Data-Driven Test Case Generation for Automated Programming Assessment

Terry Tang
Rice University
terry.tang@rice.edu

Rebecca Smith
Rice University
rjs@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

Joe Warren
Rice University
jwarren@rice.edu

ABSTRACT

Building high-quality test cases for programming problems is an important part of any well-built Automated Programming Assessment System. Traditionally, test cases are created by human experts or using machine auto-generation methods based on the problem definition and sample solutions. Unfortunately, the human approach can not anticipate the numerous ways that programmers can construct erroneous solutions. The machine auto-generation methods are complex, problem-specific, and time-consuming.

This paper proposes a fast, simple method for generating high-quality test sets for a programming problem from an existing collection of student solutions for that problem. This paper demonstrates the effectiveness of the proposed method in online programming course assessments. The experiments showed that, when applied to large collections of such programs, the method produces concise, human-understandable test sets that provide better coverage than test sets built by experts with rich teaching experience.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Reliability, Verification

Keywords

Automated Programming Assessment System; Automatic Test Case Generation; Data-Driven; MOOC

1. INTRODUCTION

Automated Programming Assessment (APA) systems have been widely used in many areas to evaluate program correctness and efficiency. Universities and companies have built automated systems for judging programming competitions, such as UVa [7], PC^2 [3], TopCoder [6], and Google Code Jam [1]. Educators have created Leetcode [2], Rosalind [5], and Project Euler [4] to teach programming skills and algorithms through problem-solving exercises. Recently, many massive open online courses (MOOCs) have relied heavily on automated assessment systems for grading programming assignments [18].

The effectiveness and efficiency of an APA system is largely determined by the quality of its test cases. Manually building test cases is tedious and time-consuming, and makes it difficult to guarantee a high level of error detection. Thus, researchers have put a great deal of effort into test case auto-generation to reduce human effort and increase test quality [10, 16, 21].

Traditional software testing systems target a single existing piece of software, and are typically used to validate software that is already believed to be largely correct. In contrast, APA systems target a broad spectrum of programs implementing a single specification, and are typically used as a step in the incremental feedback-debugging cycle. As a result, they face additional requirements. First, they demand fast execution to provide a rapid feedback cycle. Therefore, they typically employ test suite reduction strategies [11] to minimize the number of tests while retaining quality. Second, they should be capable of assessing yet-to-come implementations based on a problem description and a few correct sample solutions. Third, APA systems often benefit from presenting test cases in a logical order, *e.g.*, increasing complexity, particularly in the context of offering feedback and partial credit to students in programming courses.

Shayma *et al.* [16], categorized traditional test case auto-generation techniques into random-based, search-based, and data mining-based methods. Random-based methods randomly generate a large number of tests within a constrained search space [8]. Given an existing piece of code, techniques like mutation-driven selection [12] can be applied in random-based methods to eliminate unimportant test cases. However, future submissions are hard to predict, and thus cannot guide test selection in this way.

Search-based methods [15] use more advanced algorithms such as genetic algorithms [14] and particle swarm optimization [18] to directly search for high quality test cases. However, these methods are complex and computation intensive. Moreover, they are not easily generalizable, requiring problem-specific fitness function selection and tuning.

ITiCSE '16, July 09-13, 2016, Arequipa, Peru

DOI: http://dx.doi.org/10.1145/2899415.2899423

Data mining-based approaches have been proposed to reduce the number of test cases without losing coverage [17, 20, 22] by identifying hidden input-output (I/O) relations. However, these methods are typically more complex, requiring a large number of training samples and significant tuning in order to achieve accurate predictions.

