Autograding in CS 61B

Ethan Ordentlich

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-151
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-151.html

May 12, 2023

Autograding in CS 61B

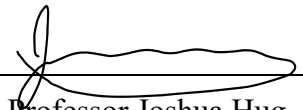
by Ethan Ordentlich

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Joshua Hug
Research Advisor

12 - May -2023

Date

* * * * * * *

Professor Adam Blank
Second Reader

5/12/23

Date

Autograding in CS 61B

Ethan Ordentlich
eordentl@berkeley.edu
UC Berkeley

ABSTRACT

This technical report describes the state of autograding in CS 61B in the Spring 2023 semester. Students submit to Gradescope, and receive feedback generated and delivered by a suite of autograder tests; BSAG, an autograder configuration tool; and jh61b, a Java test framework on top of JUnit 5 and Truth assertions. Students receive feedback from a wide variety of tests, including black box unit tests and random tests. We also describe nontraditional automated tests, including a complexity analyzer, introspector, and a tester for student-written tests.

1 INTRODUCTION

CS programs increased drastically in size over the past decade, especially at UC Berkeley. Currently, computing majors (CS, EECS, and Data Science) graduate around 2000 students per year, approximately 23% of degrees awarded. Individual courses have also increased in size, where CS 61B enrolls typically 1000 students in fall semesters, and 1500 students in spring semesters. While the size of course staff has increased, the large number of students makes it difficult for courses to provide help, even as the number of course staff increases.

One major time-consuming activity for course staff is grading, or evaluating student programs and producing feedback. In order to serve larger numbers of students, many courses use automated grading (autograding), where student-submitted programs are evaluated automatically by a program. Autograders are prevalent at many universities, and are a subject of active research and development [19, 20]. The usage of autograders helps courses scale to serve additional students as staff hours dedicated to grading are reallocated towards other course duties: office hours, answering questions on course forums, teaching section, or developing content.

However, naive autogenerated feedback, such as the verbatim output of a test suite, is difficult for students to interpret. As a result, staff in office hours act as pseudo-graders and spend time generating feedback from student programs on an ad-hoc basis in a short time period. In large courses, such as CS 61B, not all students are able to easily access office hours or work with course staff for clarification on autograder outputs. Autograder output has a higher impact in large and resource-stressed courses, because more students will interact with it more often without an alternative. Continued research and development into enabling autograders to produce more appropriate and pedagogically valuable feedback can reduce the amount of time that both students and staff spend interpreting grader outputs.

After the autograder evaluates the student's submission, it must also produces a "score" which can take various forms. The score may be numeric, computed by a count of tests passed, a weighted average, or by some other metric. It may also be a form of specifications grading, where the submission is described by a certain level of proficiency.

In this report, we describe the usage of autograders in UC Berkeley's CS 2 course (CS 61B: Data Structures and Algorithms). We first describe the autograding infrastructure that enables generating feedback and delivering it to a large number of students. Then, we describe newly implemented autograder features targeted at generating more effective feedback and the implications of these features. Finally, we propose future directions for autograder development in CS 61B.

2 RELATED WORK

2.1 Tools to Generate Feedback

A common autograding technique is to run students' submitted code against a set of unit tests, compare the actual output with expected output, and report the results. Since this only checks output, it is also known as *blackbox* autograding. However, the feedback that can be generated solely from blackbox autograding is limited in scope and pedagogical utility [20].

Haldeman et. al [13] describe CSF^2, a process by which instructors "bucket" student submissions to find common mistakes, and write hints accordingly. While their method achieves a high accuracy of error categorization, it does so in a CS 1 context, where the range of possible errors is distinct from a CS 2 context. Additionally, they note that incorrect hints were caused by "errors that would be difficult to detect with blackbox testing" [13].

Blackbox autograding focuses solely on the correctness of the student's program to an external observer. In contrast, *whitebox* autograding involves analyzing the program directly, whether dynamically or statically. Pedal [12], a Python framework for creating feedback, provides mechanisms to statically analyze Python code. It also provides a sandbox to execute student code, and unit test aspects of its execution, such as the number of calls to a specific function.

