# Proof Blocks: Autogradable Scaffolding Activities for Learning to Write Proofs

### Seth Poulsen
sethp3@illinois.edu
University of Illinois at Urbana-Champaign
USA

### Mahesh Viswanathan
vmahesh@illinois.edu
University of Illinois at Urbana-Champaign
USA

### Geoffrey L. Herman
glherman@illinois.edu
University of Illinois at Urbana-Champaign
USA

### Matthew West
mwest@illinois.edu
University of Illinois at Urbana-Champaign
USA

## Abstract

In this software tool paper we present Proof Blocks, a tool which enables students to construct mathematical proofs by dragging and dropping prewritten proof lines into the correct order. We present both implementation details of the tool, as well as a rich reflection on our experiences using the tool in courses with hundreds of students. Proof Blocks problems can be graded completely automatically, enabling students to receive rapid feedback. When writing a problem, the instructor specifies the dependency graph of the lines of the proof, so that any correct arrangement of the lines can receive full credit. This innovation can improve assessment tools by increasing the types of questions we can ask students about proofs, and potentially give greater access to proof knowledge by increasing the amount that students can learn on their own with the help of a computer.

## CCS Concepts

• **Mathematics of computing → Discrete mathematics**; • **Social and professional topics → Computing education**; • **Applied computing → Computer-assisted instruction**.

## Keywords

discrete mathematics, CS education, automatic grading, proofs

## 1 Introduction

Constructing mathematical proofs is one of the critical, yet difficult skills that students must learn as a part of the discrete mathematics curriculum. A panel of 21 experts using a Delphi process

agreed that 6 of the 11 most difficult topics in a typical discrete mathematics course are related to proofs and logic [10]. Proofs and proof techniques are included by the ACM curricular guidelines as a core knowledge area that should be understood by any student obtaining a degree in computer engineering, computer science, or software engineering [11, 20, 27].

One problem discrete math instructors face is being able to provide students with rapid feedback on their proof writing skills, since proofs must be graded by hand by instructors or teaching assistants. With the exception of students who are able to sit down with instructors during office hours to receive immediate feedback, most students receive significantly delayed feedback on the correctness of the proofs which they have constructed while completing their homework or exams.

What if students were able to receive in-flow automated feedback on their proofs, just as they are able to with code they write? Providing students a way to write proofs in such a way that a computer can give automated feedback can be a huge advantage. For many students, this will simply be a convenience factor, but for others, gaining automated feedback can be a huge step in increasing equity and access in discrete mathematics education. For example, consider students who are unable to make it to office hours to receive help due to family commitments, or whose university courses are understaffed. For these and other populations, automated feedback has the potential to make a huge difference by giving them access to feedback they wouldn't have otherwise received.

Another difficulty for instructors is scaffolding students as they try to make the jump from seeing their instructor write a proof to writing proofs themselves. To combat this same issue in code writing, researchers have created new types of learning environments and problems including Parson's Problems [21] and block programming languages like Scratch [17] and Blockly [9]. The scaffolding provided by both Parson's Problems and block programming languages have been shown to help students learn more quickly at the beginning of the learning process [8, 29]. Transitioning from seeing others write proofs to writing them on their own requires students to use multiple skills, including writing logical statements and analyzing sequences of logical statements to make sure that each statement is supported by previous ones. Due to the complexity of the task, we believe that students should be given scaffolding for learning to write mathematical proofs, as with writing code, and they will receive similar benefits.