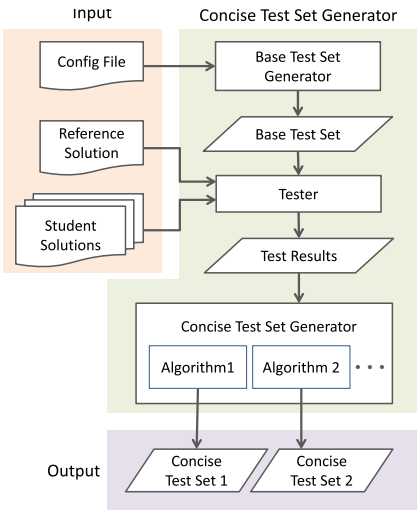

Figure 1: FEAT Structure and Workflow

This paper describes a simple and easy-to-use toolchain for creating high-quality test sets for automated assessment systems, Feedback and Evaluation via Auto-generated Tests (FEAT). It utilizes a collection of previously-submitted student solutions to guide the test generation process, under the assumption that no error is unique in a crowd-sourcing scenario [13].

FEAT consists of three modules (Figure 1): the base test set generator, the tester, and the concise test set generator. The first module auto-generates a large pool of test cases for a particular programming assignment, based on an instructor-provided problem specification. This pool is referred to as the *base test set*. The second module uses the base test set to check the correctness of a training set of student solutions against an instructor-provided reference solution. The third module uses the testing results to construct *concise test sets* that are small subsets of the base test set. These concise test sets are carefully chosen such that they detect every erroneous solution that could be found using the much larger base test set, while providing the efficiency needed for use in an APA system.

The effectiveness of FEAT was tested in a series of assignments from massive open online courses [19]. The test sets for these assignments were manually built by the authors in the first course session, but were reported as incomplete by students. Evaluation demonstrated that FEAT generates superior test sets, providing students with higher-quality feedback while freeing instructors from the tedious, time-consuming work needed to use APA systems.

This paper presents three key contributions:

- **Base test set generation:** This paper proposes a simple specification for the inputs to a programming problem and describes an algorithm for creating a large set of test cases based on this specification.
- **Concise test set generation:** Given a collection of previous solutions to a programming problem, this paper describes test reduction methods for selecting concise subsets of the base test set with the property that all programs that fail on the base test set also fail on the concise test sets. Two types of concise test sets are presented: approximately minimal and gradated.
- **Comparison with expert test sets:** Finally, this paper compares the test coverage of these auto-generated concise test sets to that of expert-generated test sets, showing that the concise test sets provide greater test coverage.

2. BASE TEST SET

This section presents the first module of the FEAT toolchain: a generic, automated approach to generating an expansive set of test cases based on a simple yet powerful means of inductively defining the inputs of the function being tested. The resultant base test set B serves as the pool of candidate tests from which the two concise test sets are ultimately drawn (see Section 4). Further, this section presents a method for assigning a complexity heuristic to each test case. These heuristics enable the construction of gradated test sets, designed to give students actionable feedback (see Section 4.2).

FEAT employs a hybrid approach to base test set generation. A fully-exhaustive test set — *i.e.*, all valid combinations of arguments — is guaranteed to catch all incorrect programs, but generating such a test set is often impossible and nearly always computationally prohibitive. In contrast, generating a random test set is tractable, but has a high probability of missing edge cases. Thus, B is the union of two disjoint test sets: an exhaustive set covering a manageable subset of the input domain, and a random set providing partial coverage of the remaining portion of the domain.

2.1 Specifying the Domain for Test Cases

To generate the base test set, the user must specify four pieces of information in a config file: parameter types, argument domains for both exhaustive and random generation, and the desired number of random test cases.

Valid parameter types include all built-in Python types, such as lists, dictionaries, strings, and integers, nested arbitrarily deeply. Further, types may be classes, specified as composites of built-in types. Argument domains bound the values that FEAT will use as arguments for each parameter, and may be continuous or discrete.

FEAT also provides three optional means of further constraining the arguments, to facilitate testing functions with arbitrarily complex specifications. First, the config file language includes keywords, such as `sorted`, for constraining a single parameter. Second, dependencies may exist between parameters. The user can express such dependencies by defining and using variables; Section 2.4 presents an example of such a dependency.

Keywords and dependencies are lightweight, intuitive means of expressing constraints, but they do not always suffice. Thus, the user may provide a validation function with parameters identical to those of the function being tested. This function returns `True` if the arguments are valid, and `False` otherwise. FEAT uses this function to post-process its tentative base test set, discarding invalid cases.

