Using Programming Autograder Formative Data to Understand Student Growth

Ruben Acuña
School of Computing and Augmented Intelligence
Arizona State Univerity
Tempe, Arizona, USA
Ruben.Acuna@asu.edu

Ajay Bansal
School of Computing and Augmented Intelligence
Arizona State Univerity
Tempe, Arizona, USA
Ajay.Bansal@asu.edu

***Abstract*—This Innovative Practice Full Paper presents the experience of using formative data produced by an autograder tool to understand how students complete assignments in a second year computing course. Autograding has increasingly been used in computing courses to evaluate student work. Using autograders supports the scalability of classes and provides other advantages such as accurate assessment, reproducibility over semesters, and rapid feedback. As a consequence of using an autograder, students produce a trace of their problem-solving process while completing assignments. In contrast to the summative assessment performed on a final submission, this trace captures the formative steps that were involved in the development of the final submission. Analyzing this trace can provide insights into student problem-solving methodology as well as the structure of the problems being assigned, and the impact of classroom interventions during an assignment. To produce this view into student learning, we have developed an autograder with the initial goal of improving student instruction. It uses a combination of static and dynamic analysis techniques to support accurate assessment and the ability to check for more complex requirements than can be captured by approaches that compare program output with a ground truth. The tool has been used in a sophomore level course on data structures & algorithms. This course uses the Java programming language and introduces topics such as lists, searching, sorting, binary search trees, priority queues, hash tables, and graphs. Our current analysis work focuses on analyzing traces produced by students to identify difficult parts of an assignment (as represented by grading regressions), and the path through possible solution states that students take as they complete an assignment. In this paper, we discuss the design of this tool, what insights the data captures provide into student learning, and give ideas for future directions supported by gathering formative assessment data.**

***Index Terms*—engineering education, software engineering, student assessment**

I. Introduction

Substantial practice is a critical aspect of programming courses, however, assessing student work is time intensive [1]. To address this, many tools have been developed to perform automated assessment (AA) of student work [2], [3]. One significant advantage of using AA is the ability to produce immediate feedback to students [1]. In the process of evaluating student work, an AA tool produces a sequence of assessments preformed on a student's submission at different points in its development. This trace of student activity provides visibility into the process by which students complete assignments, which has the potential to provide insights into a student's problem-solving process. For example, a trace can indicate if a student is following a logical sequence of steps when completing an assignment, or where they may have had trouble completing the assignment. This information can suggest areas for improvement in instruction or assignments. In this paper, we describe our approach to understanding how students complete assignments, first by developing an automated assessment tool for a course on data structures & algorithms, and then proposing two methods for automatically analyzing the trace it produces. This particular course is part of the critical pathway in a four-year computing program. Students take this course in the second year, at which point they are proficient with a general programming language (Java), and are learning techniques for the development of complex software.

Our autograder was originally developed to improve student learning through the generation of rapid formative feedback. The autograder was developed in Python, using the Gradescope [4] cloud environment to execute assignments. It uses a combination of static and dynamic analysis approaches to assess functional and non-functional requirements of a student's submission. This combination of approaches is important to capture detailed information on the state of a submission. The autograder automatically stores the trace of student assessments as they complete assignments. We propose a simple model for student traces, which our autograder supports but which may be implemented by other systems. In this work, we focus on techniques for understanding rubrics, and the solution path which students take when completing assignments. First, we discuss fragility scores which indicate which parts of an assignment students have difficulty completing. This method can be used to identify parts of an assignment that should be revised. Second, we discuss the Student Action Graph (SAG) which captures the different states of an assignment as it is completed, and the relative probability that a student working on an assignment passes from one state to another. This method can used to evaluate the process by which students complete assignments. In a single assignment, it can be used to check if students apply a logical plan during development. With multiple assignments, it can be used to determine if students have improved in their software development process.

