Spoof-It: Correcting incorrect proofs as a method to learn proof-writing

Omar Ibrahim
B.S./M.S. Candidate
2022
Paul G. Allen School of Computer Science

Advised by:
Lauren Bricker
Robbie Weber

ABSTRACT

The ability to write proofs is a core skill of an undergraduate computing degree. However, it is also a notoriously difficult skill to learn and master. I piloted a new technique for teaching proof-writing skills to students: having students identify errors in given incorrect proofs (”spoofs”), in order to better understand the ways proofs are constructed and the common pitfalls of proof-writing. I created and used the Spoof-It tool to provide a way for students to complete these kinds of problems and receive immediate feedback on their work, as a novel method to improve their proof-writing skills. Despite limited results, from my experiences described here, I believe that “spoof” problems fill an important gap in discrete mathematics assessments, are relatively easy to implement, and when used in the right contexts, have the potential to benefit students’ ability to write their own proofs from scratch.

Table of Contents

List of Figures

List of Tables

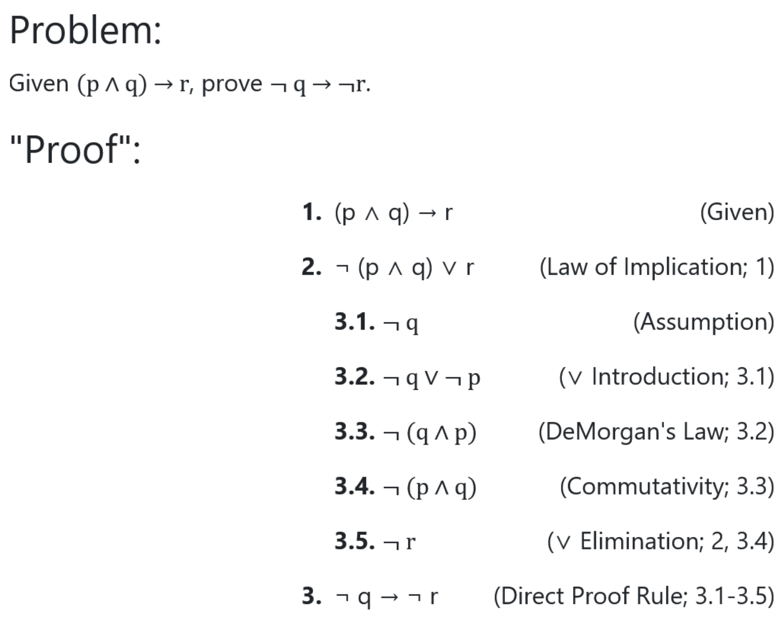
1 INTRODUCTION

Proof-writing is a notoriously tricky skill for students to learn at all levels; it is also a core part of computer science education. 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 [1, 12, 13]. Computer Science students often take a discrete mathematics course that is meant to teach students those foundational proof-writing skills, and it is generally agreed upon that some of the most difficult topics in those courses are related to proofs and logic [9].

There is not much published research on improving methods for teaching proof-writing, however, the problems students face learning to write proofs are not all unique to proof-writing. Students' struggle with writing proofs is, at least partially, a struggle to write a functional and well-structured product in an unfamiliar format or syntax. This framing of the difficulties suggests an analogy between learning to write code and learning to write proofs. In other computing areas, we have seen approaches for teaching students to write code that provide scaffolding for students to learn to program without having to write all (or any of) the code themselves. There is an abundance of literature in the introductory computer science space on helping students gain the basic skills for code-writing; thus, I posit there may be value in adapting approaches from that area for teaching students the basics of proof-writing. Parson's problems, which ask students to assemble a set of pre-written lines of code into a functional program fulfilling a given purpose, are one example that has proven successful in teaching students to code [7, 8]. Similar approaches for assembling proofs from given blocks have also shown promise in teaching students to write proofs [19], which indicates that other mechanical approaches for teaching code-writing can be successfully adapted for teaching proof-writing.

Bug-fixing problems are one such approach. There is some basis to suggest that bug-fixing problems can measure student code-writing ability as well as traditional code-writing questions can [6], and possibly also teach students to write code as well as code-writing problems [8]. I wanted to investigate if the bug-fixing approach might also apply to teaching students to write proofs. One reason why proof-writing is a difficult skill for many students to master is the variety of mistakes that can be made; students commonly commit logical fallacies such as beginning with the conclusion of the proof, assuming the converses of given theorems are true, proving something weaker than the conclusion of the proof, or circular reasoning [20]. I hypothesized that we could improve their proof-writing skills by exposing them to and teaching them about the common mistakes that might be made in proofs.

