2023 Annual Conference & Exposition
Baltimore Convention Center, MD | June 25 - 28, 2023

The Harbor of Engineering
Education for 130 Years

ASEE

Paper ID #39699

Auto-graded Scaffolding Exercises For Theoretical Computer Science

Prof. Jeff Erickson, University of Illinois Urbana-Champaign

Jeff Erickson is a Sohaib and Sara Abbasi Professor in the Department of Computer Science. He received an BS in computer science and mathematical sciences from Rice University, an MS in information and computer science from UC Irvine, and an PhD in computer science from UC Berkeley. His past research has focused on computational geometry, computational topology, graph algorithms, and their applications; he is also the author a popular free algorithms textbook. His awards include a Sloan Research Fellowship, an NSF CAREER award, and numerous teaching and research awards from the University of Illinois.

Jason Xia, University of Illinois at Urbana - Champaign
Eliot Wong Robson
Tue Do
Aidan Tzur Glickman
Zhuofan Jia
Eric Jin
Jiwon Lee
Patrick Lin
Steven Pan
Samuel Ruggerio
Tomoko Sakurayama, University of Illinois, Urbana-Champaign
Andrew Yin
Yael Gertner, University of Illinois, Urbana-Champaign

Dr Gertner joined the Computer Science Department at the University of Illinois in 2020 as a Teaching Assistant Professor. She received her B.S. and MEng in Electrical Engineering and Computer Science from MIT, and Ph.D. in Computer and Information Science at the University of Pennsylvania. She was a Beckman Fellow at the University of Illinois Urbana-Champaign. Her current focus is on broadening participation in Computer Science and Computer Science Education She has been developing materials and teaching for iCAN, a new program for broadening participation in CS for students who have a bachelor's degree in a field other than computer science.

Brad Solomon, University of Illinois, Urbana-Champaign

Auto-graded Scaffolding Exercises for Theoretical Computer Science

1 Introduction

This paper describes an ongoing effort to develop auto-graded scaffolding exercises for an upper-division theoretical computer science class, with a steady-state enrollment of about 400 students per semester, at the University of Illinois Urbana-Champaign. Our exercises are designed to reinforce the processes of designing, analyzing, and formally reasoning about computation. In particular, we aim to reward student progress toward correct solutions, both with partial credit and with narrative feedback, consistently with the rubrics used to grade written homeworks and exams. Our exercises are built within PrairieLearn,[25] a platform for auto-graded assessments already in heavy use both within the department and across campus.

Most of our auto-graded exercises are organized as *guided problem sets*. Each guided problem set consists of a small number of multi-stage exercises, implemented as a sequence of questions that guide students through the process of solving a design or proof question. Our guided problem sets support multiple correct solutions, detect common mistakes, automatically provide counterexamples for incorrect answers, provide helpful narrative feedback, and award partial credit consistent with grading rubrics for written homeworks and exams. We have also incorporated several new interactive tools that enable students to submit solutions similar to written homework. More than 2000 students have used our exercises since our development effort began in early 2021.

To provide some background, we describe both the organization, learning goals, and pedagogical challenges of our theoretical computer science class in Section 2, followed by a similar high-level overview of our scaffolding exercises in Section 3. In Section 4, we describe several of our new interactive elements. Section 5 shows a complete guided problems set aimed at the derivation of a dynamic programming algorithm. Finally, we directly surveyed the students in the Fall 2022 offering of the course about their experience working with both the new exercises and the traditional written homeworks; we report the results of our survey in Section 6.

2 Course Description

Our auto-graded exercises are designed to support an upper-division theoretical computer science class, CS 374, which covers a mixture of finite automata, formal languages, algorithm design and analysis, and computational complexity. The class is required for all computer science and computer engineering majors at Illinois, and almost all students in class are in one of these majors. Enrollment has been steady at 370 to 400 students for several semesters but has grown to almost

450 students in Spring 2023, thanks to a significant jump in admissions to both majors in Fall 2021. The course has a single lecture section that meets twice per week, plus several smaller lab sections, each with at most 40 students, that also meet twice per week.

In the taxonomy developed by Luu *et al.*,[14] CS 374 covers algorithm strategies (greedy, divide-and-conquer, backtracking, transform and conquer (reductions), dynamic programming), some fundamental algorithms (graph traversal and shortest paths), basic algorithm analysis (mostly as review), basic automata and complexity theory (finite-state machines, regular languages, and NP-completeness), and some more advanced algorithms (graphs). Almost all graded work consists of open-ended algorithm design and proof questions, in weekly written homework assignments and exams; there is no graded programming component. Coursework is weighted approximately as follows: 10% for auto-graded exercises, 20% for written homework, 20% each for two midterm exams, and 30% for a final exam. All instructors are active researchers in theoretical computer science. Complete archives of the Fall 2022 and Spring 2023 offerings of the course are publicly available.[2, 12]

2.1 Learning Goals

The primary learning goal of CS 374 is the ability to design and analyze algorithms, prove theorems about computation, and clearly and convincingly present these algorithms and proofs to a human audience. In particular, we want students to develop fluency with the process of solving problems. Lecture and lab activities, grading feedback, grading rubrics for partial credit, and our new scaffolding exercises focus on developing fluency and confidence in that process.