II. Related Work

Automated assessment tools play an important role in introductory programming courses where they are essential to assessing large classes and providing feedback [1]. AA has been used during the instruction of programming for many years, and has advantages such as high availability, grading consistency, and rapid feedback [3]. The review conducted by Ihantola et al. [2] in 2010 investigated what features have been introduced into AA systems as well as future areas of development. They found that common features included support for Java, learning management system (LMS) integration, applying unit testing, and providing sandboxing of submissions. They also report a trend to limit the number of resubmissions to reduce trial-and-error problem-solving. In contrast, [3] indicates unlimited submission success as a factor leading to the success of an AA system, together with quality assignments, clear formulation of tasks and inputs, and good feedback. In addition to assessment functionality, a comprehensive system should also have features such as course administration, content authoring, and user-friendly online testing [5]. Integration with an LMS is important for the overall learning process [6]. In the more recent survey by Ullah et al. [1], 17 tools AA tools currently in use were reviewed. Tools used a combination of approaches to evaluating submissions. Static approaches include using software metrics, style checkers, and similarity analysis. Dynamic approaches focused on performing pattern matching on program output for test cases. Static approaches can be beneficial since they do not require code to compile, and are thus able to provide feedback at early stages of development [7]. One limitation of AA systems is that systems are often specific to a course or institute, explaining the large of number systems available [6]. In addition to improving instruction through feedback [1], which may take many forms [8], using AA tools has the potential to engage self-regulated learning skills as suggested by Garcia et al.'s work on e-learning tools [9]. It was found that tools encouraged self-evaluation as well as goal setting and planning [9]. In a survey of classroom reports on impacts to student learning when introducing AA, it was found that even with different approaches and policies, student outcomes improved [10]. Although there have been concerns that students dislike AA systems [10], more recent studies found that students had a positive appraisal [11].

Several commercial cloud-based tools for managing different types of assessments have also been developed. Gradescope [4] is a platform that provides an improved workflow for written assessments such as exams and quizzes. A typical workflow involves creating an assignment in Gradescope, uploading the original and student submissions, and then using AI-assisted grading to quickly mark materials. It also provides support for programming assignments by providing a cloud environment. Although Gradescope does not directly provide functionality for grading programs, it provides the essential technical environment for deploying an autograder. Another approach is provided by the ZyBooks platform [12]. ZyBooks provides dynamic textbooks that help to engage students through animations and embedded questions. ZyBooks are typically modeled after existing textbooks but provide functionality for instructors to customize them by selecting sections and problems. A related tool, ZyLabs, provides support for automatically deploying programming labs to students.

III. Methodology

A. Course Context

We aim to analyze the trace of student problem solving in a sophomore class on Data Structures & Algorithms (DS&A) in an ABET accredited Bachelors of Science in Software Engineering (BSSE) program. This program is offered both face to face, and asynchronous online, and has seen significant growth in the past few years. DS&A courses are critical in computing curriculums, as they teach fundamental topics used by other courses. Common topics include abstract data types (ADT), analysis of algorithms, sorting, binary search trees, hashtables, graphs, and dynamic programming. Most topics students will encounter during an interview come from a DS&A course. In our program, DS&A is a pre- or co-requisite to four required courses: software design, software agility, operating-systems, and distributed systems. Due to its centrality, improvements to a DS&A course offer the potential for significant and program-wide improvements in student attainment. Our particular course has four high-level goals:

- Understand and apply big-O analyses of algorithms.
- Gain experience in the object-oriented programming paradigm.
- Judge the appropriateness of alternate implementations of elementary data structures.
- Learn application of elementary data structures.

The course uses Algorithms by Sedgewick and Wayne [13] and is delivered with a mix of lecture and active learning activities. Students are assigned weekly homework, which are comprised of short answer problem sets (e.g., asking student to analyze a scenario and make some decision) and Java programming assignments. Programs are substantial in size, often without base code, to ensure that students have proper practice with the course materials and programming in general. Students are given PDF descriptions of expected behavior, and interface files which define what students should construct. Assignments were originally assessed by graders manually against an instructor provided rubric. See Table I for an overview of the programming assignments.

B. Autograder

Based on the need to support our specific class, and to better focus on producing a trace of student activity for an assignment, we developed our own automatic assessment tool. In particular, we choose to support granular assessment of student work (e.g., examining properties of methods), rather than by text comparison of outputs as supported by other tools (see [8]). The basic design of our system is shown in Figure 1. In the general flow, students submit assignments to a server which executes our tool on it, and then returns the feedback/results

TABLE I
PROGRAMMING ASSIGNMENTS IN DS&A COURSE.

Module	Topic	Task
M1	Data Abstraction	Create a matrix ADT.
M2	Linked Lists	Create a linked Deque ADT.
M3	Linked Lists	Create a layered List ADT.
M5	Basic Sorting	Benchmark sorting algorithms.
M6	Mergesort	Divide and conquer algorithms.
M9	Symbol Tables	Create a binary search tree ADT.
M10	Hashtables	Create two hashtable ADTs.
M12	Graphs	Create a toplogical sort algorithm.

to the student as input to their learning process, and also to the instructor to understand student progress. At a high level, the autograder for a particular assignment has two components, which perform static and dynamic analysis of the code as appropriate (suggested by [1]). The static analysis portion is shared between assignments, while the dynamic portion varies.