JavAssess [14] is a Java library that provides functions to inspect, analyze, and edit loaded Java source code at runtime. Parihar et. al [21] further explore the idea of modifying student programs with GradeIT by using program repair to fix minor syntax errors, allowing testing submissions that do not compile. We apply ideas from these and other libraries to evaluate and provide feedback on student implementations of data structures in a large course.

Additionally, instructors often want students to learn to test their own code, and therefore provide automated feedback on student test suites. Web-CAT [9], alongside unit tests, supports evaluating students via code coverage and mutation testing. The supported code coverage types include line coverage and method coverage of a reference solution, which are complex to report to students without giving away some information about the reference. Mutation testing, where a reference solution is modified so that effective tests should fail, encounters a similar problem. It is difficult to communicate mutations that are not caught to students, as they are buggy reference solutions. Testing Tutor [7] is a web-based platform that

provides "conceptual" feedback on the testing concepts that the student's test suite is missing. While it is not clear how a reference test suite is specified and tagged, the authors report that conceptual feedback results in better learning outcomes and is perceived more favorably by students when compared to detailed coverage-based feedback. Our approach does not directly use a reference test suite, instead instrumenting a reference solution to identify desired behaviors. We also integrate test suite feedback with the autograder as a whole.

2.2 Effects of Automated Feedback

One advantage of autograders is that they allow students to receive feedback on "submission" without waiting for a human. This can occur either through a provided test suite that can be run locally, or through submission to an external platform, typically a website. Mitra et. al [18] found that this instant feedback increases the quality of student submissions and improves confidence and engagement even after it is removed. They also find that on later assignments without instant feedback, students that received instant feedback on prior assignments performed similarly to students that did not. Leite and Blanco [17] find that students that receive human feedback achieve a higher conceptual understanding than students that receive automated feedback in an AI class. Their autograding scheme returns an estimated score immediately, and detailed feedback at a later fixed date. Even when the instantaneous feedback was extremely limited to a total estimated score, students still used the grader as a blackbox debugger. Kyilov and Noelle [15] find that students receiving binary instant feedback with low granularity attempted fewer exercises, and were twice as likely to cheat.

The availability of instantaneous feedback, regardless of the quality, enables students to repeatedly submit to the autograder without verifying their work beforehand. This runs counter to an important programming skill, checking checking one's own code for correctness. Baniassad et. al [6] observe this effect in a junior-level software engineering course, and attempt to curb it by penalizing students for submitting solutions that regress. The authors report that the penalty scheme reduced submissions to the autograder without significantly impacting project scores, at the cost of some student stress. Leinonen et. al [16] go further, and provide instantaneous feedback with score penalties for any extra submissions. They report that more students appreciated scheduled automated feedback (*fewer* opportunities for feedback) than instantaneous feedback with penalties for overuse. We do not utilize penalties, and instead apply a rate limiting scheme on submissions to encourage students to test their own code.

3 BACKGROUND INFORMATION

3.1 CS 61B

At UC Berkeley, CS 2 is called CS 61B: Data Structures and algorithms. It is the second of three courses required to declare the CS major. In Spring 2023, 78.5% of the students had taken CS 61A (CS 1 taught in Python), and 87% of the students intended to major in Computer Science, EECS, or Data Science.

The course is taught using Java 17 in the IntelliJ IDE, and is divided into three phases:

- Phase 1 (4 weeks), introduction to Java and usage of data structures
- Phase 2 (6 weeks), implementation of data structures
- Phase 3 (4 weeks), algorithms

Scoring in CS 61B uses a bag-of-points system, with specific point thresholds required for each letter grade. Students receive points by completing assignments (labs, homeworks, and surveys), submitting progress and check-in surveys, and exams. While discussion sections and worksheets are available for students to attend, they are not a formally scored component.

In CS 61B, the autograded programming assignments are Homework 2; Projects 0, 1, and 2; and 10 Lab assignments [4]. In total, these assignments make up 24.5% of the points available in the class. While higher emphasis is placed on exams, which make up 50% of the points available, students spend a significant amount of time working on programming assignments, especially the projects. Nearly all students are expected to get a full score on every programming assignment through automated feedback and resubmissions.

Students that struggle on these assignments can receive help in a variety of ways. They can attend lab section or office hours, where they can work with a member of course staff. If they have a question outside office hours, or their question requires significant staff debugging, they are also able to ask on the course forum, EdStem.