(a) Proof Blocks screenshot
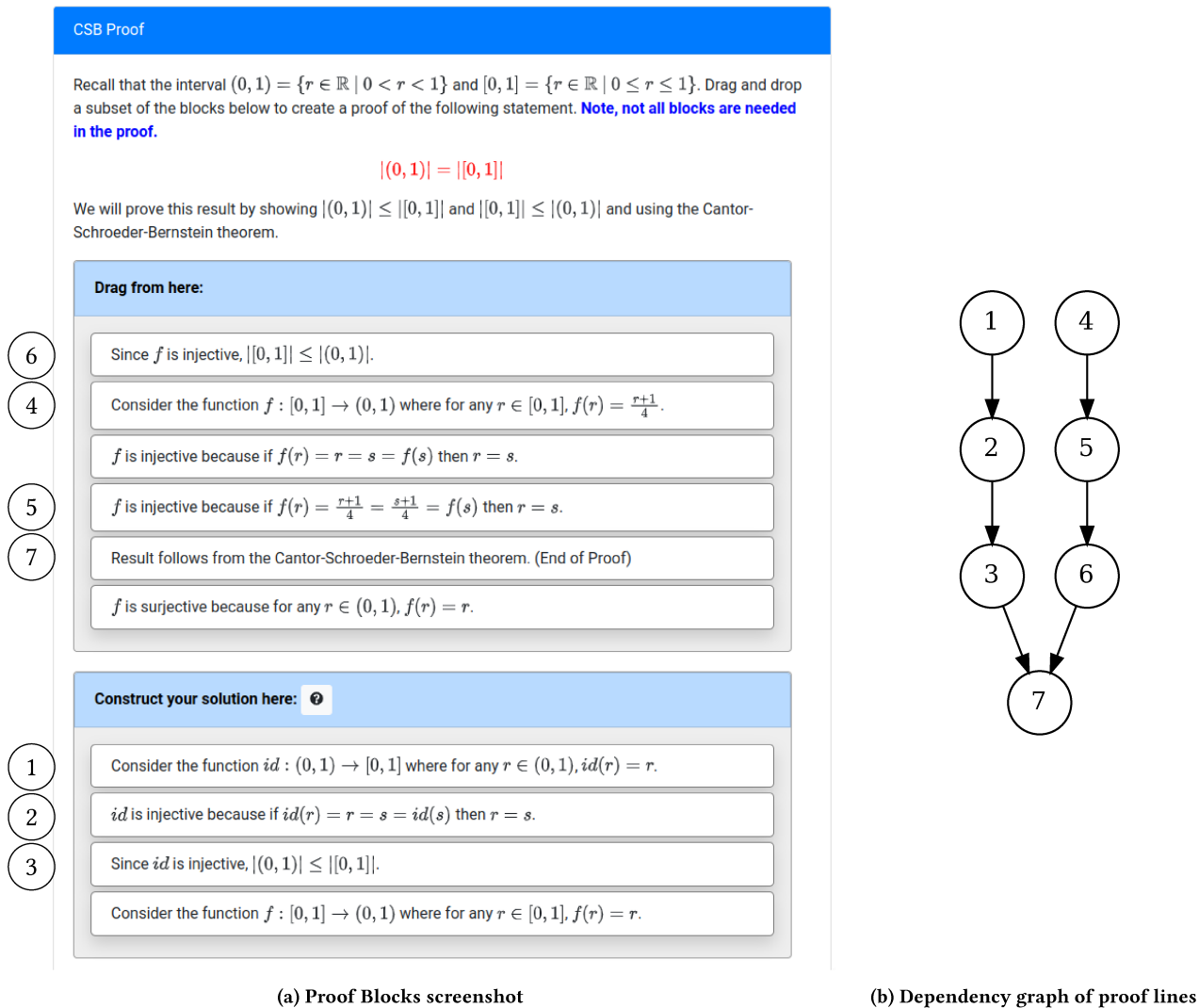
(b) Dependency graph of proof lines

**Figure 1: Example of the Proof Blocks user interface used by students. Individual lines of the proof start out shuffled in the starting zone, and students attempt to drag and drop them into the correct order in the target zone. The instructor wrote the problem with 1, 2, 3, 4, 5, 6, 7 as the intended solution, but the Proof Blocks autograder will also accept any other correct solution as determined by the dependency graph shown in (b) For example, both 4, 5, 6, 1, 2, 3, 7 and 1, 4, 2, 3, 5, 6, 7 would also be accepted as correct solutions.**

In this paper, we present Proof Blocks, a novel user interface for students to construct mathematical proofs by dragging and dropping prewritten statements into the correct order (see Figure 1).