The autograder functionality is implemented on top of the Gradescope cloud environment for virtualized programming assignments. Gradescope provides: 1) a web interface that manages a roster and assignments, parses JSON formatted grade data, and interfaces with an LMS (such as Canvas), and 2) the infrastructure to spin up docker containers that users have developed. Gradescope leaves the generation of the JSON grade file to the user. Our tool generates this file based on student submissions.

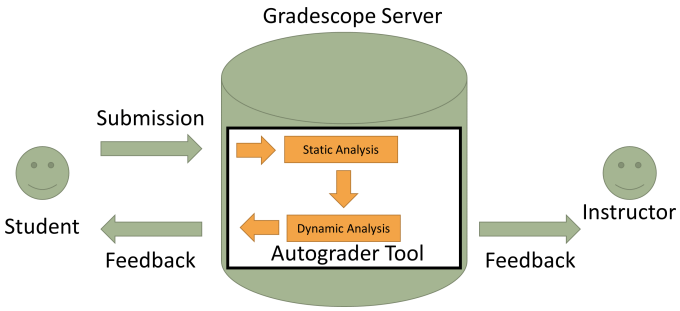

Fig. 1. Overview of autograder system. Students submit to a system hosted on Gradescope, which returns feedback to them and (on demand) the instructor.

In developing the autograder to capture the trace of a student's problem-solving process, we tried to minimize the number of changes made to assignments as they were migrated to the autograding platform. This motivated us to apply both static analysis, focusing on the structure of the program, and dynamic analysis, focusing on the run-time behavior, approaches. The static analysis portion of our tool was implemented in Python. It provides functionality to check various properties of a students submission that cannot be easily determine through test cases, such as the performance of a particular method. The dynamic analysis portion of our tool is based on executing a set of JUnit[1] test cases. This approach was suggested by Gradescope. In general, JUnit provides a clear way to express tests for different aspects of assignment functionality, however, its use requires the development of tests aligned with instructional goals.

[1]https://junit.org

C. Trace Model

The assessment of an assignment by an autograder is structured using a rubric, which is a set of test cases. This rubric can be represented as a vector R with a length equal to the number of tests, and where each element is a positive integer that represents the maximum score that may be earned on that criteria of the rubric. Thus, $sum(R)$ is the maximum score for the assignment. Each time a student submits their assignment for assessment, an autograder produces a vector A, where $|A| = |R|$ and $A_i \in \{0, R_i\}$. Let the trace of a student's progress on an assignment be a sequence of assessments denoted T. T_i indicates a sequence of all A_i, and T_{ij} indicates the score received on criteria i during assessment j. Within T, the number of assessments, and the time at which the assessment occurred, is arbitrary (that is: students only submit when they feel the need).

As an example, consider an assignment with three rubric criteria, with the first and last parts worth 1 point, and the middle worth 2. Here: R = [1, 2, 1], with the maximum score being 5. If a student submits a version of the assignment where no criteria is satisfied, then A = [0, 0, 0]. If the student then completes the last criteria, A = [0, 0, 1]. When they have completed all criteria, A = [1, 2, 1]. Then, T = [[0, 0, 0], [0, 0, 1], [1, 2, 1]] for this one student. This trace indicates that the student made two attempts before meeting the first two criteria, and only one attempt for the last one.

We may choose to simplify a rubric R and any associated trace T by grouping certain parts of the rubric together. In this case, each element in the simplified rubric R' is calculated as a sum of a set of indices in the original rubric. An assessment is similarly simplified. For example: $R'_0 = sum(R_0 : R_4)$, and so $A'_0 = sum(A_0 : A_4)$. Every element in R (or A) must be summed exactly once in the simplified vectors. Thus: $sum(R') = sum(R)$, and $sum(A') = sum(A)$.

In our system, the autograding tool produces (and stores) T as each student works on an assignment. A class will be said to have N students, meaning N traces will be produced for an assignment.

D. Fragility Scores

The fragility score (FS) for a rubric criteria represents the relative likelihood that a specific criteria will be difficult to implement due to regressions. Computing FSs for a rubric provides a way to assess which parts of an assignment are difficult for students, and can suggest where refinements to the assignment may be made. A score of .5 indicates that a criteria's status (working or broken) never changes over the entire trace. A score near 1 indicates that criteria status has changed multiple times. In general, we would expect that a criteria with a lower FS is easier for a student to implement.