The projects require significantly more design and implementation than labs. The three projects that used significant autograding are:

- Project 0: Awakening of Azathoth, in which students implement hangman choosers and guessers using Java's built-in data structures (Phase 1).
- Project 1: Deques, in which students implement both a linked list-backed (1A) and array-backed version (1B) of a deque interface. A required component of this project is writing a comprehensive test suite (late Phase 1).
- Project 2: NGordNet, in which students implement the backend for an explorer of the Google NGrams dataset. This project was split into two halves. In the first half, students implement methods to parse and query the dataset. In the second half with optional partners, students implement a DAG of hyponyms (Phase 2).

In our discussion of autograding practices in CS 61B, we will focus primarily on Projects 0 and 1, to which the author contributed heavily.

3.2 Student Workflow

Students in CS 61B use two separate pieces of software to submit assignments and receive grades. They maintain their work in a course-provided GitHub repository and receive starter code by merging from a base "skeleton" repository, where each assignment is in a separate subfolder. Every assignment is completed in the same repository with the exception of partner assignments (which are completed in a second repository dedicated to the partnership). To submit, students commit and push their code to GitHub, and submit from this repository on Gradescope using the integration. The assignment autograder runs in a Gradescope Docker container, pulls the tests from a private GitHub repository, and outputs an

autograding log in Gradescope's output format [3]. Students are able to resubmit to the same assignment and receive instant feedback from the autograder, subject to limits on the rate of submission.

4 INFRASTRUCTURE

4.1 BSAG

CS 61B has a wide variety of assignments, each of which has different test suites and different scoring mechanisms, among other things. The course staff that design and deploy assignments are often not experienced with the details of the autograding software or how Gradescope reports output. Rather than having staff write code that runs tests and manually compiles a report, we use a configurable middleware application.

Prior to Spring 2023, CS 61B used ASAG (A Simple AutoGrader), which configured assignments with a YAML file. A config executes *steps* in sequence, where a step is a Python class with a callable method that can perform some computation, pass data to future steps, and/or create log entries. However, ASAG was difficult to use, as modifying it required access to the source code and knowledge of its architecture. Its internal representation uses nested Python dictionaries, which are difficult to use with type hints and static analysis. It also provided little feedback to staff when an assignment configuration was incorrect.

To resolve these issues, the author wrote and deployed a successor to ASAG called BSAG, the Better Simple AutoGrader, first used in Spring 2023 [10]. BSAG uses Python 3.10 best practices, and uses the Pydantic library [5] to specify and validate assignment configurations and other loaded data.

Similar to ASAG, BSAG provides some default steps, such as a step to parse Gradescope submission metadata and a step to output in Gradescope's results format. BSAG further supports loading external step definitions from external packages in two ways. Users can define their own entry point to BSAG, and provide their steps as local files; or install Python packages that contain BSAG steps, which are detected via Python's plugin mechanism. CS 61B currently uses the latter method to install a package with steps to compile and execute tests for Java programs, and aggregate their scores [2]; and a separate private package with steps to interface with the course's custom LMS. While certain packages are private, the explicit separation between the public core and the possibly private steps has allowed us to open-source BSAG.

An example BSAG configuration file is shown in Figure 1. The shared parameters are common settings that are reused across several phases, such as the location of the grader files, the student submission, and how long a command should take before timing out. The execution plan defines the order in which steps should be executed, with the configuration it should be executed with. For example, this configuration file parses Gradescope's submission information, followed by processing extensions from our custom LMS Beacon. These two steps compose together, as the latter step assumes that the first has executed, and modifies the due date in the loaded metadata. The jh61b steps are loaded from a separate package, and specify how the autograder's Java tests are compiled, run, and aggregated into Gradescope's score format. Finally, we process lateness, and output the results as Gradescope expects them.

```
shared_parameters:
  grader_root: "/autograder/course-materials/labs/
      ↪ lab01/grader/submit"
  submission_root: "/autograder/submission/lab01/src"
  command_timeout: 10
  assignment_id: lab01

execution_plan:
  - gradescope.sub_info
  - beacon.extensions
  - jh61b.check_files:
      pieces:
        ArithmeticTest:
          student_files:
            - Arithmetic.java
          assessment_files:
            - ArithmeticTest.java
  - jh61b.compilation
  - jh61b.dep_check
  - jh61b.assessment:
      piece_configs:
        ArithmeticTest:
          require_full_score: true
          aggregated_number: 1
  - jh61b.final_score:
      scoring:
        ArithmeticTest: 1
      max_points: 256

teardown_plan:
  - jh61b.motd
  - gradescope.lateness
  - gradescope.results
```