Table 1: Base Test Set Results on Student Solutions

Prob. Index	Problem		Base Test Set Size			Num Student Solutions	Num Incorrect Solutions	Runtime (hours)
	Module	Function	Exhaustive	Random	Total			
1	2048 Game	`merge`	781	219	1000	55523	39617	12:16
2	Graph Theory	`compute_in_degrees`	285	215	500	5145	1577	0:16
3	Graph Theory	`in_degree_distribution`	285	215	500	5109	2805	0:15
4	Graph Theory	`make_complete_graph`	30	20	50	5120	1094	4:27
5	DNA Alignment	`build_scoring_matrix`	864	136	1000	3374	1090	12:59
6	Yahtzee	`expected_value`	627	373	1000	65217	37733	14:52
7	Yahtzee	`gen_all_holds`	462	538	1000	57758	35597	51:15
8	Yahtzee	`score`	461	539	1000	65652	26336	4:45

2.2 Auto-Generating Test Cases

Base test set generation has two phases: exhaustive generation and random generation. Exhaustive generation operates as follows. First, for each individual parameter, FEAT recursively generates all possible arguments. During generation, FEAT maintains metadata for each argument: a set of values for each variable that are compatible with that argument. FEAT uses this metadata to prune the search tree when it encounters disjoint sets of compatible values for the same variable across nested layers of the argument.

Once FEAT has amassed a set of consistent arguments for each parameter, it takes the cross product of these sets to generate all possible test cases, again pruning the search upon incompatibility. Last, if a validation function was provided, it uses this function to filter the set of test cases.

Random generation is simpler. While the size of the random test set is less than the target, FEAT randomly selects a value for each variable. It then randomly selects each argument subject to these variable constraints. If the resultant test case is not yet in B, it is added to the random test set.

2.3 Assigning Test Case Complexity

After generating B, FEAT assigns a floating point complexity $C[t]$ to each individual test case t. Complexity is defined as $C[t] = \prod C[t_i]$, where $C[t_i]$ is the complexity of the i^{th} argument and $C[t_i]$ is the product of the average complexity of each nested component. Component complexity is the length for sequences and value for primitives. By default, each component of each argument is incorporated into this heuristic. However, the user may optionally specify a subset of arguments and/or components to use.

As an example, consider the list `[1, 2, 3]`. The complexity of the list itself is its length (3.0), and the complexity of its contents is the average value (2.0), for a default complexity of 6.0. Imagine that the user declares the contents irrelevant; in this case, the complexity will be 3.0.

2.4 Configuration Example

In the dice game Yahtzee, a player rolls a set of dice and then holds some subset of the dice while re-rolling the remaining dice. Consider a function `expected_value` with three inputs — the dice held (a sorted tuple of integers), the number of sides on each die (an integer), and the number of dice to be re-rolled (an integer) — and computes the expected value after the roll. Naturally, the values of the held dice may not exceed the number of sides on the dice.

Figure 2 shows a sample config file for this function, with syntax slightly condensed for brevity. Parameter types are expressed under the `[types]` header using a combination of keywords (`sorted`) and Python types (`tuple`, `int`), with parentheses indicating nesting. For instance, `sorted tuple (int)` denotes a tuple of integers sorted in ascending order.

```
[types]        sorted tuple (int); int; int
[e domain]     0-3 (1-m); m; 1-3
[r domain]     2-8 (0-n); n; 0-3
[variables]    m 1-6; n 6-10
[complexity]   True (False); True; True
[num random]   545
```

Figure 2: Sample Config File

Exhaustive and random domains (`[e domain]` and `[r domain]`) for each parameter are expressed as inclusive ranges, representing lengths for sequences and values for primitive types. For instance, the first exhaustive domain (`0-3 (1-m)`) stipulates that the tuple of held dice has a length on [0, 3], where each element in the tuple is an integer on [1, m].

Dependencies between parameters — in this example, the fact that values representing rolled dice may not exceed the number of sides — are captured using variables. Consider the exhaustive case. The domain for the number of sides is m, defined under the `[variables]` header as [1, 6], and the upper bound on the values of the rolled dice is likewise m, ensuring that no die's value exceeds the number of sides.