I created Spoof-It as a tool to provide students with practice identifying and fixing errors in given proofs without writing proofs from scratch. Spoof-It presents students with a claim along with a logically incorrect proof of that claim; students must determine what part of the proof is logically incorrect, and describe why it is incorrect. Figure 1 shows an example of a problem in Spoof-It. These "spoof" problems are structured in such a way that the student solving them does not have to write any part of the actual proof. The goal is that students become more familiar with logical inconsistencies and common errors they might make when writing proofs, and better recognize, avoid, and fix those same errors writing their own proofs. These common errors include committing basic logical fallacies, improperly declaring variables, using incorrect methods for proving implications, confusing different inference rules, etc. These 'spoof' problems could serve as a better first step for students learning new proof-writing concepts and techniques before writing the full proofs themselves, and provide an opportunity to provide students with immediate feedback on the correctness of their work.



Problem:

Given $(p \land q) \rightarrow r$, prove $\neg q \rightarrow \neg r$.

"Proof":

1.	$(p \land q) \rightarrow r$	(Given)
2.	$\neg (p \land q) \lor r$	(Law of Implication; 1)
3.1.	$\neg q$	(Assumption)
3.2.	$\neg q \lor \neg p$	($\lor$ Introduction; 3.1)
3.3.	$\neg (q \land p)$	(DeMorgan's Law; 3.2)
3.4.	$\neg (p \land q)$	(Commutativity; 3.3)
3.5.	$\neg r$	($\lor$ Elimination; 2, 3.4)
3.	$\neg q \rightarrow \neg r$	(Direct Proof Rule; 3.1-3.5)

Fig. 1. A problem in Spoof-It where the writer has mistakenly claimed in step 3.5 that because one component of an OR statement is true, the other must be false.

2 RELATED WORK

2.1 Parson's Problems and Proof Blocks

There has been much research in the CS Education (CS Ed) field on the idea of teaching students to write code without writing the actual code themselves. One such example is Parson's problems, which have been extensively studied in CS Ed for their ability to identify common logical student errors while being more engaging to students than traditional programming exercises [7], provide immediate feedback for students, and reduce cognitive load for students. Further, there is no statistical difference in student learning performance between Parson's problems and traditional code-writing exercises [8]. There is less such work on the theoretical side of CS. However, there is research on implementing similar methods in discrete mathematics courses to have students learn to construct proofs and be evaluated on proof-writing ability mechanically rather than by writing proofs entirely from scratch [16], most notably Poulsen, et al. in creating Proof Blocks as a proof-writing analogue for Parson's problems [19]. These works have shown promising results for applying mechanical problem solving to students learning to write proofs.

Additionally, there is prior work on having students fix incorrect code rather than writing the code themselves. Prior research by Cheng, et al. suggests these methods can be used to measure student code-writing ability as well as traditional problems can [6]. There does not appear to be much similar published work with proofs.

2.2 Teaching Proof Writing

There has been some research on what students struggle with when writing proofs and why. In one study, Weber demonstrated that among undergraduate students learning to write proofs, students are unable to write proofs despite knowing and being able to apply all of the prerequisite facts to prove the statement [23]. This finding shows that the struggles with writing proofs are not simply about prerequisite knowledge, but are inherently connected to being able to structure and construct a proof, which demonstrates a need for more scaffolded practice with proof-writing for students forming those skills. In discrete mathematics, most problems presented to students are writing parts of or full proofs, but these problems may not provide enough scaffolded support to serve as functional formative assessments [26]. Hodds, et al. showed that giving students self-explanation training to better engage with proofs had lasting benefits on proof comprehension in pedagogical settings [10]. I hope that in a similar fashion, we can improve student proof comprehension through spoof problems that require students to not only identify the incorrect part of the proof, but also identify why it is incorrect and how it can be fixed.

Students in proof-writing classes are usually introduced to new ideas and techniques by being shown or reading proofs [21, 24], so their learned ability to correctly write proofs on their own is intrinsically linked to their ability to read and comprehend proofs. However, there is also a wealth of work in mathematics pedagogy that shows students aren't able to reliably judge if the proofs they are reading are valid or invalid [11, 14, 21, 25], implying that they may not be able to judge if the proofs that they *write* are valid or invalid either. Thus, I view it as a vital intervention to improve student proof comprehension through explicitly teaching students to identify and correct errors in proofs. A good place to start is training students to identify and correct errors in proofs that they already know are logically incorrect: spoofs. That training can build to determining *whether* a given argument is proof or spoof, and then further extended to training on examination of their own work through the same lens.