Homework and exam problems commonly have multiple correct solutions. Clearly presented correct solutions receive full credit even if those solutions are not anticipated by the instructor; partially correct solutions are scored relative to the closest correct solution.

Students are expected to answer most homework and exam questions using a mixture of semi-structured English and pseudocode, rather than code. While turning well-designed algorithms into practically efficient code is an important skill, that skill is already well-developed in prerequisite programming and data-structure classes. Instead, we want our students to focus on the abstract structure, correctness, and efficiency of computation, while staying deliberately agnostic about low-level implementation details or specific programming language syntax.

2.2 Challenges

The freeform nature of written homework and exam questions is a significant strength of our course, but it does come at a cost. There has been some promising work on grading simpler narrative questions using natural language processing,[11] but these tools require gathering and labeling significant training data, and they do not scale to larger blocks of text. In practice, machine grading is simply impossible for most work in CS 374. Instead, almost all work must be graded by human beings—mostly by graduate and undergraduate teaching assistants—and grading all submissions of a single problem, even with group homework submissions and detailed grading rubrics, takes significant time.

CS 374 also includes scaffolding activities in twice-weekly labs, which focus on relevant problem-solving processes, provide targeted feedback, and help students gain confidence in their mastery of the course materials, but even these activities address dozens of students at a time. The scale of the class makes it impossible to offer personal guidance and immediate feedback to every individual student.

A common complaint of students in many algorithms classes is a lack of *worked examples*. Every semester the instructors of CS 374 provide complete solutions and grading rubrics for over 100 problems from labs, homeworks, and exams, including at least one solved problem on every homework handout and solutions to a full set of practice exams, and we spend two hours every week solving problems in labs. Despite this deluge of worked examples, our students clamor for more, especially immediately before exams. Although these complaints are frustrating at first glance, we believe they have merit, because solutions are only the finished product; they offer little insight into the process for solving similar problems.

Finally, CS 374 is widely perceived as one of the most challenging courses in the undergraduate computer science and computer engineering curricula. Informal feedback in course evaluations reflects a common perception that success in theory classes is more a function of raw intelligence than hard work—"You either get it or you don't." or "You just have to be clever." This fixed mindset[8, 28] is likely exacerbated by the extreme competitiveness of admission to our computing majors. The perceived difficulty of the class, and the pervasive fixed mindset among many students, is a significant source of unnecessary anxiety.

3 Structure and Design Goals

As mentioned earlier, most of our auto-graded exercises are organized into "guided problem sets", each containing a series of exercises related to a single problem or skill. Guided problem sets are not intended to replace written homeworks or exams, but rather to replicate the kind of interactive leading questions that a student might be asked in a discussion/lab section or in office hours.

The design goals for these guided problem sets reflect the goals for other components of the course, including lectures, labs, and grading rubrics. First, for each type of problem, auto-graded exercises should reinforce the solution process recommended in other parts of the course for that problem type. Said differently, we want to proved the students with *working* examples, not just more *worked* examples. A good example here is the use of Proof Blocks,[19, 20, 21] which reinforce the recommended practice of writing proofs by successive refinement rather than line-by-line in order.

Another design goal is to support multiple correct solutions, and to recognize and reward progress toward any correct solution. For example, if we ask for a deterministic finite automaton (DFA) that accepts a certain language, we should reward full credit to *every* DFA that accepts that language. We also aim to provide meaningful feedback on partially correct solutions. For example, many of our exercises automatically provide explicit counterexamples for incorrect answers; for other problems, the grading code detects common mistakes and offers matching feedback.

Whenever possible, we avoid questions that invite blind exploration, especially multiple choice questions; we want solving the problems to be a learning process, not a process of elimination. Even with versioning and randomization, it is common to see students solve multiple-choice questions by submitting blindly-chosen answers to learn which answers are incorrect. We also aim to provide partial credit that rewards progress and targeted feedback that guides students toward correct solutions, instead of merely grading questions as correct or incorrect.

