Proof Blocks: Autogradeable 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

Proof Blocks is a software tool which enables students to write proofs by dragging and dropping prewritten proof lines into the correct order. These proofs can be graded completely automatically, enabling students to receive rapid feedback on how they are doing with their proofs. When constructing 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 can 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

ACM Reference Format:
Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2018. Proof Blocks: Autogradeable Scaffolding Activities for Learning to Write Proofs. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Understanding and writing 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 [9]. 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 [10, 13, 17].

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.

In contrast, students working on programming assignments are able to get constant, continuous feedback from their computer as they write code. Even though they aren't able to receive full feedback on the correctness of their solutions, there are many correctness properties which they can easily check on their own with the help of their compiler and both self-written and instructor-provided automated tests.

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 [14] and block programming languages like Scratch [12] and Blockly [8]. 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 [7, 19]. Transitioning from seeing others write proofs to writing them on their own requires students to use multiple skills, including writing logical statements

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

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.

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 [15].

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.

2 RELATED WORK

A few other software tools have been created to enable students to create proofs in the computer in such a way that they can receive automated feedback. Some use text-based representations, while others use visual representations of proofs.

Polymorphic Blocks [11] is a novel user interface which presents propositions as colorful blocks with uniquely shaped connectors as a signifier of which types of propositions can be connected into a proof. While the user interface has been shown to engage students in learning proofs, it supports only propositional logic. The Incredible Proof Machine [5] guides students through constructing proofs as graphs. As with Polymorphic Blocks, the user interface is engaging, but the formality of the system limits the topics which can be effectively covered.

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. While Jape can allow students to construct proofs in arbitrary logics, it requires the instructor to implement these logics in its own custom programming language before students can use them to construct proofs.

MathsTiles [3] is a block-based programming interface for constructing proofs for the Isabelle/HOL proof assistant. In theory, having an open-ended environment where students could construct arbitrarily complex proofs seems like it could be a huge advantage. However, in user studies, the authors found that students only had a chance at being successful while using MathsTiles if they were provided only a small instructor-procured subset of blocks, namely, those needed to complete the problem at hand.

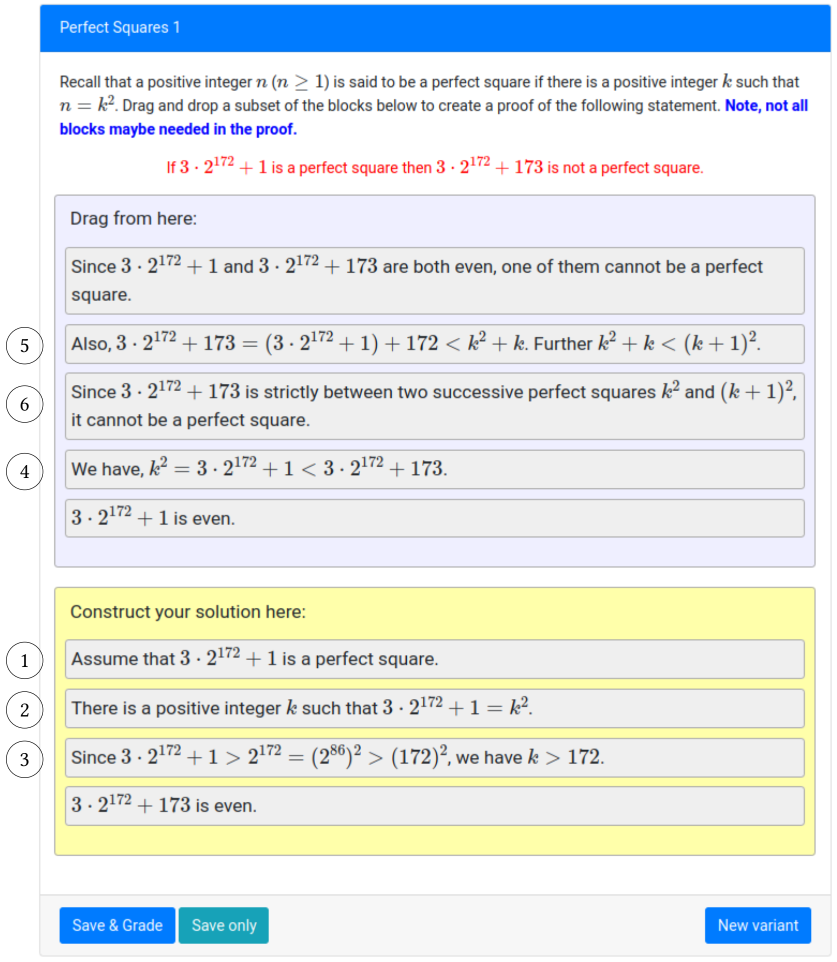
In reviewing the design of existing tools for computerized proofs it is clear that there is a tension between two desirable properties: ease of use for beginners, and ability to handle complex proofs. The tools which have an elegant, easy to understand interface (Polymorphic Blocks, The Incredible Proof 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 formally 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.