FS are computed separately for each criteria in a rubric. Initially, we compute fwb as the number of times in the trace for a criteria that a submission goes from working (non-zero

score) to broken state (zero score). Likewise, we find fbw as the number of times the submission goes from broken to working. Then,

$$fs = \frac{1}{1 + e^{-(fwb+fbw)}}$$

As examples, consider:

- A baseline where nothing changes during the trace: [0, 0, 0, 0, 0, 0]. This yields fs = .5, fwb = 0, fbw = 0.
- When a feature is implemented and does not break: [0, 0, 0, 1, 1, 1] This yields fs = .73, fwb = 0, fbw = 1.
- When a feature is implemented and later breaks: [0, 0, 0, 1, 0, 1] This yields fs = .95, fwb = 1, fbw = 2.

In all of these examples, we are following a single criteria. Analyzing a complete rubric (simplified or otherwise) requires repeating this process for each criteria. Hence, we produce a fragility vector (F) for each student where $F_i = fs(T_i)$, where T_i represents a sequence composed of each A_i in sequence. If we suppose that the three examples are different criteria, then F = [.5, .73, .95]. Such scores would indicate that each of the criteria were successively harder for the student to implement.

To make understanding the FS for an assignment more accessible, we apply clustering to determine prototypical student experiences. We use the Python package sklearn [14] to apply the k-means algorithm to the fragility vector produced for each student. This results in a mapping of student fragility vectors to one of several clusters. For each cluster, we compute the mean FS (per criteria) between all students assigned to that cluster, and call it the *prototypical student* for that cluster.

E. Student Action Graph

As a method to understand the actions which a student may take when completing an assignment, we take inspiration from the idea of Markov chains (see [15]) and create what we refer to as a Student Action Graph (SAG). Note that while we construct something similar to a Markov chain, the independence assumption does not hold (since it does account for changes in the state of the student, as they proceed through the assignment). In a SAG, each node represents the state of the assignment, that is, some assessment A. Each transition represents a student working on a submission and performing an action which changes the assessment of the assignment. For instance, they may complete a feature in the rubric, or break a feature that was already functioning. Each transition has a weight that relates to the probability that when a student is in a state, they will perform that action which leads to another state. Note that while the transition weights are computed as probabilities, they are relative to the traces which are being used and may not represent how an arbitrary student moves through the assignment.

In general, a student's progress through an assignment forms a path from a source node (where score is 0) to a terminal node (where the score is maximal). Students who do not complete the assignment will have a path that terminates somewhere else in the SAG. Potentially, a SAG may contain $2^{|R|}$ possible states, as all the combinations of the rubric. In practice, not all parts of a rubric are independent (e.g., when a student implements criteria R_i, they always implement R_j as well), so not all states will be present in the final graph. Transitions only exist in a SAG when there is a T such that a student passed through those consecutive states.

Before constructing a SAG, three pre-processing steps occur: 1) all $A \in T$ where $sum(A) = 0$ are removed. (In practice, this represents removing any assessments performed on code which did not compile, and therefore does not provide meaningful information). 2) All student traces of length less than 8 are omitted. This reduces the bias that can occur in the edge probabilities. 3) Every T is adjusted to begin with an evaluation with a zero score (representing the state prior to a student beginning an assignment).

To construct the graph (following the idea of a Markov chain), we initially start by creating a node for each unique A in all traces. At the same time, we calculate α_i which counts the number of times each A is seen (i is taken to be an index into all possible assessments), and α_{ij} which is the number of times A has a transition to A' (meaning that $..., A, A', ...$ is part of T). We then calculate edge weights as: $w_{ij} = \alpha_{ij}/\alpha_i$.

After the graph is constructed, we simplify its structure with three steps: 1) Remove self-loops from each vertex V while maintaining $sum(weight(out\_edges(V)) = 1$. 2) Remove all nodes with exactly one predecessor and one successor. Weights are again preserved. 3) Repeat as done in step 1. These simplifications are intended to produce a SAG that is meaningful while not being too complex to understand.

IV. Results

In this work, we focus on traces obtained from the first assignment in our DS&A course. Although this assignment is simpler than others in the course, the infrastructure we have developed applies to any assignment for which we can generate the previously described trace model.

In the first assignment, students are asked to create a matrix ADT. This assignment is given during the course's module on data abstraction and the ADT is required to use best practices for an immutable type. Students are provided with a PDF that gives some background on the assignment, and then a Java interface which defines a set of nine methods that must be implemented. Students also have to create a default constructor for the class. For each method, students are given a method signature and a description of the expected behavior of the method. Some methods are common in Java such as the constructor, equals, and toString, while others are specific to the concept of a matrix (e.g., scale, plus, minus, multiply).