Finally, the `[complexity]` header specifies which features of the input define its complexity. Here, `True (False); True; True` indicates that increasing the number of dice held, number of sides, and number of dice rolled increases the complexity of the test case, but changing the particular values of the held dice does not affect the complexity.

3. TESTING STUDENT SOLUTIONS

Once the base test set has been created, student solutions can be tested. Tests are first run on a reference solution to acquire the expected results, and then run on a corpus of student solutions, S, to ascertain correctness.

Recall that the goal of running the base test set on S is not merely to check these particular solutions. Rather, these solutions serve as a training set to identify high-quality test cases — those that trip up many erroneous solutions — and to derive the concise test sets (see Section 4). The concise test sets can then be used to efficiently test future solutions.

The tester maintains a mapping D of each solution, S_i, to the set of test cases that it failed on, B_i (*i.e.*, $B_i = D[S_i]$). This data enables selection of a concise subset of B without sacrificing coverage of any known incorrect solutions.

FEAT was used to generate and run base test sets for eight problems from four different programming assignments. Table 1 shows the inputs ($|B|$, $|S|$) and outputs (number of incorrect solutions identified, runtime) of the tester.

The number of student solutions tested ranged from 3639–73156; between 20–78% of those solutions proved incorrect. The size of the base test set was typically config-

ured to 500–1000 test cases, with one exception: `generate_complete_graph`, which takes as its input an integer and generates a complete graph with that many nodes. In this case, the simplicity of the parameter types led to diminishing returns as the size of B was increased.

Using such an extensive training set leads to strong coverage, but demands tradeoffs in time. Runtime varied greatly, influenced by $|B|$ and $|S|$; the minimum was 15 minutes, and the maximum was 51 hours. This non-trivial runtime motivates the use of a data-driven approach like FEAT, as performing semantic analysis on each training solution would likely prove computationally prohibitive. Further, Section 5 will show that $|S|$ can be reduced substantially while maintaining over 95% of the original coverage.

4. CONCISE TEST SETS

The base test set is designed to provide broad coverage, but is ill-suited for direct use in APA systems as its large size would slow the feedback cycle. This section describes two algorithms for extracting concise test sets from B. The first algorithm constructs an approximately minimal test set M with the property that every solution in S which fails on some element in B also fails on some element in M. The second algorithm generates a gradated test set G that is similar in size to M, but favors lower complexity.

4.1 Approximately Minimal Test Sets

As stated in Section 3, the tester computes a subset B_i of B for which the solution S_i disagrees with the reference solution. The goal of concise test set generation is to compute a subset M of B that contains at least one test case from each non-empty B_i. In other words, M has the property that $B_i \cap M \neq \emptyset$ over all B_i.

This problem corresponds to the classical hitting set problem, which is known to be NP-hard. Fortunately, there exists a simple greedy methods utilizing GRE heuristics [21] for computing a hitting set whose size is guaranteed to be within $\log(|B|)$ of the optimal size.

FEAT maintains a family F of the sets B_i that is dynamically updated as the algorithm proceeds. The GRE heuristic is *coverage*, where the coverage of a test case t with respect to F is defined as the number of sets in F that contain t. M is then constructed using a three-step iterative strategy:

1. Compute the test case t that has maximal coverage;
2. Add this test case t to M;
3. Remove those sets in F that contain t.

This process continues until F is empty. Since each entry in F corresponds to one student solution, this guarantees that M has the same coverage as B. The pseudo-code for this algorithm is as follows:

Algorithm 4.1: ApproxMinimalTestSet(D, B, S)

$M \leftarrow \emptyset,\ F \leftarrow \emptyset$
for each $S_i \in S$
  do $\begin{cases} B_i \leftarrow D[S_i] \\ F \leftarrow F \cup \{B_i\} \end{cases}$
while $F \neq \emptyset$
  do $\begin{cases} t \leftarrow \arg\max_{t \in B} \left|\{B_i \in F | t \in B_i\}\right| \\ M \leftarrow M \cup \{t\} \\ F \leftarrow F \setminus \{B_i \in F | t \in B_i\} \end{cases}$
return (M)

