Using Context-Free Grammars to Scaffold and Automate Feedback in Precise Mathematical Writing

Jason Xia
University of Illinois
Urbana, IL, USA
jasonx3@illinois.edu

Craig Zilles
University of Illinois
Urbana, IL, USA
zilles@illinois.edu

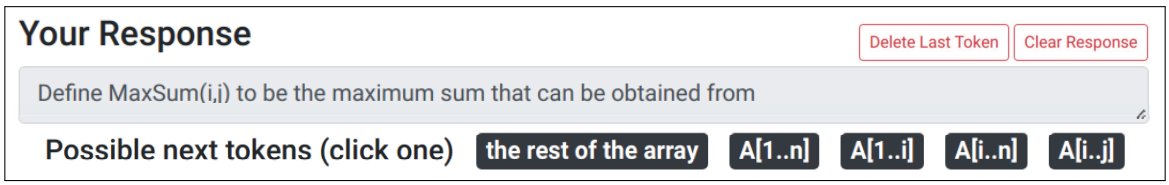

Figure 1: An example question that uses our scaffolded writing tool. In this screenshot, the student has already constructed part of their response (shown in the grey box), and they can append to their response by clicking one of the possible next tokens (shown in the black boxes). The set of possible next tokens is generated based on an instructor-defined context-free grammar. Students' responses are restricted to sentences that can be generated by this grammar. After the student completes and submits their response, the autograder uses the grammar to parse the response and provide instantaneous feedback.

ABSTRACT

In technical writing, certain statements must be written very carefully in order to clearly and precisely communicate an idea. Students are often asked to write these statements in response to an open-ended prompt, making them difficult to autograde with traditional methods. We present what we believe to be a novel approach for autograding these statements by restricting students' submissions to a pre-defined context-free grammar (configured by the instructor). In addition, our tool provides instantaneous feedback that helps students improve their writing, and it scaffolds the process of constructing a statement by reducing the number of choices students have to make compared to free-form writing. We evaluated our tool by deploying it on an assignment in an undergraduate algorithms course. The assignment contained five questions that used the tool, preceded by a pre-test and followed by a post-test. We observed a statistically significant improvement from the pre-test to the post-test, with the mean score increasing from 7.2/12 to 9.2/12.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Theory of computation** → *Dynamic programming*.

KEYWORDS

CS education, scaffolded writing, automatic grading, context-free grammars, dynamic programming

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569728

ACM Reference Format:
Jason Xia and Craig Zilles. 2023. Using Context-Free Grammars to Scaffold and Automate Feedback in Precise Mathematical Writing. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569728

1 INTRODUCTION

In computer science, large course enrollments are fueling a shift towards more autograded assignments. This pressure could lead instructors to design assignments based on what's easy to grade automatically and eschew pedagogically valuable questions that ask students to respond in English. Our perceptions of this trend led us to develop an automated system that can evaluate and give useful feedback on English statements written by students.

One common technique used in algorithms problems is *dynamic programming*. This is a technique that involves defining smaller subproblems that have a recursive dependency on each other and using these subproblems to build up to the final answer. In a dynamic programming solution, students are expected to identify and describe these recursive subproblems in coherent and precise English [1]. Formulating these descriptions clearly and precisely is a crucial step of the problem-solving process that clarifies the rest of the solution [3]. However, automatically grading this type of free-form writing is beyond the state of the art, and manual grading does not provide a quick feedback loop to facilitate improvement.

With this in mind, we set out to create a tool that can give feedback focused on the English description of subproblems in a dynamic programming solution. Using our tool, an instructor can define a context-free grammar (CFG) and require that students' submissions be contained in the CFG's language. The tool enforces this by providing a list of tokens to choose from instead of giving students complete freedom to type anything (see Figure 1). The list of options only contains tokens that are allowed to come next based on the CFG. This means that the list of options updates dynamically

as the response is being constructed. Clicking on a token causes it to be appended to the end of the response and causes the list of options to update.

When a student submits, the tool generates a parse tree of the student's submission based on the CFG provided by the instructor. Then, an autograder (written by the instructor) checks for specific features in the parse tree and assigns scores/feedback based on whether or not these features are present.

