7: Test sampling ratio vs. line coverage.

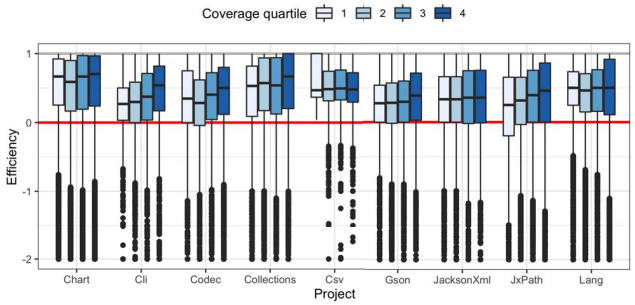


Figure 8: Efficiencies per class, broken down by project and coverage quartile.

test completeness for an arbitrary class in Csv could confidently use a TCAP of about 0.6 as a stopping criterion. However, it's less clear what TCAP threshold a developer of Chart should use; a substantial amount of TCAP density spans the range from 0.6 to 0.3. Nonetheless, these results are promising and suggest that accounting for project-specific characteristics may reduce variance and improve these predictive models. We leave a deeper exploration as future work.

6.3 Simulating Work with Initial Test Sets

Our evaluation so far has focused on the efficiency of TCAP-based mutant prioritization in a scenario where mutation testing is applied from scratch. However, mutation testing often happens after some tests have already been written. To assess whether the efficiency of TCAP-based mutant prioritization is sensitive to the efficacy of an existing test set, we performed an additional simulation. Specifically, we ran an additional 1,000 work simulations for each class, with a randomly selected, line-coverage-guided initial test set selected from that class' existing test set. For the purpose of this simulation, we measure test set efficacy as the line-coverage ratio.

Our goal is to understand the relationship between an initial test set's code coverage and the efficiency of TCAP-based mutant selection for all remaining mutants not detected by that initial test set. Consequently, we aim to generate, for each class, a collection of initial test sets with a uniform distribution of code-coverage ratios. As noted by Chen et al. [10], the relationship between test set size and code coverage is not linear. A uniform random selection over test set size would drastically oversample test sets with very high code-coverage ratios. Figure 7 visualizes this problem. Given the log-linear relationship, we resorted to inverse transformation sampling: (1) we fitted a log-linear model for each class, predicting the log of the test sampling ratio from a given code-coverage ratio; (2) we sampled 1,000 code-coverage ratios uniformly at random; (3) we computed the corresponding test sampling ratio by sampling from the inverse of the fitted model. The outcome of this sampling approach was a pool of 1,000 test sets for each class, whose code-coverage distribution was approximately uniform. Performing a work simulation for each of these initial test sets yielded efficiency values that correspond to a given code-coverage ratio.

For simplicity, Figure 8 shows these efficiency results, binned by quartiles of the code-coverage distribution. A regression analysis, using the non-binned data, confirmed the visual trend: the degree of code-coverage, achieved by the initial test suite, is either uncorrelated with efficiency or it is weakly positively correlated. The key takeaway is that the efficiency of TCAP-based mutant prioritization is agnostic to the efficacy of the existing test set.

7 THREATS TO VALIDITY

External Validity Our results are potentially limited to the projects in our empirical study, all of which are Java programs. While we chose a variety of different projects, our results do not necessarily apply to other languages or even to other Java projects with different characteristics. Additionally, our estimates of TCAP rely on existing, developer-written tests, which were likely not developed in a mutation testing setting. It is possible that these tests are not representative of those that developers would write when resolving a mutant. However, a study of Petrović et al. [46], in an industrial setting, found no qualitative differences between tests written specifically for mutants vs. tests written for other objectives.

Construct Validity As is common in mutation testing research, we approximate equivalent mutants and dominator mutants with a set of existing tests. Since test sets generally have to be quite thorough before the dynamic approximation of the subsumption relations converges to the true subsumption relation [31], we risk identifying some non-dominator mutants as dominators, and vice-versa. To counter this threat, we relied on projects that come with thorough test sets. Additionally, as described in Section 3, we assume that dominator score is a valid proxy for test completeness (w.r.t. mutation testing). While any choice of proxy poses a threat to construct validity, dominator score is a better proxy than the alternative, the mutation score.

Internal Validity We rely on an idealized model of mutation testing. For example, we assume that work units correspond to some constant amount of actual engineering effort, but there is some variance around the time required to resolve a single mutant [48]. While we have no reason to believe this is the case, it is possible that our evaluated model is biased toward mutants which systematically take either more or less actual effort to resolve.