Figure 1: Sample BSAG Configuration File

This configuration runs the autograder and provides feedback on late submissions. One modification we could make to this configuration would be to check lateness before executing the autograder, and halting if the submission is after the due date, preventing students from continuing to work on a late assignment.

4.2 Java Assertions

Java programs are typically unit tested using assertions about the output of a student's program. Assertions check an *actual* value produced from a student program in some way, and is compared to an *expected* value in some way. Libraries can make several kinds of comparisons in assertions, depending on the data type, with the most commonly used being a check for equality between the expected and actual. While the actual value is derived from a student program, the expected value may come from a variety of sources, including a hardcoded value in the test, the contents of a file, or a known correct solution. There are several libraries that provide different methods of writing assertions, and output messages in different formats when assertions fail. We show the differences

between these libraries when comparing a long integer array that differs in the last element in Table 1. One point of note is that some libraries, such as AssertJ and Truth, use the *fluent programming* style, which uses method chaining to make assertions read left to right as in English. Prior to Spring 2023, CS 61B used JUnit 4 for legacy reasons. In Spring 2023, CS 61B switched to the Truth assertion library for its fluent assertions and better-formatted assertion errors; as well as the variety of assertions it provides [11].

4.3 Executing Tests

Executing tests requires a test framework, such as JUnit 5. These frameworks allow a variety of methods for users to specify which tests to run, such as filtering based on class, method name, tags, or various patterns. The framework also determines how test results are compiled and reported to the user. While JUnit 5 supports the Open Test Reporting XML format, along with various human-readable formats, these are ill-suited for a full autograding suite tests with useful information for scoring. For example, instructors may want to assign specific scoring information to specific tests, annotate tests with additional messages outside assertions and print statements, or suppress the output of certain tests.

To run tests, CS 61B uses the jh61b library [8]. The library implements a custom Junit 5 test engine that attaches a test execution listener *listener* that consumes a specific Java annotation on unit test methods, and outputs a JSON in Gradescope's results schema. It also captures standard output, which can optionally be included in the results. A test method may be annotated with `@GradedTest(number="4", max_score=2, suppress_output=true)`, describing the test number (for ordering in Gradescope output), the relative value of the test, and whether to include the test ouptut in Gradescope results. When an assignment executes multiple test suites, we use a BSAG step to merge the results files into a single Gradescope report.

4.4 Autograding Practices

4.4.1 Blackbox Tests. The most common kind of automated test currently used in CS 61B is a blackbox test, which simply runs the student's program to produce an output, then asserts that output is "correct". Our blackbox tests fall into two categories: small unit tests and large randomized tests. Small unit tests call a student's program once, or a few times on small hardcoded inputs. In contrast, large randomized tests call a student's program many times on a variety of inputs, which are often randomly generated. Although these tests are called randomized, we ensure that they are deterministic by seeding a random number generator for consistency, fairness, and reproducibility.

With a sufficiently large number of random inputs, it is likely that many paths in the student's code will be covered, which gives us high confidence that passing the random tests means they have implemented a fully correct solution. However, this is not a guarantee of complete correctness. There may be edge cases that are extremely unlikely to be covered by pure randomness. Additionally, generating and testing a sufficient number of random inputs may take an unreasonably long time, which can be frustrating to students and slow their development velocity. Furthermore, it is difficult to generate appropriate feedback from a failed random test, because the root cause of the failure is inherently unknown. Therefore, we provide both unit tests and several other kinds of tests to provide better feedback to students.

4.4.2 Limiting Feedback. One learning objective of CS 61B is software testing, or verifying the correctness of one's own code. However, the availability of automated feedback often causes students to rely on this automated feedback instead of writing their own tests. Much related work focuses on various mechanisms of penalizing repeated submissions, and CS 61B uses a similar approach.