In addition to providing instantaneous feedback, another benefit of our tool is that it provides scaffolding by removing degrees of freedom. By fixing the overall structure of the statement, we can focus students' attention towards a few important choices that still need to be made without overwhelming them with an infinite space of choices. The goal is that, with enough practice, students will be able to write these English statements without the scaffolding provided by a CFG.

In this paper, we describe the initial implementation of the scaffolded writing tool, and we conduct a research study to determine whether the tool can help students improve their free-form writing once the scaffolding is removed.

2 RELATED WORK

We will provide a brief overview of related work in this area by discussing other approaches to autograding natural language responses in computer science and discrete math.

2.1 Statistical NLP Autograders

The problem we are addressing is an instance of automatic short answer grading (ASAG). Currently, all competitive ASAG techniques are based on machine learning [6], often involving statistical natural language processing (NLP). For example, Rus et al. (2013) built conversational intelligent tutoring systems using a statistical NLP approach that takes an expert answer and measures the semantic similarity to the student answer by modeling it as a quadratic assignment problem [13]. In CS, Fowler et al. created an autograder for "Explain in Plain English" questions based on a logistic regression model with bag-of-words and bigram features [5].

The upside of these NLP approaches is that they allow the student complete freedom in constructing their response, which most closely mimics what the student will have to do in the context of a summative evaluation.

Three downsides, however, are notable. First, they are not 100% accurate; for example, Fowler et al.'s model only achieved an accuracy of 87-89% [5]. Second, these statistical NLP approaches require a training data set, which involves a significant amount of manual grading and labeling. Third, these existing systems don't generate useful feedback beyond whether or not the answer is correct. Doing so would require designing/training additional models.

2.2 Parsons Problems and Proof Blocks

Parsons problems are a scaffolded exercise where students are asked to assemble prewritten lines of code into a correct program [9]. Another more recent tool inspired by Parsons problems is Proof Blocks, in which students drag and drop prewritten lines of a proof into a complete proof [10].

Parsons problems and Proof Blocks both provide scaffolding by taking a free-form writing task (e.g., writing a proof or program from scratch) and removing degrees of freedom from that task. Research on Parsons problems has shown that this scaffolding significantly accelerates the learning process for beginners [4]. Furthermore, in both Parsons problems and Proof Blocks, the scaffolding reduces the space of possible answers, which makes them more straightforward to grade compared to their free-form counterparts [2, 10]. Because our tool also provides scaffolding by removing degrees of freedom from a free-form writing task, we believe that the benefits discussed above should apply to our tool as well.

An important difference between our tool and these existing tools is that our tool doesn't provide students with a static bank of options; instead, it presents a context-dependent list of options that changes as the student composes their answer. We believe that this context-dependent behavior is better suited to our application than to Parsons problems or Proof Blocks. If we presented a static bank of options in our tool, students would need to filter out options which don't grammatically make sense, which is not part of the skill that the tool is trying to teach. With context-dependent behavior, students would be presented with a list of options that fit into the sentence grammatically, and they could focus all of their attention on analyzing which option contributes the correct *meaning* to the sentence. On the other hand, for writing code or proofs, determining which options are "grammatically" allowed to come next (i.e., which lines are syntactically or logically valid) is actually part of the skill that the tool is trying to teach. Thus, in these situations, it makes sense to present all of the options regardless of context.

3 IMPLEMENTATION

To explain the implementation of the tool, we will use the problem in Figure 2a as our running example. The solution to this problem is the subproblem definition shown in Figure 2b. A review of subproblem definition construction can be found online [17].

Our tool requires the design of a context-free grammar (see Figure 2c) that can generate the correct answer and also generate compelling distractors. This is a manual, creative process that benefits from an understanding of common student mistakes (which can be used as inspiration for distractors).

Now, there are two main tasks that our tool needs to accomplish using this CFG, which we will describe in the following subsections.

3.1 Generating Possible Next Tokens

The problem that we need to solve in this section can be formally described as follows:

> Given a context-free grammar (CFG) G and a list of terminals $t_1, t_2, \ldots, t_i$ that is a prefix of some sentence produced by G, find all terminals[1] t_{i+1} such that $t_1, t_2, \ldots, t_i, t_{i+1}$ is still the prefix of some sentence produced by G.