8 RELATED WORK

The large number of generated mutants has long been a recognized problem in mutation analysis and mutation testing research. This section first discusses the most closely related work by Just et al. [25] and Zhang et al. [53], and then provides a more general overview of related work.

Closely Related Work Just et al. [25] demonstrated that program context, derived from the abstract syntax tree, can significantly improve estimates of a mutant's expected usefulness. In this context, usefulness was formally quantified as *mutant utility* along three dimensions: equivalence, triviality, and dominance. In short, Just et al. showed that mutant usefulness is context-dependent: a mutation that leads to a useful mutant in one context does not necessarily do so in another. While Just et al. showed that program context correlates with mutant utility, they did not make use of that correlation. They did not build a predictive model and did not evaluate the benefits of using such a model for downstream applications such as mutation testing.

Zhang et al. [53] used machine learning to predict mutation scores both within and across projects for a fixed test suite, using features derived both from program context and, critically, runtime information such as code coverage which is only available after tests have been written. Our predictions are very different: rather than predicting expected mutation scores based on existing test suites, we predict which mutants are useful and likely lead to additional, effective test cases. This distinction goes to the heart of the differences between mutation analysis and mutation testing.

Mutation Testing in Practice Petrović et al. reported on a large-scale deployment of mutation testing at Google [45–48]. Their mutation testing system was integrated into a commit-oriented development workflow. To combat the problems of generating too many mutants, lines of code not changed by a commit were ignored, as were lines in the commit not covered by at least one existing test. For each remaining line of changed code, the system generated at most one mutant. Our work is immediately applicable to this use case. TCAP-based mutant selection allows for picking the most useful mutant for any given line.

Beller et al. [7] report on a similar large-scale deployment at Facebook. They integrated a mutation testing system into a commit-oriented development workflow and evaluated the system's acceptance by working engineers. They innovated by semi-automatically learning a small number of context-sensitive mutation operators from real faults, reducing the total number of mutants generated.

Useful Mutants Researchers have addressed the notion that some mutants are more useful than others: *disjoint* mutants (Kintis et al. [29]), *stubborn* mutants (Yao et al. [52]), *difficult-to-detect* mutants (Namin et al. [36]), *minimal* (aka *dominator*) mutants (Ammann et al. [3] and Kurtz et al. [31]) and *surface* mutants (Gopinath et al. [17]). Disjoint, stubborn, difficult-to-detect, dominator, and surface mutants are suitable in a research context, but are not directly applicable to the engineer in practice. Namin et al. [36] described *MuRanker*, a tool to identify difficult-to-detect mutants based on the syntactic distance between the mutant and the original artifact. They postulated the existence of "hard mutants" that are difficult to detect and for which a detecting test may also detect a number of other mutants. This is closely related to what Kurtz et al. have formalized as *dominator mutants* [3, 31]. The key idea that distinguishes TCAP from other proxy measures for mutant usefulness is that a mutant is useful as a test goal in mutation testing only insofar as it elicits a test that advances test completeness.

Mutant Selection Prior work on mutant selection has focused on using a subset of the mutation operators (Mathur [35], Offutt et al. [41, 42], Wong et al. [50, 51], Barbosa et al. [6], Namin et al. [37–39], Untch [49], Deng et al. [14], Delamaro et al. [13], and Delamaro et al. [12]) and choosing a random subset of mutants (Acree [1], Budd [9], and Wong and Mathur [51]). Comparisons of the two approaches (Zhang et al. [54], Gopinath et al. [15, 16], and Kurtz et al. [33]) ultimately showed the counterintuitive result that existing mutant selection approaches do not outperform random selection. Just et al.'s results [25] are consistent with these findings: they showed that context-agnostic mutant selection shows no appreciable improvement over random selection. However, their results also demonstrated that program context is predictive of mutant usefulness, which motivated our work.

Fault Coupling While the numbers of mutants generated by mutation operators are already large, even more mutation operators are needed to generate mutants that are coupled to real faults: empirical studies of mutation adequacy (Daran and Thévenod-Fosse [11], Namin and Kakarla [40], Andrews et al. [4, 5] and Just et al. [23]) showed that fault coupling is high, but also that there is room for improvement. The tailored mutant work of Allamanis et al. [2] and the wild-caught mutant work of Brown et al. [8] demonstrated mutation approaches that can close this gap, but at the cost of substantially increasing the number of generated mutants. Context-based mutant selection has the potential to make these high-coupling approaches practical. Papadakis et al. [43] investigated the relationship between various quality indicators (measures of usefulness) for mutants; of particular relevance here is that, for the faults in their study, only 17% of dominator mutants were fault-revealing (a property closely related to fault coupling). As discussed in Section 4, we did not include a dimension for fault-coupling in our model precisely to avoid the consequent blind spots. The results of Papadakis et al. suggested that these blind spots may be quite large.