Certain assignments, particularly the later projects, use a *token system* of to ratelimit student submissions. Students begin with a maximum number of tokens, and can "spend" a token to make a submission and receive feedback from the autograder. A token recharges after a specific length of time, and any submissions made when the student has 0 tokens receive a 0 score and no feedback.

4.4.3 Security. Velocity limiting and other methods of controlling the rate at which students can receive feedback requires that the test suite be private and not easily printable. Even with a public test suite with unlimited instant feedback, it is still good practice to prevent student code from writing to arbitrary files, terminating the test suite early, or faking behavior. While these concerns are likely inapplicable for the majority of students, it is still important to consider autograder security.

In CS 61B, we use velocity limiting on many assignments, and therefore have a need to secure the test suite. We implement this using the Java Security Manager and extending the `Policy` and `SecurityManager` classes. The `Policy` restricts the filepaths and system properties that student code is allowed to access. The `SecurityManager` works alongside the `Policy`, and converts `System.exit` calls that would interrupt the test suite into exceptions. To use these classes, our tests launch from a custom entry point to the jh61b runner, which instantiates and installs them before running the test suite. We use a separate entry point outside of jh61b as these security policies encode many CS 61B-specific details, and we would like to make jh61b usable by others. Secondly, separating them allows us to open-source jh61b while obscuring the precise details of our security mechanisms from students.

A significant upcoming challenge is that the Security Manager is deprecated and slated for removal without a replacement [1]. While we do not yet know in which Java version these features will be deleted, the fact that they are deprecated is strong motivation to update our practices. An alternate possibility for securing test contents is separation of student code from readable test code. After the test files are compiled, they may be copied to a separate system, or the file permissions may be updates so that the tests are run by a user that only has access to the bytecode files. Although it may still be possible for a student to leak the bytecode, it is significantly more difficult to discern the actual contents of a test from bytecode than source code. Another worthwhile practice is verifying a checksum of test data files to ensure that they have not been overwritten at time of use.

Table 1: Comparison of Java Assertion Libraries

Library	Assertion	Output

JUnit 4.13	`assertArrayEquals("not␣equal", ↪ expected, actual)`	`not equal: arrays first differed at element [999]; ↪ expected:<0> but was:<-1>`
		`Expected :0`
		`Actual :-1`

JUnit 5.9	`assertArrayEquals(expected, actual, " ↪ not␣equal");`	`not equal ==> array contents differ at index [999],`
		`Expected :0`
		`Actual :-1`

TestNG 7.7	`assertEquals(actual, expected, "not␣ ↪ equal");`	`arrays differ firstly at element [999]; expected value ↪ is <0> but was <-1>. not equal`

Hamcrest 2.2	`assertThat(actual, is(expected));`	`Expected: is [<0>, <0>, <0>, <0>, <0>, <0>, <0>, <0>, ↪ <0>, <0>, ...`
		`but: was [<0>, <0>, <0>, <0>, <0>, <0>, <0>, <0>, ↪ <0>, <0>, ...`
		The entire array is printed; we only show the first 10 elements.

AssertJ 3.24	`assertThat(actual).as("not␣equal"). ↪ isEqualTo(expected);`	`[not equal]`
		`expected: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...`
		`but was: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...`
		The entire array is printed; we only show the first 10 elements.

Truth 1.1	`assertWithMessage("not␣equal").that( ↪ actual).isEqualTo(expected);`	`not equal`
		`expected : ..., 0, 0, 0, 0, 0, 0, 0]`
		`but was : ..., 0, 0, 0, 0, 0, 0, -1]`
		`differs at index: [999]`

5 BEYOND BLACKBOX AUTOGRADING

We now discuss how we use autograding techniques to evaluate student programs beyond running the programs and checking their outputs.

5.1 Introspection

Java supports observing the structure of classes at runtime via its reflection library. One way that we provide this library to give more appropriate feedback is to look at field declarations to check that students are not declaring fields in a way that would lead to inefficient or incorrect implementations. For example, in Project 0, a common mistake is to store a map with English words as the key as a field at the class level, while it only needs to be a local variable. Reflection lets us test for this explicitly and provide a targeted feedback message.