We use a standard algorithm [14] to convert our CFG G to a *non-deterministic pushdown automaton* (PDA). Then, we simulate this PDA reading the terminals $t_1, \ldots, t_i$, which potentially creates

[1]Note that for our purposes, "terminal" is synonymous with "token".

a You are planning a road trip along a highway with n evenly-spaced hotels. These hotels have varying costs; the costs of staying overnight at each of the hotels are provided in the array $HotelCosts\ [1..n]$, where $HotelCosts[i]$ is the cost of Hotel i. Each day, you can either travel to the next hotel, or you can skip a hotel and travel forward by two hotels. Each night, you must stay at a hotel. Furthermore, you have k coupons that allow you to stay at a hotel for free. Describe a dynamic programming algorithm to determine the minimum possible cost of traveling from Hotel 1 to Hotel n.

b Define $MinCost(i, j)$ to be the minimum cost of traveling from Hotel i to Hotel n using at most j coupons.

c **SENTENCE** -> "Define" FUNCTION_DECLARATION "to be the" FUNCTION_OUTPUT "."
FUNCTION_DECLARATION -> "the subproblem" | "DP(i)" | "DP(i, j)" | "MinCost(i)" | "MinCost(i, j)"
FUNCTION_OUTPUT -> EXTREMAL_ADJ NOUN SITUATION
EXTREMAL_ADJ -> EPSILON | "minimum" | "maximum"
NOUN -> "answer" | "cost" | "hotels" | "coupons"
SITUATION -> MENTION_PARAMS_WITHOUT_EXPLAINING | "of" SUBARRAY_RESTRICTION ADDITIONAL_RESTRICTION
MENTION_PARAMS_WITHOUT_EXPLAINING -> "for i" | "for i and j"
SUBARRAY_RESTRICTION -> "traveling from" ORIGIN "to" DESTINATION
ORIGIN -> LOCATION
DESTINATION -> LOCATION
LOCATION -> "the current location" | "Hotel 1" | "Hotel i" | "Hotel j" | "Hotel k" | "Hotel n"
ADDITIONAL_RESTRICTION -> EPSILON | "using" COMPARISON_OPERATOR COMPARISON_RHS NOUN
COMPARISON_OPERATOR -> "at least" | "at most" | "exactly"
COMPARISON_RHS -> "0" | "1" | "i" | "j" | "k" | "n"

Figure 2: Example problem. (a) problem statement, (b) solution: subproblem definition, (c) instructor-designed CFG

multiple branches[2] of computation (since the PDA is nondeterministic). Next, we inspect all of the branches that have been created; if a branch has a terminal at the top of its stack, then we add this terminal to the set of possibilities for t_{i+1}. Finally, we return this set of possible next terminals to display to the student.

This part of the functionality is implemented entirely on the frontend using JavaScript. Since it runs in the client's web browser, it does not contribute any load to the grading servers and is very responsive. Note that this design is not susceptible to cheating. Although the frontend has access to the CFG, it does not know which sentences generated by the CFG are correct. Thus, even if students inspect the frontend source code, they won't gain access to any information that they shouldn't have.

On an abstract level, the PDA works by expanding the grammar lazily. We only need to expand variables[3] at the top of the stack until we get a terminal at the top of the stack. Variables that aren't at the top of the stack can be left alone, allowing us to avoid an exponential explosion in the number of branches.

For example, given the CFG for the hotel costs problem, the PDA's stack would start out as `[SENTENCE]`. Then, it would get replaced by the RHS of the production rule for `SENTENCE`, so the stack would change to `["Define", FUNCTION_DECLARATION, "to be the", FUNCTION_OUTPUT, "."]`. (The top of the stack corresponds to the left side of the array.) Now, since the terminal `"Define"` is at the top of the stack, it is the only possible next token.