Proof Blocks allows students to receive instant feedback on the proofs they have constructed to accelerate the learning process. It also provides the necessary scaffolding to help students bridge the gap between seeing others write proofs and writing proofs themselves—reminding students to use good practices such as defining variables before using them and being explicit about the proof techniques being employed. Proof Blocks also provide an opportunity for better student assessment, by providing questions which are, on average, more difficult than multiple choice questions given

to students in a typical discrete mathematics course, but easier than free response proof writing questions [23].

The rest of the paper is organized as follows: we will first discuss related work, then proceed by explaining the user interface of Proof Blocks from both the student and instructor perspective. We will also discuss our experience using Proof Blocks in a discrete mathematics course with over 400 students, and then explain the architecture of the autograder and implications for future work.

The specific contributions of this work are:

- A novel grading algorithm for drag-and-drop problems based on a directed acyclic graph
- Application of this algorithm to grading mathematical proofs

- Insights from experiences using this tool with hundreds of students

## 2 Related Work

Work in intelligent tutors for mathematical proofs goes back to work by John Anderson and his colleagues on The Geometry Tutor [1, 2, 12]. More recently, researchers have created tutors for propositional logic, most notably Deep Thought [18, 19] and LogEx [14–16]. A number of other tools have been created to help students learn to construct mathematical proofs with the aid of a computer. Polymorphic Blocks [13] is a novel user interface which presents propositions as colorful blocks with uniquely shaped connectors. The Incredible Proof Machine [5] guides students through constructing proofs as graphs. Jape [4] is a "Proof calculator," which guides students through the process of constructing formal proofs in mathematical notation with the help of the computer, but requires the instructor to implement the logics in a custom language. MathsTiles [3] is a block-based programming interface for constructing proofs for the Isabelle/HOL proof assistant. Having an open-ended environment where students could construct arbitrarily complex proofs seems like an advantage, but user studies showed that students were only successful if they were provided a small instructor-procured subset of blocks.

In reviewing the design of existing tools for computerized proofs it is clear that there is a tension between two desirable properties: ease of use for beginners, and ability to handle complex proofs. The tools which have an elegant, easy to understand interface (Polymorphic Blocks, The Incredible Proof Machine) only cover formal (and in some cases, simple) logics, limiting their usability for discrete mathematics courses where students write informal proofs on a variety of topics from graph theory to number theory. The tools which can handle an arbitrary complexity of proofs are very complex and thus difficult and time consuming for students and instructors to use, especially at the same time as trying to learn to write proofs. Proof Blocks solves this problem by allowing informally written proofs to be automatically graded, making the tool both easy to use, and able to cover topics of all level of complexity, including but not limited to number theory, properties of functions, cardinality, graph theory, Big-O, and combinatorics.

Anecdotally, we have heard of instructors using scrambled proofs to assess student knowledge both in Euclidean geometry and in higher-level mathematics. In theory, instructors may have offered such questions on paper even before the advent of computers, though we can find no explicit record of this. Ensley and Winston offer some scrambled proofs in a JavaScript applet as supplementary material to their discrete mathematics textbook [6]. Such questions could also be written using generic drag-and-drop task widgets that are present in many learning management systems. However, the ability use directed acyclic-graph based grading and to demarcate subproofs are features unique to Proof Blocks that we have not seen in any other system, and enable assessing proofs which are more complex and use a greater variety of writing styles.

## 3 Development Context

Proof Blocks was originally designed for and pilot tested in the discrete mathematics course in the computer science department at the University of Illinois at Urbana-Champaign during the Fall 2020 semester. across multiple sections. Most students are freshmen, and take the course as part of their computer science major, computer science minor, or computer engineering major. The course is designed to prepare students for the theory track in the department and usually covers logic, proofs, functions, cardinality, graphs and trees, induction, recursion, number theory, probability, basic algorithm analysis, and sometimes additional topics as time permits. In Fall 2020, the course was held completely online due to the COVID-19 pandemic. The course was split into 3 sections, each with a different instructor, for a total of over 400 students. The first author of this paper was one of the teaching assistants for the course, and the second author was one of the instructors. For more details of the course, and for the results of the statistical analyses we performed on data collected from the course, see [23].