Equivalent and Redundant Mutants Jia et al. surveyed mutation testing in general and provided a detailed review of mutation equivalence detection techniques [20]. Reducing the number of equivalent mutants presented as test goals is a key goal in making mutation testing practical, and hence a key focus of our work. Our work can be applied in addition to existing, context-agnostic techniques: prioritizing mutants with respect to TCAP allows an incremental mutant selection approach to avoid mutants that are likely equivalent in a particular context. Researchers have also considered redundancy of mutants with respect to "weak" mutation: Kaminski et al. [27, 28] for relational operator replacement (ROR), Just et al. [24] for conditional operator replacement (COR), Yao et al. [52] for the arithmetic operator replacement (AOR), and Just and Schweiggert [26] over COR, UOI, and ROR. Weak redundancy analysis techniques are sound, but only apply to a small subset of the mutation operators, do not consider propagation, and do not address the equivalent mutant problem.

9 CONCLUSIONS

To make mutation testing feasible for practitioners, it is necessary to address the vast number of mutants produced by current mutation systems and the fact that most of these mutants are not useful.

This paper introduces a new measure for mutant usefulness, called test completeness advancement probability (TCAP), built on the insight that a mutant is useful as a test goal in mutation testing only insofar as it elicits a useful test—one that advances test completeness. Furthermore, this paper demonstrates that a mutant's static program context is predictive of TCAP and that prioritizing mutants with TCAP can effectively guides mutation testing.

The main results of this paper are as follows:

(1) Program context, modeled as syntactic and semantic features of a program's abstract syntax tree, is predictive of TCAP.
(2) TCAP-based mutant prioritization, independently of initial test set code coverage ratios, improves test completeness far more rapidly than random prioritization—which perhaps surprisingly had prevailed as the state-of-the-art despite decades of research on selective mutation.
(3) Predicted TCAP and achieved test completeness are strongly correlated. While predicted TCAP shows high variance for some subjects, the results suggest that an improved prediction model could render predicted TCAP as a practical stopping criterion.

DATA & SOFTWARE AVAILABILITY

To aid reuse and replication, we provide the data and source code underlying this work at: https://doi.org/10.6084/m9.figshare.19074428.

ACKNOWLEDGMENTS

We thank Audrey Seo and Benjamin Kushigian for development support, and the anonymous reviewers for their valuable feedback. This work is supported in part by National Science Foundation grant CCF-1942055.