3 COURSE CONTEXT

Proof Blocks has been used by hundreds of students in the discrete mathematics course at a large public research university in the United States. At this university, the discrete mathematics course is taught every semester (including during the summer) and is taken by hundreds of students each 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 listed prerequisites for the course are introductory programming and introductory calculus. 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. Though taught in the computer science department, it is solely a theory class, with no programming assignments.

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. Each week students watched a lecture video recorded by one of the instructors, and then spent class time working together in small groups on a worksheet over video conferencing software, with teaching assistants present to assist. They were then assigned homework. At the beginning of the next week, students took a short exam, on the material covered the previous week. Some weeks, the students were also given a practice exam to assist in studying. For the final exam, students were given the opportunity to retake any three of the exams. Students took their exams using PrairieLearn, an online open-source homework and exam platform [20]. Exams typically consisted of three to five multiple choice or fill in the blank questions, one or two Proof Blocks questions, and one or two free response written proofs. Especially for a course of this size, Proof Blocks' fully automated grading was also a big advantage in saving course staff time which could be reallocated in other ways. In total, students were given 10 Proof Blocks questions on exams and 3 on practice exams. Students received immediate correctness feedback on each Proof Blocks question on their exams and were given up to three attempts at each question, with a decreasing number of points awarded depending on the number of attempts needed.


Perfect Squares 1

Recall that a positive integer n ($n \geq 1$) is said to be a perfect square if there is a positive integer k such that $n = k^2$. Drag and drop a subset of the blocks below to create a proof of the following statement. **Note, not all blocks maybe needed in the proof.**

If $3 \cdot 2^{172} + 1$ is a perfect square then $3 \cdot 2^{172} + 173$ is not a perfect square.

Drag from here:

Since $3 \cdot 2^{172} + 1$ and $3 \cdot 2^{172} + 173$ are both even, one of them cannot be a perfect square.

5 Also, $3 \cdot 2^{172} + 173 = (3 \cdot 2^{172} + 1) + 172 < k^2 + k$. Further $k^2 + k < (k+1)^2$.

6 Since $3 \cdot 2^{172} + 173$ is strictly between two successive perfect squares k^2 and $(k+1)^2$, it cannot be a perfect square.

4 We have, $k^2 = 3 \cdot 2^{172} + 1 < 3 \cdot 2^{172} + 173$.

$3 \cdot 2^{172} + 1$ is even.

Construct your solution here:

1 Assume that $3 \cdot 2^{172} + 1$ is a perfect square.

2 There is a positive integer k such that $3 \cdot 2^{172} + 1 = k^2$.

3 Since $3 \cdot 2^{172} + 1 > 2^{172} = (2^{86})^2 > (172)^2$, we have $k > 172$.

$3 \cdot 2^{172} + 173$ is even.

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

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 [15].

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 light-blue starting zone, and students attempt to drag and drop them into the correct order in the yellow 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 3 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.

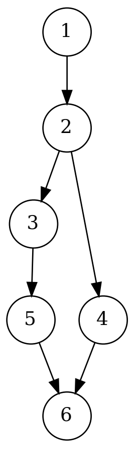


Figure 2: The dependency graph of the statements in the proof shown in Figure 1. The Proof Blocks grader will accept any topological sort of this directed acyclic graph as a correct solution.

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 4 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-drag-drop` signals to PrairieLearn to create the actual Proof Blocks user interface, and each `pl-answer` element inside of it defines a draggable line of the 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 4 is given in Figure 2.

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.

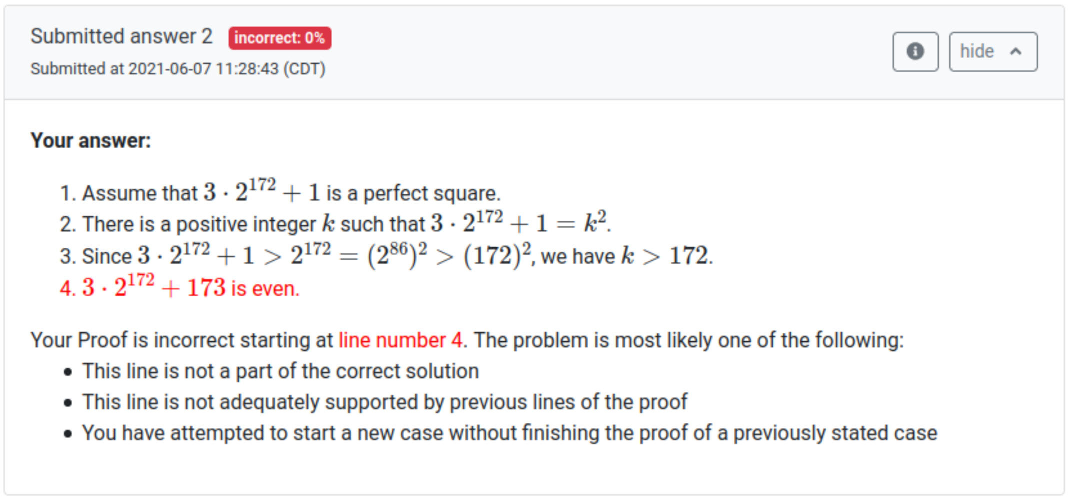
5 BEST PRACTICES FOR QUESTION WRITING

Our experience using Proof Blocks with over 400 students this semester 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.