2.3 Feedback

Research in educational psychology has shown that learning requires practice with frequent feedback [4], and has indicated that immediate feedback can be more useful on complex tasks where students have less prior knowledge [22]. This has been a contributing factor in the increasing use of autograders and other similar tools for teaching programming. However, the drive towards automated feedback for students on their programming work has not been as common within proof-based classes. It is common for discrete mathematics students to not be able to receive timely feedback on their proof-writing work. Autograding written proofs is a non-trivial task due to the requirement of assessing logical soundness from something written in natural language. Some tools have been created with the goal of providing students with fast automated feedback on proof-writing practice, notably The Incredible Proof Machine [5], Polymorphic Blocks [15], Jape [3], MathTiles [2], and Proof Blocks [19]. These tools all provide mechanical approaches to writing or assembling proofs that can be programmatically graded. I aimed to create a similar tool to fill a different niche; rather than guiding students through writing proofs by arranging given proof parts, Spoof-It helps students understand how to identify mistakes in proof-writing and how to fix them. Mechanical spoof problems allow us to provide students with the kind of immediate feedback they can receive when writing code from automated tests or compilers.

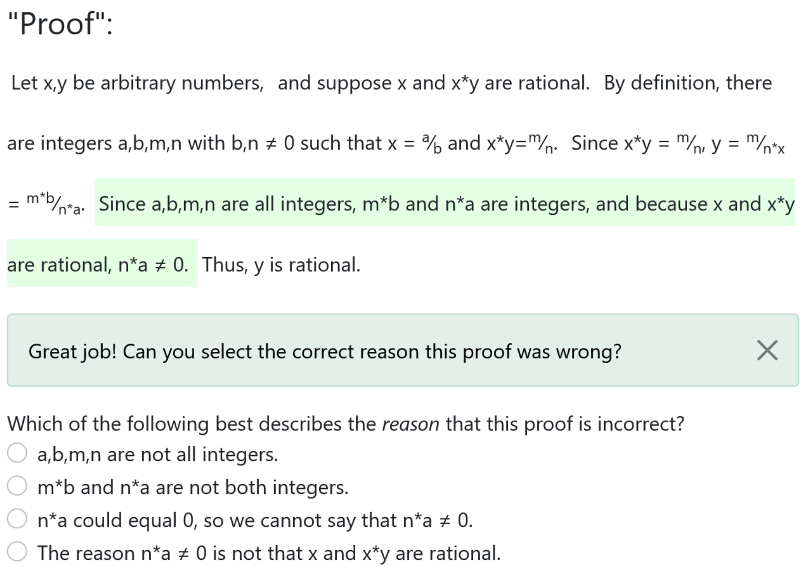
3 COURSE CONTEXT

At the University of Washington, the Allen School of Computer Science & Engineering (CSE) offers a discrete mathematics course (CSE 311, Foundations of Computing 1) that is a requirement for earning a CS or CE undergraduate degree. Students typically take the course sometime within their second year (with the exception of transfer students, who typically take it in their first term after transferring). The department also offers a simultaneous-enrollment support workshop course alongside the discrete mathematics course that is meant to provide extra support and practice for students (CSE 390Z, Mathematics for Computation Workshop). The course was created to address an achievement gap for historically minoritized students in the main discrete mathematics course, and is targeted specifically at a few student population groups (STARS, Startup, and transfer students) that are disproportionately made up of first generation students, historically marginalized students, Pell Grant eligible students, and female students [17, 18]. However, other students are also welcome to and do enroll in the course. I taught this support workshop course during the duration of this study, and worked closely with the professor teaching the main discrete mathematics course. Students who participated in this study were all volunteers from CSE 390Z. Choosing to participate in the study was entirely voluntary and did not impact students' grades in either course. The University of Washington's Human Subjects Division deemed this research exempt from requirement for IRB approval.

4 METHODOLOGY

4.1 Spoof-It Design

The inspiration for this project is problems that evaluated students' ability to locate and correct issues in incorrect proofs piloted in CSE 311 ("spoof" problems). The approach I took for this project simplified this format further for autograding implementation; I presented students with a claim and an incorrect attempt at proving that claim and asked students to identify the first place that an error was made within the proof. The goal was that in doing so, students would both learn common mistakes made in proof-writing and how to avoid them, and gain a better understanding of the proofs they were reading by reasoning through what was incorrect about them. Because of the simple mechanical presentation of these problems, student answers are relatively easy to programmatically check.