Finally, as a general rule, we also avoid free-form programming questions, in part because it is difficult to automatically grade code on any other basis than correctness on a finite set of test inputs. While transforming well-designed algorithms into practically efficient code is an important skill, that skill is not the focus of CS 374. We want students to focus instead on the structure, correctness, and efficiency of algorithms without worrying about (more strongly, while staying deliberately agnostic about) low-level implementation details or specific language syntax. These aims are consistent with the learning goals of the course.

4 Components

Many of our questions incorporate custom components intended to support the skills we want students to use on written homework and exams. We have either adapted or implemented highly interactive interfaces, while also providing useful targeted feedback through the grading code. We describe several of these custom components in the new few subsections.

4.1 Proof Blocks

We have adopted the Proof Blocks tool developed by Poulsen et al.,[19,20,21] which lets students assemble proofs by dragging and dropping individual proof steps into the correct order. We use Proof Blocks not only for proof questions, but also for developing simple algorithms, similar to Parsons programming problems[7,17] but at the level of pseudocode instead of executable code. For example, in one NP-hardness exercise, we ask students to build the pseudocode for a graph reduction using Proof Bocks. The grading code associates a small Python function with block, so in fact the student is indirectly assembling a Python program. The auto-grader either verifies that the reduction is correct or finds and displays a small counterexample, along with an explanation for why the example shows that the reduction is incorrect. This feedback is consistent with the "witness feedback" described by Bezáková et al.[1] Our course is the first anywhere to use Proof Blocks for a more advanced topic than introductory proofs.[18]

4.2 Scaffolded Writing

One of the most important skills we want students to master is clear communication of algorithmic ideas. To help develop this skill, we regularly ask students to include *landmark sentences* in their written homework and exam solutions that clearly and precisely describe their solution strategy. For example, in any dynamic programming solution, we ask students to include a precise English description of the recursive subproblems that their algorithm should solve. For example:

> Let $\mathit{MinCost}(i, j)$ denote the minimum cost of traveling from Hotel i to Hotel n using at most j coupons.

Similarly, in problems involving reductions, we ask students to include a precise statement of how the output of their reduction relates to the problem they are trying to solve or prove NP-hard; for example:

> It is possible to hike from the campground to the ranger station without being eaten by a bear, using at most k cans of bear repellent, if and only if there is a path from (s, k) to $(t, 0)$ in the graph G'.

> The output graph H contains a double-Hamiltonian tour if and only if the input graph G contains a Hamiltonian cycle.

We have developed a scaffolded-writing tool[27] that allows students to craft these landmark sentences and receive feedback and credit reflecting how close they are to correct. Rather than accepting free-form English input, which would be impossible to auto-grade, students write their solutions by choosing from a dynamic list of tokens, using an interface similar to the auto-complete function in most text-messaging apps. The list of visible tokens is controlled by a context-free grammar defined by the instructor. Clicking on a token adds the content of that token to the end of the response, and updates the list of available tokens; thus, the student interactively chooses a string generated by the underlying grammar. When a student submits, the tool passes a parse tree of the submission to the autograder, which then assigns partial or full credit and provides feedback based on specific features that are present or absent in the parse tree. Instructors can design grammars that permit both multiple correct answers and incorrect answers that reflect common mistakes, along with feedback specifically tailored to those common mistakes.

We describe a specific application of this in Section 5. Xia and Zilles[27] describe the scaffolded writing tool in more detail and describe an experimental evaluation in a large undergraduate algorithms class.

4.3 Automata Builder

One of the skills we teach early in our algorithms class is designing and drawing deterministic and non-deterministic automata. Online tools for drawing and simulating automata (also known as finite state machines or FSMs), such as JFLAP[22] and Automata Tutor,[3] have existed for many years. While these tools are capable, we could not integrate them into our assessment platform. Instead, we adapted a lightweight open-source browser-based FSM editor[24] for the front-end interface and an open-source `automata` Python library[10] for the back-end grading code. We have extended the `automata` library to meet our needs, and those extensions have since been incorporated back into the original open-source project.

Our automata editor provides complete freedom to draw, label, and edit states and transitions, including the start state and accepting states. For deterministic automata, students can declare a hidden dump/trash state to simplify their design. When the student submits, the grading code compares the language accepted by the submitted FSM to the target language, and automatically provides counterexamples if the submitted machine is incorrect. Question authors only need to specify the desired type of automaton (deterministic or nondeterministic), a maximum state limit, the input alphabet, and a formal description of one correct automaton. We emphasize that scores and feedback are based on *the language accepted by* the student's submission; every correct FSM is graded as such.