In Project 1, where students implement a data structure, we use reflection more extensively. When students implement a deque backed by a linked list, we search for an inner class that "looks like" a doubly-linked node by checking that the defined fields are exactly two pointers to the same type, and a value. If the test fails, it reports that the student's class does not have a doubly-linked node This test is made locally available to students because it provides structural feedback to prevent students from going in an entirely incorrect direction, such as using a singly linked list.

In addition to verifying the node class, we also provide feedback on the internal representation of their linked list after various operations. After the linked list is constructed, we verify that they use one of the required structures with or without sentinel nodes. We also use a test that traverses the linked list and checks that the next and previous pointers are consistent with each other. This allows us to provide feedback at the point where the student's internal representation becomes incorrect, rather than at the point where it affects the program's behavior. A third test uses reflection to count the number of reachable nodes in the linked list by following pointers to check that adding values only adds one node, and that removing values appropriately removes one node.

5.2 Complexity Analysis

CS 61B also heavily emphasizes the asymptotic runtime of algorithms. Many assignments involve implementing a data structure or algorithm, and we would like to provide feedback to students that they are implementing the assignments to the desired runtime complexity.

Using Project 1 as an example, our versions require students to have constant-time addition and removal to both the front and back of their deques, regardless of the internal implementation. This means that students should not implement their array-backed deque by shifting elements, as this would take linear time. A second

example is ensuring constant-time access, as a common mistake is to implement access by repeatedly incrementing an idex by 1.

We evaluate runtime at the method level. To estimate runtime for a single method call, we perform operations on input sizes varying from 2^8 to 2^{16}, doubling each step. For each size, we first construct an input of the specific size, then measure the time it takes to call the method. We do this 100 times per size, and take the average runtime for each size. Then, we test whether the times "look like" they are constant or linear.

Assuming that the times were generated by a runtime function of the specific class, we apply a transformation function to the times so that the hypothetical runtime becomes a constant function. For example, if we are checking whether the runtime is linear, we divide each average time by the input size. Finally, we verify our assumption by running ordinary least squares to fit a line, and checking the R^2 value and slope. Empirically, this method is good at differentiating between constant and linear runtimes.

We are aware of implementations at other institutions, but none in the context of a large data data structures course.

5.3 Testing Tests

CS 61B also has a significant software engineering component, and also has the learning objective of students verifying the correctness of their code, both in designing test cases and implementing them. To support students in reaching this learning objective, we provide feedback on the quality of their tests. This feedback generation is automated so that it can be provided alongside the automated feedback generated for the other parts of the assignment.

We provide this feedback by manually instrumenting and annotating a reference solution with flags. While some flags simply check whether a line of code was executed, other flags are more complex and involve tracking the program state over multiple method calls. See Figure 2 for an example of an instrumented `size` method for a linked list-backed deque. When a student submits to the autograder, it runs their tests against the reference solution, and outputs the names of the flags that were hit. We also provided a separate autograder that would only give feedback on student tests, that was not subject to the submission rate limits present on the autograder for the full assignment. This is the first time that many students are writing tests and thinking explicitly about edge case behavior, and we would like to provide as frequent feedback as possible.

This approach evaluates the students' testing in terms of *behavior coverage*, or how many distinct significant behaviors their tests cover. The feedback on tests presented to students is not solely quantitative, but includes information on the sorts of behaviors that their test coverage is missing. Additionally, it is a curated set of important behaviors that have been identified by course staff. This set of behaviors can exclude insignificant or overly complex behaviors that course staff deem unnecessary, or identify and include rare edge case behaviors that random tests are unlikely to encounter.

Of course, since these flags are not likely to cover all possible behavior, some students get all the flags and pass their tests, but do not pass the exhaustive suite of autograder tests. This is particularly frustrating for students, as they've received feedback that their test suite is "acceptable", but then seemingly contradictory feedback

```
// Instrumented reference solution
public int size() {
    if (size > 0) {
        Flags.LLD_FLAGS.add("size");
    } else if (size == 0 && nRemove > 0) {
        Flags.LLD_FLAGS.add("
            ↪ size_after_remove_to_empty");
    } else if (size == 0 && nEmptyRemove > 0) {
        Flags.LLD_FLAGS.add("
            ↪ size_after_remove_from_empty");
    }
    return size;
}

// Flag test file
@Test
public void sizeTestCoverage() {
    Set<String> flags = new TreeSet<>(Set.of(
            "size",
            "size_after_remove_to_empty",
            "size_after_remove_from_empty"
    ));

    System.out.println("Possible flags: " + flags);
    flags.retainAll(Flags.LLD_FLAGS);
    System.out.println("Obtained flags: " + flags);
    assertThat(flags.size()).isAtLeast(1);
}
```