Once the user selects `"Define"` as the next terminal, it gets popped off the stack, leaving us with `[FUNCTION_DECLARATION, "to be the", FUNCTION_OUTPUT, "."]`. Next, since the production rule for `FUNCTION_DECLARATION` has five potential right-hand sides, the PDA splits into five branches. In each branch, `FUNCTION_DECLARATION` is replaced with a different RHS:

- `["the subproblem", "to be the", FUNCTION_OUTPUT, "."]`
- `["DP(i)", "to be the", FUNCTION_OUTPUT, "."]`
- `["DP(i,j)", "to be the", FUNCTION_OUTPUT, "."]`
- `["MinCost(i)", "to be the", FUNCTION_OUTPUT, "."]`
- `["MinCost(i,j)", "to be the", FUNCTION_OUTPUT, "."]`

Each branch's stack has a different terminal at the top, so there are five possible next tokens for the user to select. This process continues until the stack is empty, at which point the user is allowed to submit their response.

3.2 Automated Grading/Feedback

The first step in the grading process is to use the CFG to transform the student's submission into a parse tree.[4] This transformation is illustrated in Figure 3.

Once we have have obtained a parse tree, the autograder uses *constraint-based modeling* to evaluate the response and give feedback to the student. Constraint-based modeling allows the instructor to specify a set of domain principles that must be satisfied in order for a response to be considered correct [8]. Each constraint consists of logic that checks for the presence of specific structures in the parse tree. Furthermore, each constraint can generate feedback whenever it is violated, allowing us to provide students with targeted and actionable feedback. The domain constraints that we used in our model are listed below (in the order that they are checked):

[2]Each "branch" explores a different way to expand the context-free grammar.
[3]"Expanding a variable" refers to replacing a variable with the RHS of its production rule. If a variable's production rule has n potential right-hand sides, this creates n branches, where each branch explores a different RHS for that variable.
[4]We used Python's `nltk` module to accomplish this.

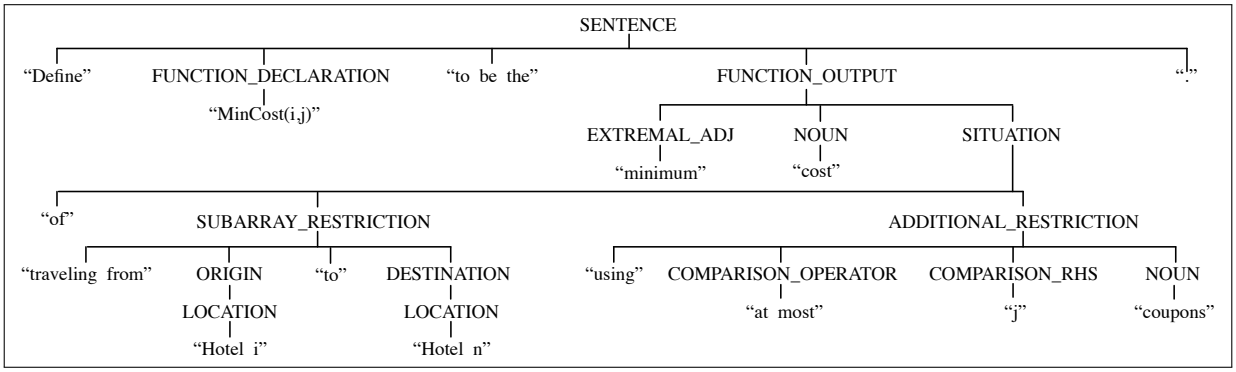

Figure 3: Parse tree for a correct response to the hotel costs problem (generated by the CFG in Figure 2)

- **Declare Function:** The definition must declare a function with a descriptive name and parameters that can be memoized.[5]
- **Clearly State Output:** The definition must clearly state what quantity the function outputs.
- **Explain Parameters:** The definition must explain how the function's input parameters affect the output of the function. Furthermore, all variables mentioned in the definition must either be defined in the original problem or declared as function parameters.
- **Can Compute Final Answer:** We can compute the final answer requested by the original problem using these subproblems. (This can either involve a single subproblem function call that directly returns the final answer, or combining together multiple function calls to obtain the final answer.)
- **Reduces Recursively:** Each subproblem can be reduced to smaller problems that are also handled by the subproblem definition. In other words, it must be possible to write a recursive formula for computing the function declared in the subproblem definition.