At the time of writing, the only feedback we provide are improperly classified strings, with no partial credit. Specifically, if the submitted FSM incorrectly accepts or rejects any string of length 8 or less, we report all such strings; otherwise, we automatically report one counterexample of minimum length. Again, this feedback is consistent with the "witness feedback" described by Bezáková et al.[1] We are considering incorporating other types of feedback and partial credit[4] into our tool in the future.

4.4 Big-O Evaluation

As a smaller but important tool, we created a new element to evaluate expressions that use asymptotic ("Big O") notation, most commonly in reporting the running times of algorithms. Our element symbolically compares student input to a reference solution using the SymPy Python library[15] and then provides feedback and partial credit targeted to common errors, such as upper bounds that are too small and therefore incorrect, upper bounds that are correct but loose, and expressions with unnecessary constant factors or lower-order terms. The element properly supports $O(\,)$, $\Theta(\,)$, $\Omega(\,)$, $o(\,)$, and $\omega(\,)$ expressions,[13] providing the necessary feedback and partial credit for each expression type. The question writer only has to describe a reference solution; the element automatically handles all grading and feedback. In addition to our target theory course, our "big-O" element has already been incorporated into four other computer science courses offered by our department.

5 Example Guided Problem Set

In this section, we walk through a typical guided problem sets, which leads students through the design of a dynamic programming algorithm. Dynamic programming is a core technique in algorithm design; it is also widely recognized as one of the most challenging topics in any algorithms course.[9,14,23,29] Developing a dynamic programming algorithm typically involves three distinct stages:

1. **Recursive structure:** Identify an appropriate recursive structure in the given problem. This requires identifying both (a) the subset of input data that each recursive subproblem needs to consider and (b) how the output of that recursive subproblem depends on that subset of input data.
2. **Recursive solution:** Write a mathematical recurrence or a recursive backtracking algorithm to solve the problem described in step 1.
3. **Iterative details:** Describe how to iteratively evaluate all recursive subproblems without repetition, by considering them in the correct order, starting with the base cases of the recurrence from step 2 and working up to the desired solution. This requires (a) identifying an appropriate data structure, (b) identifying an appropriate evaluation order, and (c) analyzing the running time of the resulting iterative algorithm.

5.1 The Problem

Most of the example guided problem set focuses on the following problem, taken directly from a 2016 final exam. This problem is intended to be straightforward for most students by the end of the semester, even under exam conditions and time pressure.

After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every trial he tries exhausts him. The bigger the trial, the longer he must rest before he is well enough to take the next trial. (Of course, he is willing to wait for it.) If a trial begins while Burr is resting, Hamilton snatches it up instead.

Burr has been asked to consider a sequence of n upcoming trial. He quickly computes two arrays $\mathit{profit}[1\,..\,n]$ and $\mathit{skip}[1\,..\,n]$, where for each index i,

- $\mathit{profit}[i]$ is the amount of money Burr would make by taking the ith trial, and
- $\mathit{skip}[i]$ is the number of consecutive cases Burr must skip if he accepts the ith trial. That is, if Burr accepts the ith trial, he cannot accept trial $i+1$ through $i+\mathit{skip}[i]$.

Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n trials.

Here is one complete solution for this problem, in the exact form we would distribute to students for a written homework or exam problem.

Solution [dynamic programming]: Let $\mathit{MaxProfit}(i)$ denote the maximum profit that Burr can earn only from cases i through n.

The problem asks us to compute $\mathit{MaxProfit}(1)$.

The *MaxProfit* function obeys the following recurrence:

$$\mathit{MaxProfit}(i) = \begin{cases} 0 & \text{if } i > n \\ \max \left\{ \begin{array}{c} \mathit{MaxProfit}(i+1) \\ \mathit{profit}[i] + \mathit{MaxProfit}(i + \mathit{skip}[i] + 1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $\mathit{MaxProfit}[1\,..\,n]$, which we can fill from right to left (decreasing i) in $O(n)$ time.

The instructor provides both students and graders with a detailed rubric for grading all dynamic programming problems. At a high level, a clear and correct English specification of a recursive function is worth 30%, a correct recursive solution is worth 40%, and the remaining iterative details are worth 30%. Students have the option of submitting iterative pseudocode *instead* of a recursive solution and an evaluation order, but they must still include an English specification of the subproblems. Most dynamic programming problems (including this one!) have multiple correct solutions; students receive full credit for any correct algorithm whose running time matches the official solution, partial credit for slower correct algorithms, and extra credit for faster correct algorithms.

5.2 The Guided Problem Set

Our example guided problem set consists of multiple stages, each addressing a different portion of the algorithm-design process.