After the semester, we made some small changes based on feedback that we recieved, and then the features of Proof Blocks were integrated into the core PrairieLearn codebase so that they would be open source and could be used more broadly. PrairieLearn is in use regularly at about 10 universities, with more universities pilot testing. Proof Blocks is being used by courses in three different departments at the University of Illinois, as well as at the University of British Columbia and the University of Chicago, and we expect these numbers to continue to grow.

## 4 User Interface

Proof Blocks is built in to PrairieLearn. Both the student and teacher user interfaces for creating and using Proof Blocks problems are user friendly, and can be used with almost no training. In an anonymous survey given to our students, 46 out of 51 students responded positively to the statement "The proof blocks user interface was easy to use," with the remaining 5 responding neutrally. Additionally, over two thirds of respondents agreed with the statements "Proof Blocks accurately represent my understanding of how to write proofs," and "Proof Blocks would be a good tool for practicing writing proofs." For more detailed survey results, see [23].

### 4.1 Student Interface

Figure 1 shows an example of the Proof Blocks user interface seen by students as they work through Proof Blocks problems. Individual lines of the proof start out shuffled in the starting zone, and students attempt to drag and drop them into the correct order in the target zone. Students were able to successfully complete proofs using Proof Blocks after completing a lecture, worksheet, and homework about proofs, with no training specifically in how to use the interface.

Figure 2 shows an example of feedback given to students working on Proof Blocks problems. This is the feedback that a student would receive if they were to select "Save & Grade" after having put their Proof Blocks into the state shown in Figure 1. To avoid giving students so much information that we are not actually testing their knowledge, they are only told at which line their proof fails and some possible reasons why, not the exact reason why or what the solution is. One area of future research is to iterate on what kind of feedback is best for students to receive when using Proof Blocks as a tool for learning to write proofs.
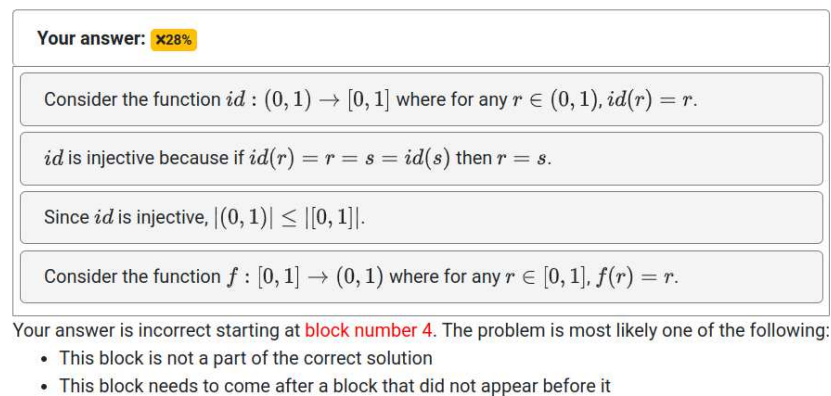
**Figure 2: Example of feedback given to students working on Proof Blocks problems. To avoid giving students so much information that we are not actually testing their knowledge, they are only told at which line their proof fails, not the reason why or what the solution is. One area of future research is to investigate what kind of feedback is best for students to recieve when using Proof Blocks as a tool for learning to write proofs.**

## 4.2 Instructor Interface

In PrairieLearn, each question written by the instructor will include (1) an HTML file defining what the students will see, and (2) a JSON file containing metadata such as the question topic, type, grading options, and author. The HTML file may use custom HTML elements defined by PrairieLearn for writing homework and exam questions. The HTML is then processed on the backend into HTML, CSS, and JavaScript before being delivered to the student's browser.

Figure 3 shows the instructor-written HTML code that generates the Proof Blocks exercise shown in Figure 1. The HTML elements that are prefixed with "pl" have special meaning to PrairieLearn, which processes them on the backend before sending the HTML to the client. The `pl-question-panel` element notifies PrairieLearn of the beginning of a new question. The `pl-order-blocks` signals to PrairieLearn to create the actual Proof Blocks user interface, and each `pl-answer` element inside of it defines a draggable line of proof.