The rationale behind this ordering of constraints is that they represent the priority of the issues that need to be fixed. For instance, if a submission doesn't pass the "Declare Function" constraint, then it doesn't make sense to give feedback on any of the later constraints; the student hasn't even declared a function, and all of the later constraints refer to properties of this nonexistent function. Similarly, "Can Compute Final Answer" comes before "Reduces Recursively" because it doesn't make sense to discuss writing a recursive formula to compute the function if this function is useless for solving the original problem.

Another design choice we made in our constraint-based model is that, in some cases, we will intentionally let an incorrect submission pass a constraint if a later constraint can provide feedback that is more relevant to the submission's issue.

Let's take a closer look at the "Can Compute Final Answer" constraint in order to examine how the logic in each individual constraint is implemented. This constraint requires that we must be able to compute the final answer requested by the original problem using the subproblems defined by the student. In the hotel costs problem, the original problem requests that at most k coupons are used, so the subproblem definition must provide a way to impose this restriction. In order to ensure that this constraint is satisfied, the autograder checks for the following conditions:

- The parse tree contains the path `ADDITIONAL_RESTRICTION -> NOUN -> "coupons"`.
- The child of `COMPARISON_OPERATOR` is `"at most"`.[6]
- The child of `COMPARISON_RHS` is `"i"`, `"j"`, or `"k"`.[7]

We can verify that the correct response in Figure 3 satisfies all three of these conditions. If the student's response violates any of these conditions, the constraint would generate the following feedback:[8]

> Your subproblem definition does not allow us to compute the final answer requested by the original problem. The problem requires that at most k coupons are used, but there is no way to impose this requirement using your subproblem definition.

Note that searching for specific structures in the parse tree is more powerful and expressive than performing simple string matching on the student's response. For example, in the constraint described above, we need to check that the definition places a restriction on the correct noun (`"coupons"`). We do this by checking that the path

```
ADDITIONAL_RESTRICTION -> NOUN -> "coupons"
```

is present in the parse tree of the student's response. Simply checking whether the student's response contains the token `"coupons"` would not achieve the same effect, because the student may have used `"coupons"` as the noun to describe the `FUNCTION_OUTPUT` without using it in the `ADDITIONAL_RESTRICTION` section, and this match would show up as a false positive.

[5] Memoization is the optimization technique that allows dynamic programming algorithms to run in polynomial time instead of exponential time. It stores the results of function calls into a table/array so that they don't have to be recomputed. In order to use this technique, the parameters of the function must be valid indexes into the memoization table. In other words, each parameter must be an integer that lies within some fixed domain (or an enum type that can be converted to an integer).

[6] In practice, we also accept `"exactly"` instead of `"at most"` because if the definition says "minimum cost of … using *exactly* i coupons", we can still compute the minimum cost using *at most* k coupons by trying all $i \in \{0, 1, \ldots, k\}$ and taking the minimum.

[7] Using `"k"` would lead to an unviable subproblem definition because the subproblems won't reduce recursively (see website [CITE] for more elaboration). However, it still satisfies the "Can Compute Final Answer" constraint, so it should not trigger this constraint's feedback. The issue with this definition will be detected by the "Reduces Recursively" constraint later in the grading process, and that constraint will generate more appropriate feedback for addressing this issue.

[8] This feedback is written beforehand by the instructor.

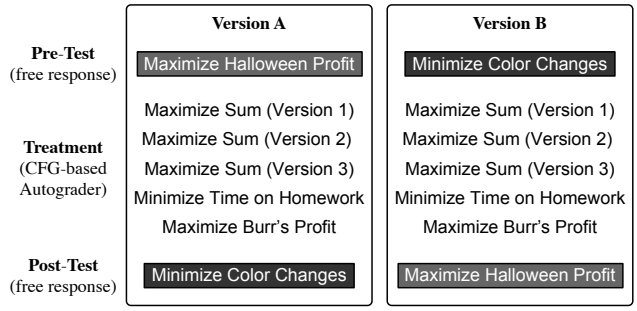

Figure 4: The question order in the two versions of the assignment. Manually-graded questions were used as a pre- and post-test, and autograded questions provided students with instantaneous feedback using the scaffolded writing tool.