Figure 2: Flags for the `size` method in Project 1A

that it is not. We have taken two responses to this. The first is to update the flags over time, to report whether additional behaviors are covered by student tests. The second is to use this as a learning opportunity to teach students that test suites are not perfect, and that passing a set of tests does not necessarily mean that their code is completely correct.

5.4 Scoring

With new kinds of autograder feedback, we also re-evaluate our scoring mechanism. Prior to Spring 2023, CS 61B gave each test method a point value. When a student submitted and received feedback on their test suite, their grade on the assignment would be the total number of points from passed test methods.

This method of scoring student programs by assigning each test a point value has several drawbacks related to how partial credit is reflected. Firstly, points are fungible, which means that a test can be worth the same number of points as a different test. Two tests worth the same number of points may not indicate the same level of completion, especially if both are small random tests. A program may miss a particular edge case that causes it to fail more tests than a different program that misses a different edge case. Ideally, both students would fix their errors and resubmit, though they may decide that their program is "good enough". The reflection tests earlier that check the internal structure of a program, aside

from providing meaningful feedback, also check that the student followed the specification for the assignment. If a student passes tests while not following the specification, it may not be desirable to give credit. Finally, adding additional tests or editing the test suite because new tests requires extra cognitive load to determine how they should be weighted against existing tests.

As we introduce additional kinds of feedback, we revise our scoring mechanisms to better describe student progress through an assignment. For Project 1A, we are able to give partial credit for constructing a correct representation of an empty linked list, then exhaustively test adding elements, followed by exhaustively testing adding and removing elements. Each subpart requires passing all relevant tests to receive credit, as a step towards specifications grading at the assignment level. As individual tests no longer contribute to a student's assignment score, we add many additional tests that exercise specific edge cases, such as repeatedly adding and removing elements from an empty list. We use similar tests in Project 1B, instead checking that the student has instantiated an array, and providing tests that exercise various possible states of the backing array.

However, switching to this form of grading must be done carefully to ensure that each subpart is of reasonable size. In Spring 2023's Project 2B, students design and implement a graph on the WordNet dataset, and implement several search algorithms to explore it [22]. Due to the scope of the project, students were allowed to complete it in partners.

The test suite for the project directly translated from a previous term, with each fully functioning search feature designated as a supbart. Each feature is tested by several medium to large randomized tests. Since there is a large design and implementation component to complete even one of these features correctly, the bottom 3% of students did not receive any credit despite spending a significant amount of time on the project. Some of these students had implemented entire subparts nearly correctly, except for small edge cases that the randomized tests failed on. This assignment used aggressive velocity limiting, with 4 tokens that regenerated every 24 hours, which limits the the feedback that students are able to obtain from the autograder.

Two thirds of the student body in total received at least one extension to complete the assignment. Of these, approximately 60% received an extension of 9 days (over spring break) and another 30% received an extension beyond that. In contrast, on each of Projects 1A, 1B, approximately one third of the students received at least one extension.

While it is likely that the difference in difficulty explains the increase in extensions, our hypothesis is that the lack of effective feedback in Project 2B is a heavy contributor as well. The poor feedback combined with long office hours waits caused students to not be able to receive the necessary guidance to correct their solutions.

6 FUTURE WORK

There are many directions for future work in developing autograders that can give more effective feedback beyond blackbox testing.

One technique is using a Java agent to modify student class bytecode when it is loaded by the Java classloader. For example, the previously mentioned complexity estimator can be made much more consistent by inserting a step counter after every instruction of a student's code, and fitting to the number of steps. We are aware of an (unpublished) implementation in CS 2 at Caltech by Adam Blank.

A common approach that TAs use at office hours is using the debugger to demonstrate why a program state is incorrect in the middle of a method. We are also interested in exploring using the Java Debugger Interface to expose, check, and provide feedback on the internal state of student programs in an autograder.