Critically, the instructor writing the problem must specify which lines of the proof must precede each other line. Though seemingly a small detail, it is what makes Proof Blocks such a powerful tool, since it allows instructors to write proofs with arbitrary English language statements. This overcomes the proof complexity constraints of earlier student computer proof systems, and makes it so that students can construct proofs that a computer can grade at any level of complexity. The proof dependencies are declared using the "depends" attribute. For example, the proof graph for the problem shown in Figure 3 is given in Figure 1b.

The instructors of the course were able to create new Proof Blocks questions without any special training by simply looking at those already created by the authors, only asking a few questions for clarification about the configuration options, which could now be answered by looking at the documentation. An instructor can choose for all of the given lines to be required, or can add in distractor lines which are not part of the proof. In our discrete mathematics course, we used test questions both with and without distractor lines. Whether or not having distractor lines in the problem leads to

better assessment or learning outcomes is an open question which we leave for future work.

## 5 Best Practices For Question Writing

Our experience using Proof Blocks with hundreds of students led us to a few best practices in having Proof Blocks problems work well for students.

The principal cause for an erroneous Proof Blocks question is because the instructor failed to recognize a possible rearrangement of the proof lines that is logically consistent. This results in a correct student response being incorrectly marked as faulty by the autograder. Unfortunately, it is easy to make such mistakes when designing a Proof Blocks question. These can be avoided if the instructor is aware of the main reasons this arises, which we outline below. In addition, we recommend that the instructor ask another member of the course staff who did not design the question, to solve the problem in different ways *without looking at the source code*. In our experience, these steps help catch all such mistakes.

One example of this is when the instructor identifies more dependencies between the proof lines than actually exist. For example an instructor may code up a problem in a manner which specifies to the autograder that each line in the proof depends on the line before it. Such strong dependencies are rarely demanded in any proof. While this is a simple scenario where additional dependencies have been identified, other cases are more subtle. They often arise because experienced mathematicians follow *stylistic norms* in addition to logical dependencies when structuring their proofs. These are so ingrained in a practicing mathematician, that stylistic norms inadvertently seep in as logical dependencies when coding up a problem. For example, one often structures proof with subgoals, with the proof of a new subgoal begun only *after* the proof of the previous subgoal has been finished. A classical example in a discrete mathematics class is where students are asked to prove a statement using induction where the proof of the induction step follows a complete proof of the base case. However, often there is no logical dependence between the statements in the subproof of each case. From a logical perspective, the proof statements for

```
<pl-question-panel>
  <p>Recall that the interval
  $(0,1) = \{r \in \mathbb{R}\: |\: 0 \lt r \lt 1\}$ and $[0,1] =  \{r \in \mathbb{R}\: |\: 0 \leq r \leq 1\}$.
  Drag and drop a subset of the blocks below to create a proof of the following
  statement. <strong style="color:blue;">Note, not all blocks are needed in the proof.</strong></p>
  <p style="color:red;text-align:center;">$|(0,1)| = |[0,1]|$</p>
  <p>We will prove this result by showing $|(0,1)| \leq |[0,1]|$ and $|[0,1]| \leq |(0,1)|$ and using the
  Cantor-Schroeder-Bernstein theorem.</p>
</pl-question-panel>

<pl-order-blocks feedback="first-wrong" answers-name="csb-v1" grading-method="dag">
  <pl-answer correct="true" tag="1" depends="">Consider the function $id: (0,1) \to [0,1]$ where for any $r \in (0,1)$,
    $id(r) = r$.</pl-answer>
  <pl-answer correct="true" tag="2" depends="1">$id$ is injective because if $id(r) = r = s = id(s)$ then $r=s$.</pl-answer>
  <pl-answer correct="true" tag="3" depends="2">Since $id$ is injective, $|(0,1)| \leq |[0,1]|$.</pl-answer>
  <pl-answer correct="true" tag="4" depends="" >Consider the function $f: [0,1] \to (0,1)$ where for any $r \in [0,1]$,
    $f(r) = \frac{r+1}{4}$.</pl-answer>
  <pl-answer correct="true" tag="5" depends="4">$f$ is injective because if $f(r) = \frac{r+1}{4} = \frac{s+1}{4} = f(s)$
    then $r=s$.</pl-answer>
  <pl-answer correct="true" tag="6" depends="5">Since $f$ is injective, $|[0,1]| \leq |(0,1)|$.</pl-answer>
  <pl-answer correct="true" tag="7" depends="3,6">Result follows from the Cantor-Schroeder-Bernstein theorem.
    (End of Proof)</pl-answer>

  <!-- Distractors -->
  <pl-answer correct="false" tag="" depends="">Consider the function $f: [0,1] \to (0,1)$ where for any $r \in [0,1]$,
    $f(r) = r$.</pl-answer>
  <pl-answer correct="false" tag="" depends="">$f$ is injective because if $f(r) = r = s = f(s)$ then $r=s$.</pl-answer>
  <pl-answer correct="false" tag="" depends="">$f$ is surjective because for any $r \in (0,1)$, $f(r) = r$.</pl-answer>
</pl-order-blocks>
```