The most common cause for errors 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 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.

The second 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.



Your answer:

1. Assume that $3 \cdot 2^{172} + 1$ is a perfect square.
2. There is a positive integer k such that $3 \cdot 2^{172} + 1 = k^2$.
3. Since $3 \cdot 2^{172} + 1 > 2^{172} = (2^{86})^2 > (172)^2$, we have $k > 172$.
4. $3 \cdot 2^{172} + 173$ is even.

Your Proof is incorrect starting at line number 4. The problem is most likely one of the following:

- This line is not a part of the correct solution
- This line is not adequately supported by previous lines of the proof
- You have attempted to start a new case without finishing the proof of a previously stated case

Figure 3: 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.

```
<pl-question-panel>
  <p>Recall that a positive integer $n$ ($n \geq 1$) is said to be a perfect square if there is a positive integer $k$ such
  that $n = k^2$. Drag and drop a subset of the blocks below to create a proof of the following statement.
  <strong><font color="blue">Note, not all blocks maybe needed in the proof.</font></strong></p>
  <p><center><font color="red">If $3\cdot 2^{172}+1$ is a perfect square then $3\cdot 2^{172}+173$ is not a perfect
    square.</font></center></p>
</pl-question-panel>


<pl-order-blocks answers-name="squares" source-blocks-order="random" grading-method="dag" solution-placement="bottom">
  <pl-answer correct="true" id="1" depends="">Assume that $3\cdot 2^{172}+1$ is a perfect square.</pl-answer>
  <pl-answer correct="true" id="2" depends="1">There is a positive integer $k$ such that $3\cdot 2^{172} + 1 = k^2$.</pl-answer>
  <pl-answer correct="true" id="3" depends="2">Since $3\cdot 2^{172} + 1 > 2^{172} = (2^{86})^2 > (172)^2$, we have $k > 172$.
    </pl-answer>
  <pl-answer correct="true" id="4" depends="2">We have, $k^2 = 3\cdot 2^{172} + 1 < 3\cdot 2^{172} + 173$.</pl-answer>
  <pl-answer correct="true" id="5" depends="3">Also, $3\cdot 2^{172} + 173 = (3\cdot 2^{172} + 1) + 172 < k^2 + k$.
    Further $k^2 + k < (k+1)^2$.</pl-answer>
  <pl-answer correct="true" id="6" depends="4,5">Since $3\cdot 2^{172} + 173$ is strictly between two successive
    perfect squares $k^2$ and $(k+1)^2$, it cannot be a perfect square.</pl-answer>

  <!-- Distractors -->
  <pl-answer correct="false">$3\cdot 2^{172} + 1$ is even.</pl-answer>
  <pl-answer correct="false">$3\cdot 2^{172} + 173$ is even.</pl-answer>
  <pl-answer correct="false">Since $3\cdot 2^{172} + 1$ and $3\cdot 2^{172} + 173$ are both even, one of them cannot be a
    perfect square.</pl-answer>
</pl-order-blocks>
```

Figure 4: 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.

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.

6 AUTOGRADER

The autograder is currently built in to PrairieLearn, but the core algorithm is about 70 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.

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 in the grader which ensures that once a given subproof is started, it is finished before any lines from a parallel subproof appear.

To write a question with a subproof, the instructor labels each of the `pl-answer` HTML elements which represent a line in a subproof with a unique string identifier for that subproof, specified using the `subproof` attribute. For more details and examples of problems with subproofs, see the Proof Blocks documentation [1]. 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 fall 2020, we have shown that Proof Blocks problems are in fact easier than written proofs, which are often very difficult. We have also shown that as test questions, Proof Blocks problems provide about as much information about student knowledge as written proof problems do. 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 [15].

8 ADOPTING PROOF BLOCKS

To use Proof Blocks with your students, start by following the onboarding instructions for PrairieLearn [2]. Once familiar with the basic workings of PrairieLearn, follow the documentation for writing Proof Blocks questions [1]. More example problems can be found in the documentation and example courses. PrairieLearn is in the process of integrating with Learning Tools Interoperability [16] 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.

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" [18]

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 [15]. 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 [6]. 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.

REFERENCES

[1] 2021. pl-order-blocks Documentation. https://prairielearn.readthedocs.io/en/latest/elements/#pl-order-blocks-element

[2] 2021. PrairieLearn Documentation. https://prairielearn.readthedocs.io/en/latest/

[3] William Billingsley and Peter Robinson. 2007. Student proof exercises using MathsTiles 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] Nathan L Ensmenger. 2012. *The computer boys take over: Computers, programmers, and the politics of technical expertise*. Mit Press.

[7] 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.

[8] 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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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*.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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, 26.1238.1–26.1238.14. https://peer.asee.org/24575.