4 EXPERIMENTAL METHOD

We evaluated our tool by deploying it on an assignment in an introductory algorithms course taught in Spring 2022 at a large U.S. public university. This course is taken by more than 300 students each semester. Most students are sophomores or juniors and take the course as part of their computer science or computer engineering major (it is required for both majors). The prerequisites are a discrete mathematics and a data structures course. Major topics covered by the course include finite automata, recursive algorithms, dynamic programming, graph algorithms, undecidability, and NP-completeness.

In this study, we were primarily investigating the effectiveness of the tool as low-stakes formative guidance, not the accuracy/reliability of the tool on a high-stakes summative assessment.

4.1 Assignment Setup

The study was performed through a single assignment (see Figure 4) using PrairieLearn [15, 16] that consisted of a pre-test question, five CFG scaffolded writing questions, and a post-test question. The pre-test and post-test questions asked students to specify subproblem definitions as they would on an exam, by writing free-form responses. These responses were typed into a blank input box and were manually graded by the first author after the assignment's due date. Students were forced to complete all seven questions in order.[9] Thus, any difference between pre-test and post-test scores can largely be attributed to the scaffolded writing questions, which were autograded and provided instant feedback. All of the questions and their interactive graders can be found online [17].

Because we expected students to complete the assignment in a single sitting, we decided to use two different questions for the pre-test and post-test to avoid a prior exposure effect. Furthermore, to distinguish the relative difficulty of these two questions from any effect of the treatment, we used two versions of the assignment (see Figure 4). The versions were identical, except the pre-test and post-test questions were swapped. Each version was assigned to a random half of the class.

[9]They could not modify their pre-test response after seeing any of the later questions, and they could not submit a post-test response before completing all of the earlier questions.

4.2 Manual Grading

The manual grading was performed by the first author. Before grading began, the submissions from Version A and Version B were mixed together into the same spreadsheet in a randomized order, and then the version labels were hidden from the grader. This avoids subconscious bias from the grader knowing which submissions are pre-tests and which submissions are post-tests.

For the manually-graded questions, we designed a rubric that produces scores on a scale of 0 to 12, with points allocated as follows:

- **Declare Function:** 3 points
  - Deduct all 3 points if no function is declared at all
  - Deduct 1 if the function does not have a descriptive name
  - Deduct 1 point if the function's parameters are not memoizable (see Footnote 5)
  - Deduct 1 point if the function reuses variables defined in the original problem as function parameters
- **Clearly State Output:** 2 points
  - Deduct both points for not stating the correct noun outputted by the function
  - Deduct 1 point for not having the correct extremal adjective in front of the noun
- **Explain Parameters:** 2 points
  - Deduct 1 point for each insufficiently explained parameter
- **Can Compute Final Answer:** 1 point
  - Deduct 1 point if the subproblems can't be used to compute the final answer
- **Reduces Recursively:** 4 points
  - Deduct 1 point for a minor error/typo that prevents the subproblem from reducing recursively
  - Deduct 2 points for each major oversight that prevents the subproblem from reducing recursively

4.3 Data Handling and IRB Approval

We received IRB approval for this research study. Students were presented with an informed consent form and given the opportunity to opt-out of the study. Participating students' data was anonymized before it was shared with the research team for analysis.

5 RESULTS

After receiving the anonymized data, we filtered out students who didn't submit both the pre-test and the post-test. This left us with 288 students in our dataset (144 from each version of the assignment). Out of these students, 172 of them earned a *strictly* higher score on the post-test than on the pre-test, and only 63 students received a lower score on the post-test. The mean score on the pre-tests was 7.2/12, and the mean score on the post-tests was 9.2/12. A more detailed breakdown of the scores is shown in Figure 5.