**Figure 3: The instructor-written HTML code that generates the Proof Blocks exercise shown in 1. The HTML elements that are prefixed with "pl" have special meaning to PrairieLearn, which processes them on the backend before sending the HTML to the client. The "depends" property on each "pl-answer" element is used to declare the dependency between statements in the proof structure.**

each case can be interleaved in any manner. Of course, emphasizing stylistic norms is just as important a learning objective, but in that case instructors should be encouraged to spell this goal out in the problem statement. To avoid such mistakes, after coding a Proof Blocks question, we encourage instructors to examine the dependencies of each line in the coded problem in isolation, without the large proof context.

Another common cause for errors arises in proofs that contain many algebraic manipulation steps. In informal proof writing, it is often acceptable to skip intermediate steps of algebraic manipulation. Coding a question in a manner that demands all the steps leads to student complaints about the autograder. There are two ways to address this problem. One is to write multiple algebraic simplification steps in a single proof statement in the problem. The second, and probably the best, is to avoid having any distractors in the problem, and notify the student that *all* blocks should be used to construct a correct proof.

The last cause for an error could be distractors. When designing a question, it is useful to remember that none of the distractors should be part of *any* correct proof. A common mistake is to have distractors that are superfluous to the correct proof; this is a problem because we can write logically correct proofs that have additional statements that do not contribute to the end goal. Thus, it is important to ensure that adding any distractor would result in a logically inconsistent argument. One simple way to ensure this is to have each distractor (on its own) be a logically inconsistent statement. Even though this might seem like an easy distractor for a student to avoid, in practice we have found that students are nonetheless confounded by such distractors. In the future, we hope to extend the Proof Blocks grader so that it can also handle having lines which can optionally be part of a correct proof.

## 6 Autograder

The autograder is currently built in to PrairieLearn, but the core algorithm is about 100 lines of Python code that could be made to work with an alternative frontend, or reimplemented in any other language.

While creating the tool, we recognized that it would be a poor student experience if the student was expected to place the lines of the proof in the exact order which the instructor first wrote them, because in many mathematical proofs, certain lines can be permuted without affecting the correctness of the proof. It would also be a poor user experience for the instructor if they had to explicitly declare every possible correct answer to each question. This led us to our current grading scheme, which is based on the dependency

graph of the lines in the proof, which is a directed acyclic graph (DAG). The instructor simply declares the dependency graph of statements in the proof, and then the grader will accept any correct permutation of the lines.

In the basic case, where a proof has no subproofs like the example in Figure 1, checking if a proof is correct is equivalent to checking if the student ordering of the lines is a topological sort of the DAG. A more rigorous treatment of the grading algorithm, and the details of our edit-distance based partial credit algorithm can be seen in [22].

## 6.1 Subproofs

Even in an introductory discrete mathematics course, an instructor may want to use proofs that have cases. For example, using cases to prove an "or" statement, or proof by induction. Here each subproof is a connected subgraph of the entire proof graph. In such cases, checking for topological sorting of the proof DAG is insufficient, because this would allow for intermixing of statements from separate subproofs in a nonsensical fashion. A correct proof is a topological sort of the lines of the proof with the added condition that the lines of each subproof must be listed contiguously. Therefore, there is an extra check which ensures that once a given subproof is started, it is finished before any lines from outside the subproof appear.