This assignment is assessed by a set of 40 test cases. As introduced previously, we chose to simplify this rubric by grouping the test cases by the methods that student were required to implement. This results in a simplified rubric containing 10 criteria for the assignment. This rubric is shown in Table II. This simplification meant that, for example, an assessment in a trace might look like [4,2,1,1,0,0,0,0,0,5], which corresponds to a submission where the constructor, three simple accessors, and toString have been implemented

fully. Alternatively, a trace [2,2,1,1,0,0,0,0,0,2] would indicate that although the three simple accessors were complete, the constructor and toString were only partially working.

TABLE II
SIMPLIFIED RUBRIC FOR MATRIX ASSIGNMENT.

Part	Method	Description	Pts
Q1	Constructor	Create a new matrix object from a 2D array.	4
Q2	getElement	Returns the element at particular point.	2
Q3	getRows	Returns the number of row.	1
Q4	getColumns	Returns the number of columns.	1
Q5	scale	Returns this matrix multiplied by a scalar.	3
Q6	plus	Returns this matrix added with another.	3
Q7	minus	Returns this matrix subtracted by another.	3
Q8	multiply	Returns this matrix multiplied by another.	5
Q9	equals	Returns true if this matrix matches another.	5
Q10	toString	Returns a string representation.	5

The data we analyze was gathered from a section of the DS&A course taught at our university during spring 2022 in the face-to-face modality. The roster included 43 students, all undergraduates. As noted later, some analyses use fewer than 43 traces. This is due to both student submission counts, and pre-processing of traces (e.g., discarding short traces). The aggregate grades for this assignment were: 33 As, 2 Bs, 3 Cs, 5 Ds (all students who did not submit). Letter grades were determined based on a percentile of the total score (100%-90% A, 89%-80% B, 79-70% C, $<$ 70% D).

A. Fragility Scores

The heatmap for the fragility scores produced by our analysis is shown in Figure 2. This plot shows the constructed prototypical students versus the 10 criteria in the simplified rubric. A lighter color indicates a lower fragility score while a darker color indicates higher fragility score.

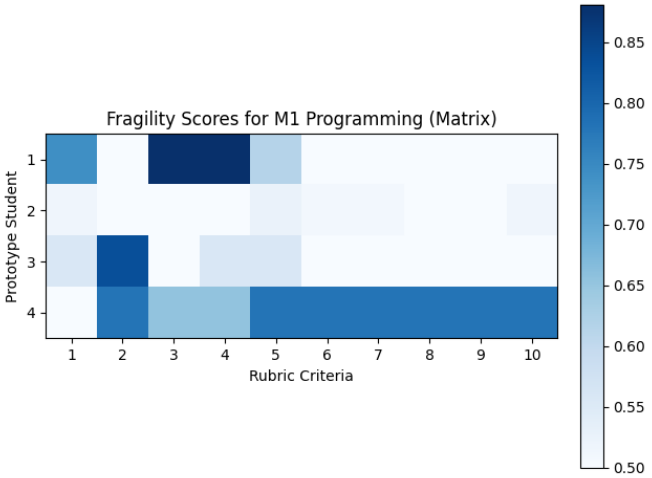

Fig. 2. Heatmap for fragility scores of prototypical students.

This plot was produced by analyzing a set of 34 submission traces. Each trace included at least 2 non-zero submissions. Based on number of segments in standard letter grading, we chose to use k-means to cluster the data into four clusters. Each cluster was averaged to extract a prototypical student. These clusters contained the following number of students: 2, 25, 4, and 3.

B. Student Action Graph

The graph produced by our analysis of the traces produced for the matrix assignment is shown in Figure 3. This graph was visualized using the tool yEd[2] and its hierarchical layout algorithm. Each node in this graph represents the state of a student's submission as assessed by the autograder. The specific node labels on the graph are for identification. Only node 0 (no work completed) and node 1023 (full score) have meaningful labels. In the visualization of the graph, each node has been shaded relative to the score that it was worth. The assessments that the nodes correspond to are described in Table III. This table shows the point scores for each state, and which methods are completed within it. A white node indicates a zero score, while a gray node indicates a full score. Each edge represents a transition between assignment states (that is: a student making progress on the assignment), and has been labeled with which features were completed, or broken (designated with a ˜).

TABLE III
NODES IN SAG FOR MATRIX ASSIGNMENT.