To perform an analysis encompassing the pre-/post-test scores from both versions of the assignment, we fit an ordinary least squares (OLS) model of the form

$$s_{ij} = \sigma_j + \beta A_{ij},$$

where s_{ij} is the score that student i received on question j, A_{ij} is 1 if student i had question j as a post-test (0 otherwise), and σ_j and β are the parameters we want to estimate, which can be interpreted as:

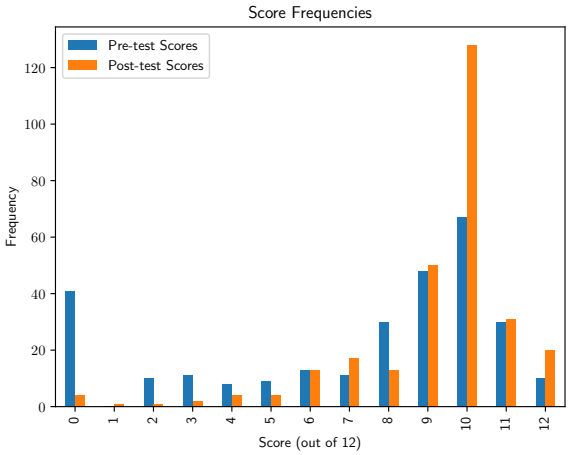

Figure 5: Histogram of scores from the manually-graded questions. The improvement from pre- to post-test results from solving five problems using our tool.

- σ_j: the mean score of question j,
- β: the score improvement resulting from the intervention (i.e., the scaffolded writing tool).

We find that $\beta = 2.014$ (95% CI $[1.513, 2.515]$, $p < 0.0001$), meaning that scores increased by roughly 2 points on a 12-point scale.

Some submissions received low scores (i.e., less than 6/12) because they were were completely off-track. Often, they tried to explain how to implement a recursive function (essentially describing the code in English) instead of stating the function's black-box specification. One example of this (which received 4/12) is shown below:

> We define a function Paint(bool TileIsPainted[], int currentTile, int m, int Color[], int currentColor) which returns an int value that represents the minimum number of color changes required. Every time we change color, we add 1 to the recursive call and change currentColor, TileIsPainted and increment currentTile. Each time we paint incorrectly, we decrement m by 1.

From the histogram, we can see that these off-track submissions were much more likely to correspond to pre-tests rather than post-tests, suggesting that the scaffolded writing tool was effective at getting students to correct major issues and understand the general purpose of a subproblem definition.

However, even within the post-test pool, relatively few submissions earned 100%. We believe this is due to the fact that each problem has unique nuances that need to be dealt with when designing the subproblem definition, and these nuances cannot necessarily be learned by "pattern-matching" from previous examples. Learning to deal with these nuances requires building up intuition through working on many dynamic programming problems, and we wouldn't expect students to master this intuition after just five autograded questions.

Often, these nuances are related to the "Reduces Recursively" constraint, which results in a 2 point deduction according to our rubric. This explains the large peak in post-test scores at 10/12. For example, on the "Minimize Color Changes" question, many students submitted something along the lines of:

> Define $MinChanges(i, j)$ to be the minimum number of color changes required to paint Tiles i through n while making at most j mistakes.

This subproblem does not reduce recursively because we need to know the color that we start with in order to determine whether the color that we choose for Tile i counts as a color change. Thus, submissions like this would receive a score of 10/12. Making the definition viable requires adding another parameter, as follows:

> Define $MinChanges(i, j, k)$ to be the minimum number of color changes required to paint Tiles i through n while making at most j mistakes if we start with Color k on our brush/tray.

In general, it is hard to recognize violations of "Reduces Recursively" constraint without actually attempting to write out the recursive formula and realizing that some aspect of it doesn't work. Even though our tool did provide automated feedback explaining why certain subproblem definitions wouldn't reduce recursively, we never explicitly asked students to write recursive formulas for computing the subproblems that they defined during the study. The lack of focus on this skill may be one reason why students were still making mistakes and oversights on the post-test. Overall, these types of oversights are still relatively minor compared to the issues in the many pre-test submissions that were completely off-track.

6 CONCLUSION AND FUTURE WORK

We have developed a tool for autograding statements in mathematical writing using a novel approach based on context-free grammars. Furthermore, we evaluated the efficacy of this tool within the context of teaching dynamic programming in a large undergraduate algorithms class, and this study produced promising results.