REFERENCES

[1] Alan T. Acree. 1980. *On Mutation.* Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA.
[2] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles Sutton. 2016. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516* (2016).
[3] Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. Cleveland, Ohio, USA, 21–31.
[4] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering, (ICSE 2005)*. St. Louis, MO, 402–411.
[5] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (August 2006), 608.
[6] Ellen Francine Barbosa, José C. Maldonado, and Auri Marcelo Rizzo Vincenzi. 2001. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification, and Reliability, Wiley* 11, 2 (June 2001), 113–136.
[7] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036
[8] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-caught Mutants. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. 511–522.
[9] Tim A. Budd. 1980. *Mutation Analysis of Program Test Data.* Ph.D. Dissertation. Yale University, New Haven, Connecticut, USA.
[10] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.
[11] Murial Daran and Pascale Thévenod-Fosse. 1996. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. *ACM SIGSOFT Software Engineering Notes* 21, 3 (May 1996), 158–177.
[12] Marcio E. Delamaro, Lin Deng, Serapilha Dureli, Nan Li, and Jeff Offutt. 2014. Experimental Evaluation of SDL and One-Op Mutation for C. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. Cleveland, Ohio.
[13] Márcio E. Delamaro, Lin Deng, Nan Li, and Vinicius H. S. Durelli. 2014. Growing a Reduced Set of Mutation Operators. In *Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES)*. Maceió, Alagoas, Brazil, 81–90.
[14] Lin Deng, Jeff Offutt, and Nan Li. 2013. Empirical Evaluation of the Statement Deletion Mutation Operator. In *6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*. Luxembourg, 80–93.
[15] Rahul Gopinath, Iftekhar Ahmed, Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation Reduction Strategies Considered Harmful. *IEEE Transactions on Reliability* 66, 3 (Sept 2017), 854–874.
[16] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. *Do Mutation Reduction Strategies Matter?* Technical Report. School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, Oregon, USA.
[17] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. Measuring Effectiveness of Mutant Sets. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 132–141.
[18] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the Limits of Mutation Reduction Strategies. In *Proceedings of the International Conference on Software Engineering (ICSE)* (Austin, Texas). 511–522.
[19] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 955–963.
[20] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering* 37, 5 (September 2011), 649–678.
[21] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 433–436.
[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 437–440.
[23] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 654–665.
[24] René Just, Gregory M Kapfhammer, and Franz Schweiggert. 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 11–20.
[25] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 284–294.
[26] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification, and Reliability, Wiley* 25, 5-7 (2015), 490–507.
[27] Garrett Kaminski, Paul Ammann, and Jeff Offutt. 2011. Better Predicate Testing. In *Sixth Workshop on Automation of Software Test (AST 2011)*. Honolulu HI, 57–63.
[28] Garrett Kaminski, Paul Ammann, and Jeff Offutt. 2013. Improving Logic-Based Testing. *Journal of Systems and Software, Elsevier* 86 (August 2013), 2002–2012. Issue 8.
[29] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *17th Asia Pacific Software Engineering Conference (APSEC2010)*. Sydney, Australia.
[30] Bob Kurtz. 2018. *Improving Mutation Testing with Dominator Mutants.* Ph.D. Dissertation. George Mason University, Fairfax, VA.
[31] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutation Subsumption Graphs. In *Tenth IEEE Workshop on Mutation Analysis (Mutation 2014)*. Cleveland, Ohio, USA, 176–185.
[32] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *Twelfth IEEE Workshop on Mutation Analysis (Mutation 2016)*. Chicago, Illinois, USA.

[33] Robert Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants. In *FSE 2016, Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. Seattle, Washington, 571–582.
[34] Brian Marick. 1999. How to misuse code coverage. In *Proc. of ICTCS*.
[35] Aditya Mathur. 1991. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*. Tokyo, Japan, 604–605.
[36] Akbar Namin, Xiaozhen Xue, Omar Rosas, and Pankaj Sharma. 2015. MuRanker: A mutant ranking tool. *Software Testing, Verification, and Reliability* 25, 5-7 (August 2015), 572–604.
[37] Akbar Siami Namin and James H. Andrews. 2006. Finding Sufficient Mutation Operators via Variable Reduction. In *Second Workshop on Mutation Analysis (Mutation 2006)*. Raleigh, NC.
[38] Akbar Siami Namin and James H. Andrews. 2007. On Sufficiency of Mutants. In *Proceedings of the 29th International Conference on Software Engineering, Doctoral Symposium*. ACM, Minneapolis, MN, 73–74.
[39] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, Leipzig, Germany, 351–360. https://doi.org/10.1145/1368088.1368136
[40] Akbar Siami Namin and Sahitya Kakarla. 2011. The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada). ACM, New York, NY, 342–352. https://doi.org/10.1145/2001420.2001461
[41] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering Methodology* 5, 2 (April 1996), 99–118.
[42] Jeff Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Baltimore, MD, 100–107.
[43] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant Quality Indicators. In *Thirteenth IEEE Workshop on Mutation Analysis (Mutation 2018)*. Vasteras, Sweden, 32–39.
[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.
[45] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)*.
[46] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[47] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. 2021. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* (2021).
[48] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *Thirteenth IEEE Workshop on Mutation Analysis (Mutation 2018)*. Vasteras, Sweden, 47–53.
[49] Roland Untch. 2009. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In *ACM Southeast Regional Conference*. Clemson, SC, 19–21.
[50] W. Eric Wong, Márcio E. Delamaro, José C. Maldonado, and Aditya P. Mathur. 1994. Constrained Mutation in C Programs. In *Proceedings of the 8th Brazilian Symposium on Software Engineering*. Curitiba, Brazil, 439–452.
[51] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software, Elsevier* 31, 3 (December 1995), 185–196.
[52] Xiangjuan Yao, Mark Harman, and Yue Jia. 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)*. IEEE Computer Society Press, Hyderabad, India.
[53] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive Mutation Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 342–353.
[54] Lu Zhang, Shan-San Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Cape Town, South Africa, 435–444.
[55] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997), 366–427.