The first stage uses our scaffolded writing tool to ask the student for an English description of a recursive problem that could be used to solve Burr's problem. This stage provides targeted feedback (including partial credit) for errors that students commonly make in writing these specifications in written homeworks and exams, including not describing a function at all, not clearly describing the output of the function, not clearly describing how that output depends on the input parameters, describing a function that cannot actually be evaluated recursively, or describing a function that does not solve the stated problem. We list here a few representative student responses (with problematic portions underlined) and the corresponding feedback from our autograder; we refer the reader to Xia and Zilles[27] for further discussion.

- **Response:** Define the subproblem to be the profit that can be obtained from the rest of the trials.
  Feedback: *Your subproblem definition should declare a function with input parameters that can be memoized.*

- **Response:** Define *DP*(i) to be the maximum profit that can be obtained from trials i through n.
  Feedback: *Please choose a descriptive function name that accurately represents what the function outputs.*

- **Response:** Define *MaxProfit*(i) to be the maximum profit that can be obtained for i.
  Feedback: *Your subproblem definition mentions the function's input parameters, but it does not clearly explain how these input parameters affect the output of the function. Can you be more specific about what the function parameters represent in the context of your subproblem?*

- **Response:** Define *MaxProfit*(i) to be the answer that can be obtained from trials i through n.
  Feedback: *Please be more precise about what quantity the function actually outputs. Just saying "answer" is too vague.*

- **Response:** Define *MaxProfit*(i) to be the maximum profit that can be obtained from the rest of the trials.
  Feedback: *Your function takes i as an input parameter, but your subproblem definition does not explain how this parameter affects the output of the function.*

- **Response:** Define *MaxProfit*(i) to be the maximum profit that can be obtained from trials 1 through i.
  Feedback: *Using prefix subproblems actually doesn't work for this problem, because these prefix subproblems don't reduce recursively. Think about why this is the case and why suffix subproblems don't have the same issue.*[1]

- **Response:** Define *MaxProfit*(i) to be the maximum profit that can be obtained from trials i through n.

[1] Defining *MaxProfit*(i) to be the maximum profit that can be obtained from trials 1 through *i if trial i must be accepted* would be correct.

Feedback: *Correct! Well done!*

The second stage starts the development of a suitable recurrence for $\mathit{MaxProfit}(i)$, by considering the most general recursive case $i < n$. The problem statement first observes that when Burr reaches the ith trial, he has exactly two choices—either take the trial or don't. Then for each choice, the student is asked to choose which other trials Burr can still take from a randomized short list of options:

- only trial $i + 1$
- only trial $\mathit{skip}[i]$
- only trial $i + \mathit{skip}[i]$
- only trial $i + \mathit{skip}[i] + 1$
- any of the trails i through n
- any of the trails $i + 1$ through n — *(correct when Burr refuses trial i)*
- any of the trials $\mathit{skip}[i]$ through n
- any of the trials $\mathit{skip}[i] + 1$ through n" — *(correct when Burr takes trial i)*

The third stage asks the student to write symbolic expressions for $\mathit{MaxProfit}(i)$, first considering Burr's choices separately ($\mathit{MaxProfit}(i + 1)$ and $\mathit{profit}[i] + \mathit{MaxProfit}(i + \mathit{skip}[i] + 1)$) and then combining them into a single expression

$$\max(\mathit{MaxProfit}(i + 1), \mathit{profit}[i] + \mathit{MaxProfit}(i + \mathit{skip}[i] + 1))$$

The grader uses using the SymPy Python library[15] to compare student input to the expected answer, so even answers like $\mathit{profit}[2 * i/2] + \mathit{MaxProfit}(5 + i + \mathit{skip}[i] - 4) + 0$ are accepted as correct.

The last required stage asks for a suitable iterative evaluation order and the resulting running time. For this example problem, the only reasonable evaluation orders are "increasing" and "decreasing", so we use a simple multiple-choice question. (For more complex dynamic programming problems that involve multiple nested loops, we ask students with several options and ask for each one which dependencies if any each order violates.) We use our custom big-O tool to evaluate the student's submitted running time and provide informative feedback.

Finally, we provide an optional Python programming exercise, which invites students to implement their iterative dynamic programming recurrence. We avoid free-form programming exercises as a general rule, primarily because they are difficult to auto-grade on any metric other than correctness on a finite set of inputs; however, many students do find them useful to build intuition. We provide a custom memoization data structure that keeps track of the bounds the student defines, the order that the code fills the memoization array, any elements that were never filled, and the final output. This structure provides targeted feedback for common errors, for example, trying to read subproblem values that are either outside the declared array bounds or that have not yet been computed.