Table 2: Test Set Size Comparison

Function	Base Test Set	Concise Test Set	
		Minimal	Gradated
`merge`	1000	31	39
`compute_in_degrees`	500	4	5
`in_degree_distribution`	500	5	7
`make_complete_graph`	50	3	3
`build_scoring_matrix`	1000	2	3
`expected_value`	1000	12	19
`gen_all_holds`	1000	6	9
`score`	1000	9	14

4.2 Gradated Complexity Test Sets

Since Algorithm 4.1 repeatedly chooses test cases with maximal coverage, the resultant M is significantly smaller than B. However, it has one major drawback: the test cases in M tend to have high complexity, which is inconsistent with good testing practice. If used in an APA system, it may cause the system to report that a user failed a complex test case when a simpler example would be more valuable to the learning process.

One possible solution to this problem is to run the test cases in M in order of their complexity to ensure that a user solution fails on the simplest test in M first. However, this approach can fall victim to the situation where Algorithm 4.1 selects few simple tests.

A better solution is to balance the coverage of a test case versus its complexity when choosing a new test case. Algorithm 4.2 computes the gradated test set G by assigning a score to each test case: the ratio of its current coverage to the square of its complexity. In each iteration, Algorithm 4.2 selects the test case t with the highest score. Ties are broken by choosing the test case with lower complexity. After selecting t, the algorithm updates set family F by removing those test cases sets that have been covered by t. The pseudo-code below outlines the process:

Algorithm 4.2: GradatedTestSet(D, B, S, C)

$G \leftarrow \emptyset,\ F \leftarrow \emptyset$
for each $S_i \in S$
  do $\begin{cases} B_i \leftarrow D[S_i] \\ F \leftarrow F \cup \{B_i\} \end{cases}$
while $F \neq \emptyset$
  do $\begin{cases} t \leftarrow \arg\max_{t \in B} \left|\{B_i \in F | t \in B_i\}\right| \,/\, (C[t])^2 \\ G \leftarrow G \cup \{t\} \\ F \leftarrow F \setminus \{B_i \in F | t \in B_i\} \end{cases}$
return (G)

Table 2 shows the size of M and G for the same eight problems introduced in Section 3; in each case, both concise test sets are substantially smaller than the corresponding base test set. While each $|G|$ is greater than or equal to the corresponding $|M|$, the gradated test sets benefit from gradual growth in complexity. Figure 3 compares the complexities of the tests in M and G for the problem `expected_value`. M contains complex test cases with random ordering, shown as *M-Original* and sorted as *M-Sorted*. In contrast, G achieves the same coverage as M with notably less complexity.

The auto-generated gradated test sets were deployed in a programming MOOC. The APA system for that MOOC tests student' programs against sorted tests cases in G from

Figure 3: Comparison of test case complexities in two concise test sets for the problem `expected_value`

easy to complex. The system provides partial credit to erroneous programs based on which tests in G that they fail, rather than making a binary correct/incorrect judgment. Following the test-driven development principle [9], the system also returns the first — and therefore simplest — failed test case to the student, aiding the student in debugging their code. Students repeat this practice until they are satisfied with their scores, practicing their coding and debugging skills in the meanwhile.

5. METHOD SENSITIVITY ANALYSIS

Since FEAT selects test cases based on their results on the training set, the error detection ability of the resultant concise test sets on future programs is driven by the number and quality of programs in this training set. Too few or too similar programs can bias the test selection, leading to false positive verdicts on future submissions. On the other hand, a large training set will slow the testing phase, and may be difficult to come by. Thus, it is necessary to study the relationship between training set size and test set coverage in order to navigate these tradeoffs.