To write a question with a subproof, the instructor wraps each subproof in a `pl-block-group` element. The `pl-block-group` element may then be given its own tag, so that lines that logically depend on the entire subproof can refer to them in their `depends` attribute. For more details and examples of problems with subproofs, see the Proof Blocks documentation [25]. As noted in Section 5, it is important to note that subproofs declared only for stylistic, and not logical, reasons can be misleading for student unless they are explicitly notified of the style which they are to follow.

## 7 Evaluation

Using data from hundreds of student exams from the discrete math course in Fall 2020, we have shown that Proof Blocks problems provide about as much information about student knowledge as written proof problems do. We have also shown that as test questions, Proof Blocks problems are in fact easier than written proofs, which are often very difficult. An anonymous survey given to these students showed that students felt that Proof Blocks problems accurately represented their ability to write proofs, and that the user interface was easy to use. Full details of this evaluation can be seen in [23]. Ongoing evaluation work seeks to explore the possibility that Proof Blocks can help students learn more efficiently than writing proofs from scratch, just as Parson's Problems can help students learn more efficiently than writing code from scratch [8].

## 8 Adopting Proof Blocks

To use Proof Blocks with your students, start by following the onboarding instructions for PrairieLearn [26]. Once familiar with the basic workings of PrairieLearn, follow the documentation for writing Proof Blocks questions [25]. More example problems can be found in the documentation and example courses. PrairieLearn is in the process of integrating with Learning Tools Interoperability [24] to enable easier sharing of student data across learning platforms. Feel free to reach out to the authors with any questions, or about the possibility of adding Proof Blocks support on other platforms.

## 9 Limitations

The key limitation of Proof Blocks is that it restricts what students can do, only allowing them to place prewritten lines into their proof rather than allowing them to write whatever they want. As with Parson's Problems and block based programming languages, we expect that there is a certain skill level at which Proof Blocks will become a hindrance rather than a help to students, but this is of course expected for all forms of education scaffolding. Similarly, we believe that Proof Blocks will can be a huge help for students who are just getting started in learning to write proofs. Another limitation is that proofs will be graded correctly only as long as the instructor correctly codes the question—but this is really no worse than most other types of exam questions given to students. Finally, Proof Blocks is currently only usable within the PrairieLearn. Ongoing efforts to improve interoperability between PrairieLearn and other learning platforms will help ease adoption.

## 10 Future Work and implications

The versatility of the Proof Blocks platform makes it ideally suited for many future avenues of research. Next, we would like to enable automatic generation of Proof Blocks problems so that students can have essentially unlimited practice. We will also want to research a way to predict the difficulty of a given generated problem, so students can be guided through questions of varying difficulty as the learn, and for fairness on assessments. Beyond discrete mathematics, there are great possibilities in using Proof Blocks problems for other courses involving proofs such as algorithms courses, or even high school geometry.

As noted, Proof Blocks are a good way to bridge the gap between students learning to read and write proofs. To give further support to students as they learn to write proofs, we can try variations on Proof Blocks. For example, we could have students drag and drop lines of a proof which are mostly prewritten, but have some blanks for the students to fill in, much as Weinman et al. have done with their "Faded Parson's Problems" [28]

Proof Blocks can increase access to proof knowledge by helping students gain more rapid feedback on the proofs they write. Additionaly, it can improve assessment tools by increasing the types and difficulty levels of questions we can ask students about proofs, and save many hours of instructor grading time which can be reallocated to office hours or other effective means of helping students [23]. There has been some evidence that mathematics is acting as a gatekeeper to learning programming, and that it doesn't actually predict performance in software developers [7]. Furthermore, many people going into software development study curricula that involve less math than a standard computer science curricula. Proof Blocks can also provide a solution in this case: rather than teaching less mathematics, Proof Blocks provides a middle ground. Students can be introduced to logical thinking and proof writing in a gentler way, potentially reducing the gatekeeping of mathematics while helping students learn the content.

## Acknowledgments

# References

[1] JR Anderson, CF Boyle, and G Yost. 1985. The Geometry Tutor, proc. of 9th Internation Joint Conference on Artificial Intelligence.

[2] John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *The journal of the learning sciences* 4, 2 (1995), 167–207.