6 Student Survey

We designed the auto-graded scaffolding exercises to support students in learning the course objectives. So that students would be motivated to engage with these exercises throughout the course, we designed them to be easy and enjoyable to use. We also focused on making sure the content of these exercises closely matched the required content of the class, so students felt that that these exercises are valuable to improving their competency in the class. All three of these factors—ease of use, enjoyability of exercises, and a clear connection to increased competency—are correlated with improving motivation.[5,6,26]

The exercises themselves are designed to increase students' confidence and ability to design and analyze algorithms which are the core learning goals of the course. We anticipated that the exercises will increase students ability to better understand the process of designing and analyzing algorithms. To evaluate the success of these new exercises, we directly surveyed students in the Fall 2022 offering of CS 374 about their experience working with both the new exercises and the traditional written homeworks.

We administered the survey at the last week of class, after they had ample engagement with the exercises. Our survey asked the students to express their agreement to 28 statements on a 5-point Likert scale ("strongly disagree", "somewhat disagree", "neither agree nor disagree", "somewhat agree", and "strongly disagree"). The first 14 statements were about the new auto-graded scaffolding exercises; the remaining 14 were exactly the same statements but about the traditional written homeworks. We anticipated that students would be more likely to agree with being motivated, confident, and perceive greater understanding with regard to the auto-graded scaffolding exercises, as compared with the written homeworks.

6.1 Survey Design

We measured students' intrinsic motivation by modifying questions taken from the Intrinsic Motivation Inventory.[5,6] Specifically, we examined their quantitative views and expectations in five broad sub-categories: Interest/Enjoyment, Perceived Competence, Effort/Importance, Value/Usefulness, and Pressure/Tension. We also designed questions that would specifically measure our goals to increase students' understanding of algorithm design and analysis.

The survey asked students to rate their level of agreement with 14 statements listed below, first about the guided problem sets, and then about the written homeworks. Statement 1 measures interest/enjoyment; statements 12, 9, 13, and 6 measure different aspects of value and usefulness; statements 4 and 14 measure effort/importance; and statement 5 examines the pressure/tension introduced by the new set of exercises. Statements 10, 11, 2, 8, and 7 are designed to address our specific learning goals.

1. I enjoyed doing them very much.
2. Completing them made me more confident in my ability to solve problems.
3. They made me realize that I can be good at this.
4. It was important to me to do well at this task.

5. I was anxious while working on this task.
6. I would be willing to do this again because it has some value to me.
7. I think doing them could help me to do better on the exam.
8. After completing them, I have a better understanding of the material.
9. They made it easier to learn the material.
10. They helped me break down the process into clear steps.
11. They helped me know where to begin to design an algorithm.
12. They were easy to use.
13. They were a good use of my time.
14. I spent a lot of time on them.

6.2 Results

We administered our IRB-approved survey to students during the last week of the course. A total of 301 students filled out the survey, of which 263 students consented to participate in the research study. (Students were offered extra credit as compensation for filling out the survey, even if they did not consent to participate in the study.) Responses from three consenting students were dropped because they did not complete the first half of the survey, leaving a total of 260 complete responses.

The results of our Likert-scale survey questions are shown in Figure 1. For each statement, the response distribution for the auto-graded exercises is shown directly above the response distribution for written homework. We used Wilcoxon rank-sum test to compare these distributions for each statement. For almost all statements, there was a statistically significant difference between the response distributions ($|z| > 10$, $p < 10^{-5}$). For statement 8 ("After completing them, I have a better understanding of the material."), the difference between distributions was smaller but still statistically significant ($z \approx 2.96$, $p \approx 0.003$). For the last two statements 7 and 4 ("I think doing them could help me to do better on the exam" and "It was important to me to do well at this task"), there was no significant difference between the response distributions ($|z| < 0.4$, $p > 0.5$).

6.3 Discussion

Our intended goals when producing these new exercises were to support student learning through more engaging and valuable exercises. On the engagement front, our preliminary results are fairly unanimous. Students indicated a strong preference for the auto-graded exercises on all 'enjoyment' and 'effort' statements. Likewise, there was a large reduction in self-reported 'anxiousness' when compared to written questions. This is a great sign for the continued development and expansion of these tools to other classes, concepts, and material.