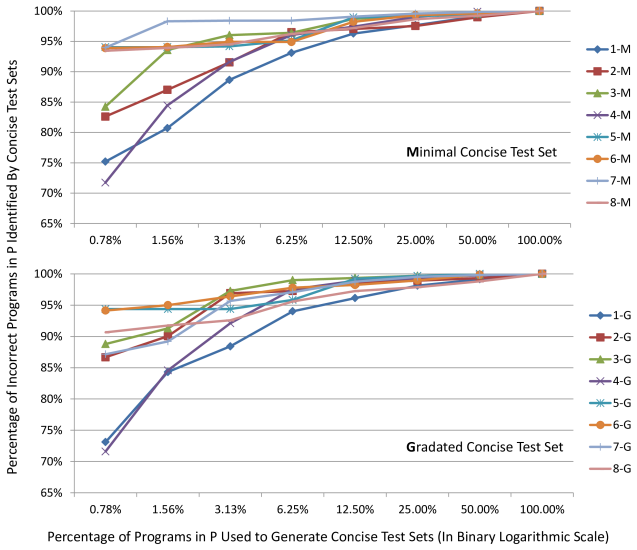

Figure 4: FEAT's coverage with respect to $|S|$

FEAT's sensitivity to training set size was evaluated on the same eight problems analyzed in Sections 3 and 4. For each problem, 1000 solutions were randomly selected to serve as the test pool P. From this pool, the percentage of programs randomly selected to serve as the training set S was gradually increased from 0.78125% – 100%. The resultant concise test sets were then used to evaluate all programs in P. To account for the randomness in the selection of S, this process was repeated five times for each problem. Figure 4 shows the average coverage for each training set size.

Intuitively, the percentage of incorrect programs in P identified by FEAT's concise test sets increases with $|S|$. More surprisingly, for all eight problems, using a mere 12.5% of the programs in P achieved over 95% test coverage of P. In other words, the test coverage of the generated concise test sets does not increase linearly with respect to the number of training programs. Rather, it increases quickly when $|S|$ is small and then plateaus. This suggests that FEAT can generate concise test sets with good coverage even using a reasonably small number of training programs.

These results also shows that more challenging problems, such as `merge` (1-M, 1-G) — for which 71% of student solutions were identified as incorrect (see Table 1) — tend to require a larger training set in order achieve high coverage, as there is a wider variety of ways in which students may err. As shown in Figure 4, the coverage of `merge` increases more gradually than that of easier problems. Yet, coverage still increases nonlinearly and eventually reaches a plateau.

6. COMPARISON WITH EXPERT TESTS

The gradated concise test sets computed by Algorithm 4.2 were deployed in the second session of the authors' programming MOOC. Student response was positive, with no reports of incorrect solutions passing the machine grader's tests. Additionally, this section presents a more methodological analysis of the coverage of the concise test sets versus the instructor-created test sets used in the first session of the MOOC. All student solutions for the eight problems studied in previous sections were analyzed.

Table 3 reports the results of comparing the coverage of the expert test sets and the auto-generated concise test sets. Recall that both M and G have the same coverage as their common base test set B. Thus, the "C" in the headings of this table simultaneously represents the results from B, M, and G. The four columns in the table indicate, respectively, the number of student solutions that pass both test sets, fail both test sets, pass the expert and fail the concise, and fail the expert and pass the concise.

Most importantly, note that the entries in Column 4 are very small — often zero — compared to those of the other columns. This reflects the fact that the concise test set caught almost all of the errors that the expert test caught. The expert test set occasionally caught a few incorrect solutions that were not identified by the concise test sets; however, this was rare, as this situation only occurs when a solution does not fail *any* of the tests in the base set. In this situation, the expert test set usually included tests that exploited problem knowledge that was not encoded in the configuration file provided to FEAT.

Another important observation is that, for many problems, the values in column three were large in comparison to column two. This indicates that the expert test set failed to detect a non-trivial fraction of the programs marked incorrect by the concise test set. For example, the expert test set for `score` allowed almost two-thirds of the solutions marked incorrect by the concise test set to pass. This indicates that

the expert test set had substantial deficiencies, correlating with the student complaints from the first session.