[3] William Billingsley and Peter Robinson. 2007. Student proof exercises using Math-sTiles and Isabelle/HOL in an intelligent book. *Journal of Automated Reasoning* 39, 2 (2007), 181–218.

[4] Richard Bornat and Bernard Sufrin. 1997. Jape: A calculator for animating proof-on-paper. In *International Conference on Automated Deduction*. Springer, 412–415.

[5] Joachim Breitner. 2016. Visual theorem proving with the Incredible Proof Machine. In *International Conference on Interactive Theorem Proving*. Springer, 123–139.

[6] Douglas E Ensley and J Winston Crawley. 2005. *Discrete mathematics: mathematical reasoning and proof with puzzles, patterns, and games.* John Wiley & Sons.

[7] Nathan L Ensmenger. 2012. *The computer boys take over: Computers, programmers, and the politics of technical expertise.* Mit Press.

[8] 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.* 20–29.

[9] N. Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000

[10] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C Loui, and Craig Zilles. 2008. Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education.* 256–260.

[11] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* Association for Computing Machinery, New York, NY, USA.

[12] Kenneth R Koedinger and John R Anderson. 1990. Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science* 14, 4 (1990), 511–550.

[13] Sorin Lerner, Stephen R Foster, and William G Griswold. 2015. Polymorphic blocks: Formalism-inspired UI for structured connectors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems.* 3063–3072.

[14] Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2015. A pilot study of the use of LogEx, lessons learned. *arXiv:1507.03671 [cs]* (July 2015). http://arxiv.org/abs/1507.03671 arXiv: 1507.03671.

[15] Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2019. A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae. *Journal of Computer Assisted Learning* 35, 5 (2019), 620–632. https://doi.org/10.1111/jcal.12365 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/jcal.12365.

[16] Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2020. Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction. *Electronic Proceedings in Theoretical Computer Science* 313 (Feb. 2020), 17–34. https://doi.org/10.4204/EPTCS.313.2 arXiv: 2002.12552.

[17] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[18] Behrooz Mostafavi and Tiffany Barnes. 2017. Evolution of an Intelligent Deductive Logic Tutor Using Data-Driven Elements. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 5–36. https://doi.org/10.1007/s40593-016-0112-1

[19] Behrooz Mostafavi, Guojing Zhou, Collin Lynch, Min Chi, and Tiffany Barnes. 2015. Data-Driven Worked Examples Improve Retention and Completion in a Logic Tutor. In *Artificial Intelligence in Education*, Cristina Conati, Neil Heffernan, Antonija Mitrovic, and M. Felisa Verdejo (Eds.). Vol. 9112. Springer International Publishing, Cham, 726–729. http://link.springer.com/10.1007/978-3-319-19773-9_102 Series Title: Lecture Notes in Computer Science.

[20] The Joint Task Force on Computing Curricula. 2014. *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* Technical Report. New York, NY, USA.

[21] 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.

[22] Seth Poulsen, Shubhang Kulkarni, Geoffrey Herman, and Matthew West. 2022. Efficient Partial Credit Grading of Proof Blocks Problems. https://doi.org/10.48550/ARXIV.2204.04196

[23] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2021. Evaluating Proof Blocks Problems as Exam Questions. In *Proceedings of the 2021 ACM Conference on International Computing Education Research.*

[24] Charles Severance, Ted Hanss, and Joseph Hardin. 2010. Ims learning tools interoperability: Enabling a mash-up approach to teaching and learning tools. *Technology, Instruction, Cognition and Learning* 7, 3-4 (2010), 245–262.

[25] PrairieLearn Team. 2021. pl-order-blocks Documentation. https://prairielearn.readthedocs.io/en/latest/elements/#pl-order-blocks-element

[26] PrairieLearn Team. 2021. PrairieLearn Documentation. https://prairielearn.readthedocs.io/en/latest/

[27] Association for Computing Machinery (ACM) The Joint Task Force on Computing Curricula and IEEE Computer Society. 2016. *Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering.* Technical Report. New York, NY, USA.

[28] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. Exploring challenging variations of parsons problems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* 1349–1349.

[29] David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children.* 199–208.