Node	Score	Completed
0	0	No progress.
128	1	getRows
192	2	getRows, getCols
256	2	getEle
257	7	getEle, toStr
288	5	getEle, scale
289	10	getEle, scale, toStr
313	16	getEle, scale, plus, minus, toStr
384	3	getEle, getRows
448	4	getEle, getRows, getCols
449	9	getEle, getRows, getCols, toStr
451	14	getEle, getRows, getCols, equals, toStr
480	7	getEle, getRows, getCols, scale
481	12	getEle, getRows, getCols, scale, toStr
482	12	getEle, getRows, getCols, scale, equals
507	23	All except constructor, multiply
511	28	All except constructor
961	13	Constructor, getEle, getRows, getCols, toStr
992	11	Constructor, getEle, getRows, getCols, scale
993	16	Constructor, getEle, getRows, getCols, scale, toStr
1016	17	All except multiply, equals, toString
1019	27	All except multiply
1023	32	All

The graph was produced by analyzing a set of 19 submission traces. Each trace included at least 8 non-zero submissions. Since there are 10 criteria that may be evaluated, there are $2^{10} = 1024$ possible submission states. However, not all unique states were seen in the traces. Initially, the graph contained 44 nodes and 118 edges. After simplification, this graph contained 23 nodes and 57 edges.

The final graph was also analyzed to determine two aspects which provide information about how students complete the

[2]https://www.yworks.com/products/yed

assignment: sink nodes and the most likely solution path were identified. Five of the nodes (961, 313, 992, 993, 1023) were sink nodes with no out-edges. Starting at source node (0), and exploring the graph by selecting edges in order of their edge weight found the following most likely (relative to collected data) series of states: 0, 128, 449, 451, 1019, 1023.

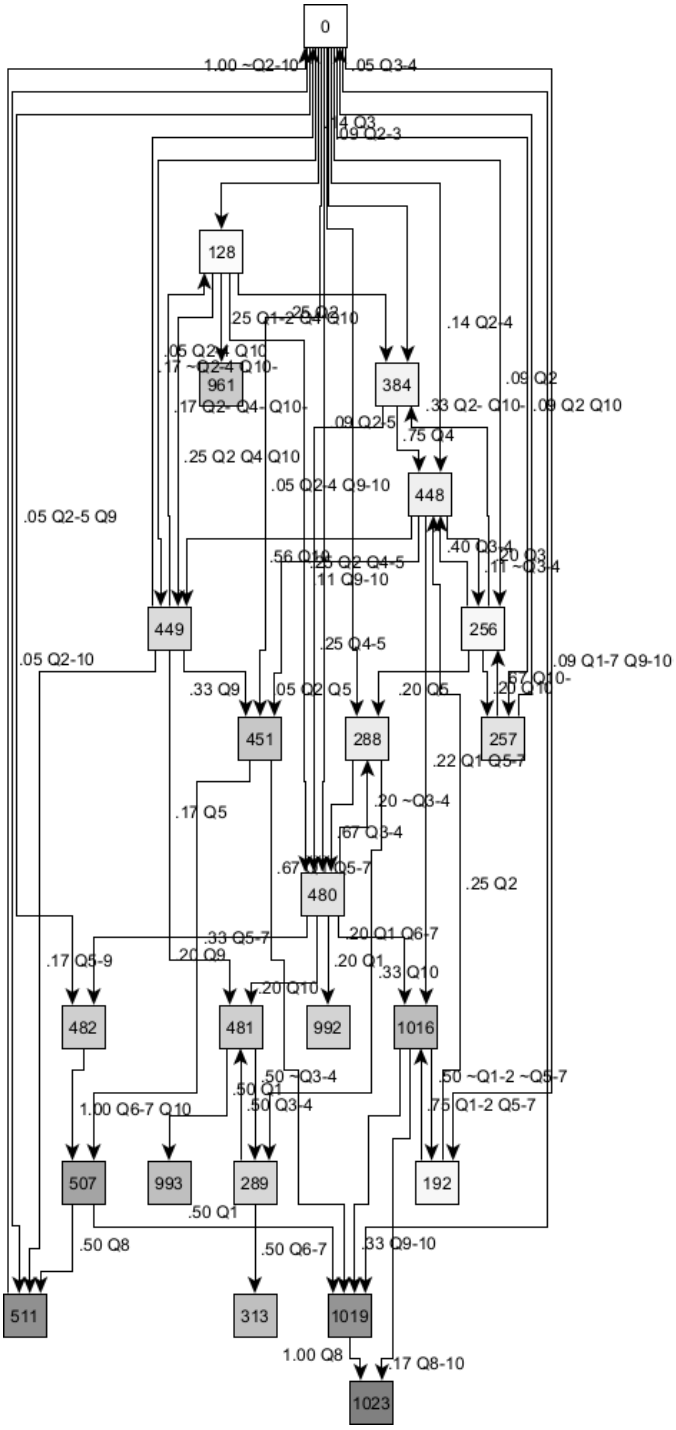

Fig. 3. Student action graph for matrix assignment.