Our preliminary findings for student's opinions on the 'value' of these questions and their 'competency' after completing them is less clear. Most of the related survey questions did demonstrate that students' confidence and perceived understanding strongly favored the scaffolded

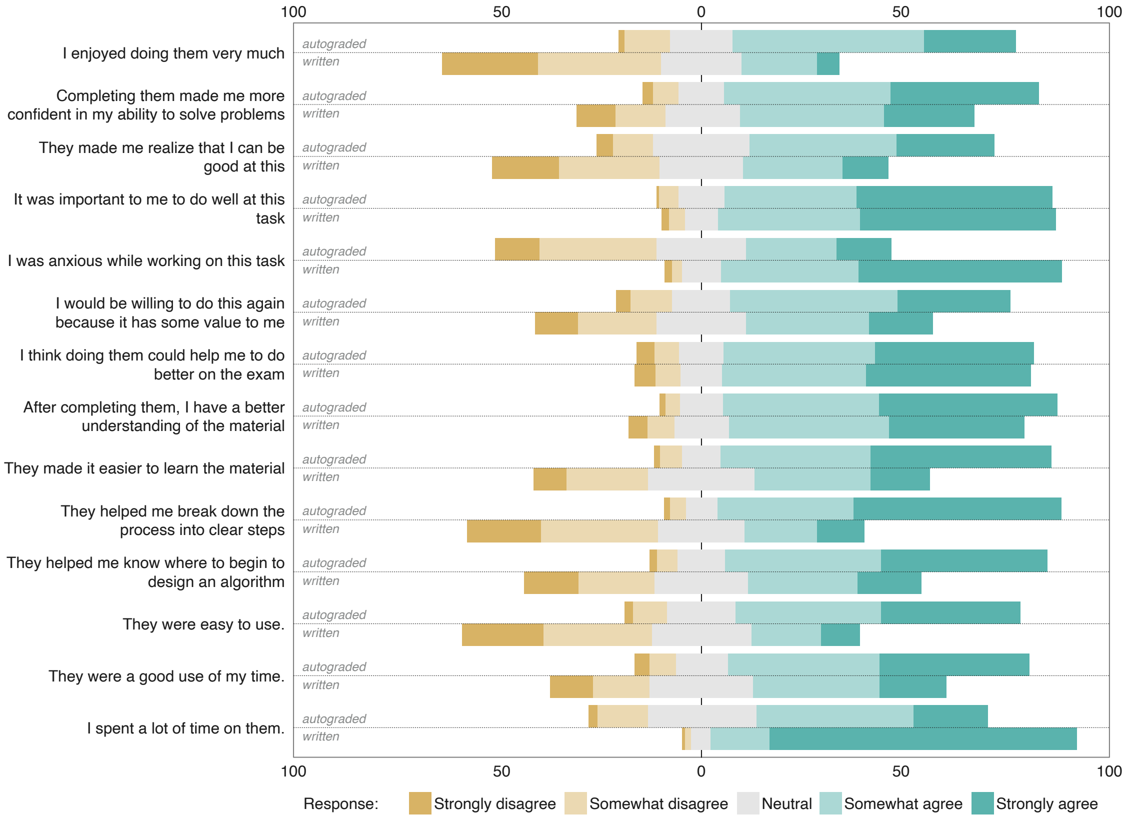

Figure 1: Results of our survey of 260 students. For each statement, responses for auto-graded exercises are shown immediately above responses for written homework.

exercises over written questions, but when asked directly about perceived impact on future exam performance, there was no real distinction.

To try to distinguish if there is a relationship between student's responses (i.e. their perceived competency) and their actual evaluation in the class, we stratified student responses based on their grade for each midterm and final individually. This generally did not affect our observations; all statements yielded equally significant responses for both groups, consistent with the unstratified analysis shown in Figure 1, with the exception of statement 8 ("After completing them, I have a better understanding of the material"). Here the top 50% of the class did not distinguish between the auto-graded and written exercises, but the bottom 50% were strong proponents of the new system.

As we continue to develop more material using these new tools, we hope to further evaluate the relationship between student impressions and self-reported motivation against their actual performance in the class.

References

[1] I. Bezáková, K. Fluet, E. Hemaspaandra, H. Miller, and D. E. Narváez, "Witness feedback for introductory CS theory assignments," in *Proc. 52nd ACM Technical Symposium on Computer*

Science Education (SIGCSE), 2021, p. 1300.

[2] C. Chekuri *et al.*, “CS/ECE 374: Algorithms & Models of Computation,” University of Illinois Urbana-Champaign, Spring 2023. [Online]. Available: https://courses.engr.illinois.edu/cs374al1/sp2023/

[3] L. D’Antoni, M. Helfrich, J. Kretinsky, E. Ramneantu, and M. Weininger, “Automata tutor v3,” in *Proc. 32nd International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, no. 12225. Springer, 2020, pp. 3–14.

[4] L. D’Antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann, “How can automatic feedback help students construct automata?” *ACM Trans. Comput.-Hum. Interact.*, vol. 22, no. 2, pp. 9:1–9:24, 2015.

[5] E. L. Deci, H. Eghrari, B. C. Patrick, and D. R. Leone, “Facilitating internalization: The self determination theory perspective,” *Journal of Personality*, vol. 62, pp. 119–142, 1994.