There are many potential directions for future work. Although our study only focused on a specific task in a single course, we believe that the scaffolded writing tool can be expanded to a variety of tasks across a wide range of subjects. In fact, we have already started incorporating the tool into different aspects of our algorithms class such as NP-hardness reductions. More broadly, we believe that the tool could be used to scaffold and autograde *self-explanation* tasks, where students are asked to explain steps in an expert solution. Self-explanation has been shown to facilitate acquisition of knowledge and procedures [7, 11, 12], so having more chances to practice it and receive instantaneous feedback could be very beneficial for students.

In addition, there is room for improvement in the tool's ease of use. For instructors, it would be useful to provide assistance in constructing the CFG, perhaps through a tool that could suggest distractors by identifying common mistakes from student answers to a free response form of the question. For students, we'd like to provide a more convenient way to edit the middle of responses that doesn't require discarding everything past the edit point.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2121424.

REFERENCES

[1] Lijun Chen, Joshua A. Grochow, Ryan Layer, and Michael Levet. 2022. Experience Report: Standards-Based Grading at Scale in Algorithms. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 221–227. https://doi.org/10.1145/3502718.3524750

[2] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[3] Jeff Erickson. 2019. *Algorithms*. Self-published, IL, USA, Chapter 3: Dynamic Programming, 97–157. http://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf

[4] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. 2017. Solving Parsons Problems versus Fixing and Writing Code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '17)*. Association for Computing Machinery, New York, NY, USA, 20–29. https://doi.org/10.1145/3141880.3141895

[5] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. *Autograding "Explain in Plain English" Questions Using NLP*. Association for Computing Machinery, New York, NY, USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[6] Lucas Busatta Galhardi and Jacques Duílio Brancher. 2018. Machine Learning Approach for Automatic Short Answer Grading: A Systematic Review. In *Advances in Artificial Intelligence - IBERAMIA 2018*, Guillermo R. Simari, Eduardo Fermé, Flabio Gutiérrez Segura, and José Antonio Rodríguez Melquiades (Eds.). Springer International Publishing, Cham, 380–391.

[7] Mark Hodds, Lara Alcock, and Matthew Inglis. 2014. Self-explanation training improves proof comprehension. *Journal for Research in Mathematics Education* 45, 1 (2014), 62–101.

[8] Antonija Mitrovic. 2010. *Modeling Domains and Students with Constraint-Based Modeling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 63–80. https://doi.org/10.1007/978-3-642-14363-2_4

[9] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

[10] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2021. Evaluating Proof Blocks Problems as Exam Questions. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/3446871.3469741

[11] Alexander Renkl, Robin Stark, Hans Gruber, and Heinz Mandl. 1998. Learning from Worked-Out Examples: The Effects of Example Variability and Elicited Self-Explanations. *Contemporary Educational Psychology* 23, 1 (1998), 90–108. https://doi.org/10.1006/ceps.1997.0959

[12] Bethany Rittle-Johnson. 2006. Promoting Transfer: Effects of Self-Explanation and Direct Instruction. *Child Development* 77, 1 (2006), 1–15. https://doi.org/10.1111/j.1467-8624.2006.00852.x arXiv:https://srcd.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8624.2006.00852.x

[13] Vasile Rus, Sidney D'Mello, Xiangen Hu, and Arthur Graesser. 2013. Recent Advances in Conversational Intelligent Tutoring Systems. *AI Magazine* 34, 3 (Sep. 2013), 42–54. https://doi.org/10.1609/aimag.v34i3.2485

[14] Michael Sipser. 2013. *Introduction to the Theory of Computation* (third ed.). Course Technology, Boston, MA.

[15] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, Seattle, Washington.

[16] Matthew West, Nathan Walters, Mariana Silva, Timothy Bretl, and Craig Zilles. 2021. Integrating Diverse Learning Tools using the PrairieLearn Platform. In *Seventh SPLICE Workshop at SIGCSE*.

[17] Jason Xia. 2022. Scaffolded CFG-based Writing Tool Demo. https://scaffoldedwriting.pythonanywhere.com.