Table 3: Test Result Comparison

Function	✓ E ∗ ✓ C	× E × C	✓ E × C	× E ✓ C
`merge`	15860	37776	1838	25
`compute_in_degrees`	3562	1555	21	6
`in_degree_distribution`	2301	2742	61	3
`make_complete_graph`	4025	865	229	0
`build_scoring_matrix`	2283	1030	60	0
`expected_value`	27256	34573	2923	3
`gen_all_holds`	11424	31576	669	0
`score`	39316	9320	17016	0

∗ Note: ✓ E means "correct on expert test set" and × C means "incorrect on concise test set" and so on so forth

7. CONCLUSION AND FUTURE WORK

This paper introduced a data-driven approach to generating high-quality concise test sets for feedback and assessment in programming courses, as well as an implementation of this method in the FEAT toolchain. FEAT incorporates concise test set generation algorithms for producing near minimal test sets and slightly larger gradated test sets with simpler cases. The gradated test sets are designed to be student-friendly for feedback and evaluation. With simpler tests, the students can more easily understand and debug problems with their programs. Both algorithms take advantage of a diverse pool of student solutions, resulting in superior coverage as compared to expert-generated tests.

For the problems studied, 1.3–64.6% of the known incorrect programs were identified as incorrect by the auto-generated test sets, but erroneously deemed correct by the expert test set. Moreover, only 0–0.4% of the known incorrect programs were identified as incorrect by the expert test set, but erroneously deemed correct by the generated test sets. This highlights the advantages of utilizing student submissions to generate tests.

FEAT exploits a relatively large base test set in order to find incorrect programs. The fact that a few programs were identified as incorrect by the expert tests but not the auto-generated concise tests means that students made errors that were not exercised by any tests in the base test set. To increase coverage, the base test set could be expanded, either by adding more random test cases or by more targeted selection of additional test cases. One method to identify valuable additional test cases would be to incorporate semantic analysis of the reference solution or student solutions. The challenge would be to optimize the analysis so that it is feasible to apply to a large collection of solutions.

Another interesting question for future study is the relationship between the problem specification and the size of the resulting concise test set. For some problems, only a small number of tests were needed; for others, the size of the size of the concise test set was larger, indicating that there were more ways for a student to err. Quantifying this relationship may shed some light on the "difficulty" of various programming problems.

8. ACKNOWLEDGEMENTS

This material is based upon work supported by NSF Award CCF-1320860: "Computer-Aided Grading, Feedback, and Assignment Creation in Massive Online Programming Courses" as well as an NSF Graduate Research Fellowship under Grant No. 1450681.

9. REFERENCES

[1] Google Code Jam. https://code.google.com/codejam.

[2] LeetCode. https://leetcode.com/.

[3] PC^2. http://www.ecs.csus.edu/pc2/.

[4] Project Euler. https://projecteuler.net/.

[5] Rosalind. http://rosalind.info/problems/.

[6] TopCoder. https://www.topcoder.com/.

[7] UVa. https://uva.onlinejudge.org/.

[8] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[9] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] A. Belinfante, L. Frantzen, and C. Schallhart. 14 tools for test case generation. In *Model-Based Testing of Reactive Systems*, pages 391–438. Springer, 2005.

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

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, 2012.

[13] A. Kittur. Crowdsourcing, collaboration and creativity. *ACM Crossroads*, 17(2):22–26, 2010.

[14] R. Kumar, S. Singh, and G. Gopal. Automatic test case generation using genetic algorithm. *International Journal of Scientific & Engineering Research (IJSER)*, 4(6):1135–1141, 2013.

[15] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[16] S. M. Mohi-Aldeen, S. Deris, and R. Mohamad. Systematic mapping study in automatic test case generation. 2014.

[17] K. Muthyala and R. Naidu. A novel approach to test suite reduction using data mining. *Indian Journal of Computer Science and Engineering*, 2(3):500–505, 2011.

[18] S. Srikant and V. Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1887–1896. ACM, 2014.

[19] T. Tang, S. Rixner, and J. Warren. An environment for learning interactive programming. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 671–676. ACM, 2014.

[20] L. Wu, B. Liu, Y. Jin, and X. Xie. Using back-propagation neural networks for functional software testing. In *Anti-counterfeiting, Security and Identification, 2008. ASID 2008. 2nd International Conference on*, pages 272–275. IEEE, 2008.

[21] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[22] Z. Zhang and Y. Zhou. A fuzzy logic based approach for software testing. *International journal of pattern recognition and artificial intelligence*, 21(04):709–722, 2007.