[6] E. L. Deci and R. Ryan, “Self-determination theory,” in *Handbook of Theories of Social Psychology*, P. A. M. van Lange, A. W. Kruglanski, and E. T. Higgins, Eds. Sage Publications Ltd., 2012, vol. 1, ch. 20, pp. 416–436.

[7] Y. Du, A. Luxton-Reilly, and P. Denny, “A review of research on Parsons problems,” in *Proc. 22nd Australasian Computing Education Conference (ACE)*, 2020, pp. 195–202.

[8] C. S. Dweck, *Mindset: The New Psychology of Success*. Random House, 2006.

[9] E. Enström and V. Kann, “Iteratively intervening with the “most difficult” topics of an algorithms and complexity course,” *ACM Trans. Comput. Educ.*, vol. 17, no. 1, pp. 4:1–4:38, 2017.

[10] C. Evans, “Automata,” Github repository, 2022. [Online]. Available: https://github.com/caleb531/automata

[11] M. Fowler, B. Chen, S. Azad, M. West, and C. Zilles, “Autograding ‘Explain in plain English’ questions using NLP,” in *Proc. 52nd ACM Technical Symposium on Computer Science Education (SIGSCE)*, 2021, pp. 1163–1169.

[12] S. Har-Peled *et al.*, “CS/ECE 374: Algorithms & Models of Computation,” University of Illinois Urbana-Champaign, Fall 2022. [Online]. Available: https://courses.engr.illinois.edu/cs374/fa2022/a/

[13] D. E. Knuth, “Big omicron and big omega and big theta,” *SIGACT News*, vol. 8, no. 2, pp. 18–24, 1976.

[14] M. Luu, M. Ferland, V. N. Rao, A. Arora, R. Huynh, F. Reiber, J. Wong-Ma, and M. Shindler, “What is an algorithms course? Survey results of introductory undergraduate algorithms courses in the U.S.” in *Proc. 54th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2023, pp. 284–290.

[15] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka,

A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "SymPy: Symbolic computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.

[16] D. Parsons and P. Haden, "Parson's programming puzzles: A fun and effective learning tool for first programming courses," in *Proc. 8th Australasian Conference on Computing Education (ACE)*, vol. 52, 2006, pp. 157–163.

[17] S. Poulsen, "Personal communication," February 2023.

[18] S. Poulsen, S. Kulkarni, G. Herman, and M. West, "Efficient partial credit grading of proof blocks problems," in *Proc. 24th International Conference on Artificial Intelligence in Education (AIED)*, ser. Lecture Notes in Computer Science. Springer, 2023, to appear.

[19] S. Poulsen, M. Viswanathan, G. L. Herman, and M. West, "Evaluating proof blocks problems as exam questions," in *Proc. 17th ACM Conference on International Computing Education Research (ICER)*, 2021, pp. 157–168.

[20] ——, "Proof blocks: Autogradable scaffolding activities for learning to write proofs," in *Proc. 27th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '22)*, vol. 1, 2022, pp. 428–434.

[21] S. H. Rodger and T. W. Finley, *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett, 2006.

[22] M. Shindler, N. Pinpin, M. Markovic, F. Reiber, J. H. Kim, G. P. N. Carlos, M. Dogucu, M. Hong, M. Luu, B. Anderson, A. Cote, M. Ferland, P. Jain, T. LaBonte, L. Mathur, R. Moreno, and R. Sakuma, "Student misconceptions of dynamic programming: A replication study," *Comput. Sci. Educ.*, vol. 32, no. 3, pp. 288–312, 2022.

[23] E. Wallace, "Finite state machine designer," Github repository, 2015. [Online]. Available: https://github.com/evanw/fsm

[24] M. West, G. L. Herman, and C. Zilles, "PrairieLearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning," in *Proc. 2015 ASEE Annual Conference & Exposition*, 2015, pp. 26.1238.1–26.1238.14.

[25] A. Wigfield and J. S. Eccles, "Expectancy–value theory of achievement motivation," *Contemporary Educational Psychology*, vol. 25, no. 1, pp. 68–81, 2000.

[26] J. Xia and C. Zilles, "Using context-free grammars to scaffold and automate feedback in precise mathematical writing," in *Proc. 54th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2023, pp. 479–485.

[27] D. S. Yeager and C. S. Dweck, "Mindsets that promote resilience: When students believe that personal characteristics can be developed," *Educational Psychologist*, vol. 47, no. 4, pp. 302–314, 2012.

[28] S. Zehra, A. Ramanathan, L. Y. Zhang, and D. Zingaro, "Student misconceptions of dynamic programming," in *Proc. 49th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2018, pp. 556–561.