Finally, while we have thus far viewed autograder feedback through the lens of a novice student programmer, we are also interested in how autograders can enable experienced course staff to give effective help more efficiently. Automated feedback can contain more information that a student would have difficulty parsing, but a trained staff member would be able to use to identify possible causes. This would help reduce the office hours time spent on problem diagnosis, allowing staff to spend more time directly instructing students.

ACKNOWLEDGMENTS

Thanks to Adam Blank for their guidance and support in pursuing a career in CS education; Lisa Yan for her advice on writing this report; and Josh Hug for introducing me to teaching in CS 61B at Berkeley and providing feedback and guidance. Additionally, thanks to the course staff of CS 61 B(L) and CS 61BL for bearing with my autograder implementations and providing useful feedback. We would also like to acknowledge CS 161 course staff at Berkeley for the inspiration for testing tests with flags.

REFERENCES

[1] 2022. JEP 411: Deprecate the Security Manager for Removal. https://openjdk.org/jeps/411
[2] 2022. jh61b for BSAG. https://github.com/Berkeley-CS61B/bsag-jh61b original-date: 2022-12-23T06:36:10Z.
[3] 2023. Autograder Specifications - Gradescope Autograder Documentation. https://gradescope-autograders.readthedocs.io/en/latest/specs/#output-format
[4] 2023. Course Info | CS 61B Spring 2023. https://sp23.datastructur.es/about.html
[5] 2023. Pydantic. https://docs.pydantic.dev/latest/
[6] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Virtual Event USA, 1062–1068. https://doi.org/10.1145/3408877.3432430
[7] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. 2021. A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Virtual Event USA, 87–93. https://doi.org/10.1145/3408877.3432417
[8] CS 61B Staff. 2022. Berkeley-CS61B/jh61b. https://github.com/Berkeley-CS61B/jh61b original-date: 2022-12-12T06:24:01Z.
[9] Stephen H Edwards and Manuel A Pérez-Quiñones. 2008. Web-CAT: Automatically Grading Programming Assignments. *SIGCSE Bull.* 40, 3 (June 2008), 328. https://doi.org/10.1145/1597849.1384371
[10] Ethan Ordentlich. 2023. BSAG (Better Simple AutoGrader). https://github.com/Berkeley-CS61B/BSAG original-date: 2022-12-07T07:23:16Z.
[11] Google. 2023. Truth - Fluent assertions for Java and Android. https://truth.dev/
[12] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. 2020. Pedal: An Infrastructure for Automated Feedback Systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, Portland OR USA, 1061–1067. https://doi.org/10.1145/3328778.3366913
[13] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing Meaningful Feedback

for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 278–283. https://doi.org/10.1145/3159450.3159502

[14] David Insa and Josep Silva. 2018. Automatic assessment of Java code. *Computer Languages, Systems & Structures* 53 (Sept. 2018), 59–72. https://doi.org/10.1016/j.cl.2018.01.004

[15] Angelo Kyrilov and David C. Noelle. 2015. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 122–126. https://doi.org/10.1145/2828959.2828968

[16] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A Comparison of Immediate and Scheduled Feedback in Introductory Programming Projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1 (SIGCSE 2022, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 885–891. https://doi.org/10.1145/3478431.3499372

[17] Abe Leite and Saúl A. Blanco. 2020. Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, Portland OR USA, 44–50. https://doi.org/10.1145/3328778.3366921

[18] Joydeep Mitra. 2023. Studying the Impact of Auto-Graders Giving Immediate Feedback in Programming Assignments. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM, Toronto ON Canada, 388–394. https://doi.org/10.1145/3545945.3569726

[19] Sidhidatri Nayak, Reshu Agarwal, and Sunil Kumar Khatri. 2022. Automated Assessment Tools for grading of programming Assignments: A review. In *2022 International Conference on Computer Communication and Informatics (ICCCI)*. 1–4. https://doi.org/10.1109/ICCCI54379.2022.9740769 ISSN: 2329-7190.

[20] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3 (Sept. 2022), 1–40. https://doi.org/10.1145/3513140

[21] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Bologna Italy, 92–97. https://doi.org/10.1145/3059009.3059026

[22] Princeton University. 2010. About WordNet. https://wordnet.princeton.edu/