V. Discussion

A. Fragility Scores

Based on the prototypical students inferred from the original data, we can identify different patterns in how students complete the assignment. Each of these patterns will indicate how students may have difficulty with the assignment as determined by completing and then accidentally breaking support for a feature (method).

- Prototype 1 (P1): These students had trouble with the constructor, and three of the other methods (getRows, getColumns, and scale). Since getRows and getColumns are often used by the other methods in the class, it is likely that this type of student was unable to attempt the more advance criteria (leading to a low FS).
- Prototype 2 (P2): These students had a relatively straightforward experience completing the assignment and did not cause regressions when implementing most methods. Only the constructor, scale, and toString caused an occasional regression.
- Prototype 3 (P3): These students were similar to those in P1. The difference is that they had trouble with getElement instead of the constructor.
- Prototype 4 (P4): These students had trouble with the majority of the assignment. For Q2 to Q10, multiple attempts were made, where criteria were supported and then broken. Although Q1 indicates a low FS, this may represent that it was never supported.

Interpreting the above results depends on the number of students in each cluster. In order: 2, 25, 4, and 3. In general, this indicates that the assignment does not have a particularly troublesome (fragile) rubric criteria since P2 represents students who smoothly completed the assignment, and it had the most students associated with it (24 of 34). P4 seems to indicate students may have trouble with math and/or programming in general. The combination of P1 and P2 suggest that there is a bottleneck in completing the assignment where students may have trouble with the basic accessors before moving on to the main matrix methods. Although the number of students in these prototypes is low, this suggests providing an intervention such as showing an example accessor in class or adding a note to the assignment about how they can be implemented.

B. Student Action Graph

Typical Solution Path: States in the graph are followed by a set of successor states that a student may visit as they continue development of their solution, with some weight indicating the relative likelihood that transition will be followed. By starting at the source node (0; the zero score state), and selecting edges based on weight, we can determine what path is most often used to generate a solution to the assignment. This path has several potential uses. Here, we will focus on determining if students are completing an assignment in a reasonable order. Although it depends on the specific aims of a course, we would expect that students tackle the different aspects of an assignment from easy to more difficult. If this is not the case,

it would indicate that students require additional direction on problem analysis and/or action planning. Another related use (not applicable to this assignment) would be to check that the order in which topics are taught in lecture align with the order in which students approach the assignment. This can help to ensure that students have the proper background to work on a problem, and to reduce situations where students spend extraneous effort on something that will be taught later.

For M1, this path was found to be: 0, 128, 449, 451, 1019, 1023. This indicates students start by implementing one of the simplest (and lowest weight) methods: getRows(). They then implement getElement, getRows, getColumns, and toString. The first three make up the simplest methods in the assignment. toString is more complicated but may have been prioritized for its value in debugging. Next, students implement equals. This method is more complicated but uses the nested loop structure of equals. Equals should also be a familiar method to students, which may have led to its selection before the new methods specific to the assignment. In the second to last step, students completed the more complex methods specific to this assignment: the constructor (requires immutability), scale, plus, minus. The latter three methods have almost identical structure (only the operator applied differs). Lastly, students implemented multiply, which is the most complicated operation.

For this assignment, students appear to be demonstrating a good approach to solving the assignment. They start with the simple accessor methods, then continue with methods that are already familiar, before moving onto more complicated methods, and finally to the most complicated method.

Sink Nodes: A sink node (meaning no out-edges) generally represents a student getting stuck on an assignment. For this assignment, five sinks were found: 313, 961, 992, 993, 1023. The last (1023) indicates the full credit state, where the student has completed the assignment. In general, a sink node suggests that students need to be provided additional help, whether it be information in the assignment such as a hint, or instruction in the classroom (a student may have understood the assignment but not had background knowledge to complete it).

The sink nodes in this assignment, suggest two types of issues: in state 313, the student was able to implement three of the basic matrix operations, but could not complete the simple accessor methods. This may indicate a student with math knowledge but not comfortable with object-oriented programming. The other three states (961, 992, and 993) are similar. Here, the student has implemented the constructor and the simple accessor methods. They may also have implemented scale (where the matrix is multiplied by a constant) or toString. The student was unable to proceed to the more complex matrix operations like add, minus, multiple, and equals.

VI. Conclusions

In this work, we have discussed the design of an automated assessment tool and applications of the activity trace that it captures. Although automatic assessment tools are often deployed to address issues in scaling a course, we aim to leverage them to improve student learning by understanding how students proceed through assignments. Using AA provides the opportunity to capture formative assessments performed on students, which are a rich source of information on a student's problem-solving process. We have described a simple model for rubric-based assessment, and its relation to an overall trace which follows a student's actions when completing an assignment. Due to the needs of our context, we have developed our own assessment tool for capturing these traces but the model is simple enough to be produced by other autograding tools. Based on our trace model, we have discussed two ideas for leveraging trace information: fragility scores (FS), and the Student Action Graph (SAG). FSs represent the difficulty of implementing part of an assignment as measured by the number of times a regression occurs (a feature is implemented and then later broken). The FSs for a rubric can be used to identify which parts of an assignment are difficult and need to be refined or suggest where additional instruction is required. Fragility scores can also be clustered to determine prototypical student experiences when completing an assignment, which can indicate specific bottlenecks that are encountered by students. A SAG represents the ways that students complete an assignment by tracking which assessment states a student passes through as they construct their solution. The graphical features of a SAG enable an instructor to evaluate student problem-solving process, leading to instructional changes when needed. It can also identify places where students get stuck in an assignment.

We have many avenues for future work, and now mention only a few: improvements to our existing approaches, and comparing different student cohorts. One improvement to our methods is to address issues with the large number of nodes that may be present in a SAG. In our current work, we address this through use of rubric groups, where we sum multiple criteria to create a simplified rubric. Another approach that may improve the computational tractability of generating a SAG would be to perform automatic dimensionality reduction. For instance, we would apply a technique like Multiple Correspondence Analysis [16]. Another SAG improvement is performing other graphical analyses, such as finding paths with low probability or transitions which lead to states worth fewer points. Our current work has focused on only one assignment within one class in one modality. We would like to expand our analyses to the other assignments in the class which are now autograded, and to the class in an online setting. In the far future, we would also look at constructing a status board system, which pulls live trace information and informs the instructor of the status of the class, and enables them to judge impacts of interventions they perform.

References

[1] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on novice programming: Strengths and limitations of existing systems," *Computer Applications in Engineering Education*, vol. 26, no. 6, pp. 2328–2341, 2018.

[2] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery, 2010, p. 86–93.

[3] V. Pieterse, "Automated assessment of programming assignments." Open Universiteit, Heerlen, 2013, p. 45–56.

[4] A. Singh, S. Karayev, K. Gutowski, and P. Abbeel, "Gradescope: A fast, flexible, and fair system for scalable assessment of handwritten work," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*. Association for Computing Machinery, 2017, p. 81–88.

[5] I. Mekterović, L. Brkić, B. Milašinović, and M. Baranović, "Building a comprehensive automated programming assessment system," *IEEE Access*, vol. 8, pp. 81 154–81 172, 2020.

[6] J. C. Caiza and J. M. Del Alamo, "Programming assignments automatic grading: review of tools and implementations," in *7th international technology, education and development conference (INTED2013)*, 2013, p. 5691.

[7] S. M. Arifi, I. N. Abdellah, A. Zahi, and R. Benabbou, "Automatic program assessment using static and dynamic analysis," in *2015 IEEE World Conference on Complex Systems (WCCS)*, 2015, pp. 1–6.

[8] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," vol. 19, no. 1, 2018. [Online]. Available: https://doi.org/10.1145/3231711

[9] R. Garcia, K. Falkner, and R. Vivian, "Systematic literature review: Self-regulated learning strategies using e-learning tools for computer science," *Computers & Education*, vol. 123, pp. 150–163, 2018. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1103970913

[10] R. S. Pettit, J. D. Homer, K. M. McMurry, N. Simone, and S. A. Mengel, "Are automated assessment tools helpful in programming courses?" in *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, 2015.

[11] E. Barra, S. Lopez-Pernas, A. Alonso, J. F. Sanchez-Rada, A. Gordillo, and J. Quemada, "Automated assessment in programming courses: A case study during the covid-19 era," *Sustainability*, vol. 12, no. 18, 2020. [Online]. Available: https://www.mdpi.com/2071-1050/12/18/7451

[12] C. Gordon, R. Lysecky, and F. Vahid, "The shift from static college textbooks to customizable content: A case study at zybooks," in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1–7.

[13] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[15] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[16] M. Greenacre and J. Blasius, Eds., *Multiple Correspondence Analysis and Related Methods*. Chapman and Hall/CRC, 2006.