"Proof":

Let x,y be arbitrary numbers, and suppose x and x*y are rational. By definition, there are integers a,b,m,n with b,n ≠ 0 such that x = $^a/_b$ and x*y=$^m/_n$. Since x*y = $^m/_n$, y = $^m/_{n*x}$ = $^{m*b}/_{n*a}$. Since a,b,m,n are all integers, m*b and n*a are integers, and because x and x*y are rational, n*a ≠ 0. Thus, y is rational.

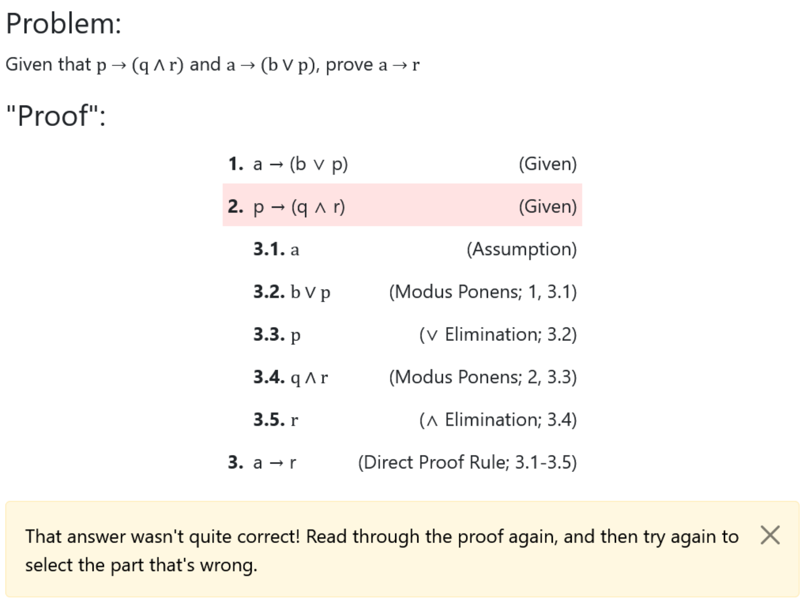
Great job! Can you select the correct reason this proof was wrong?

Which of the following best describes the *reason* that this proof is incorrect?

- a,b,m,n are not all integers.
- m*b and n*a are not both integers.
- n*a could equal 0, so we cannot say that n*a ≠ 0.
- The reason n*a ≠ 0 is not that x and x*y are rational.

Fig. 2. A correct answer screen in Spoof-It prompting students to select why the spoof was incorrect, shown after a student correctly identifies the error in the given spoof

The Spoof-It tool used by students was designed by the author to provide a simple interface for students to interact with problems. It is a web-app that presents a statement to prove along with an (incorrect) proof of that statement. Some of the statements were true but "proven" incorrectly, while other statements were false and "proven" due to mistakes or fallacies in the proof. Student participants were given access to the tool, with problem types being unlocked for them to complete as they learned the associated course material in their discrete math course. When a problem was completed correctly, students were given a congratulatory message and prompted to explain why that part of the proof was incorrect (shown in Figure 2).



Problem:

Given that $p \to (q \wedge r)$ and $a \to (b \vee p)$, prove $a \to r$

"Proof":

1.	$a \to (b \vee p)$	(Given)
2.	$p \to (q \wedge r)$	(Given)
3.1.	a	(Assumption)
3.2.	$b \vee p$	(Modus Ponens; 1, 3.1)
3.3.	p	($\vee$ Elimination; 3.2)
3.4.	$q \wedge r$	(Modus Ponens; 2, 3.3)
3.5.	r	($\wedge$ Elimination; 3.4)
3.	$a \to r$	(Direct Proof Rule; 3.1-3.5)

That answer wasn't quite correct! Read through the proof again, and then try again to select the part that's wrong.

Fig. 3. An incorrect answer screen in Spoof-It, shown after a student incorrectly identifies the error in the given spoof

When a user submitted an incorrect answer, they were prompted to try again, and not allowed to submit the same answer again (shown in Figure 3). They were given unlimited attempts to solve the problems. While there was concern that the unlimited attempts might be abused to find a correct answer by brute force, this did not happen with any of my student participants (possibly because there were no grade-based stakes for them getting the correct answer).

4.2 Problem Design and Selection

The most basic form of spoof problem is asking students to identify significant errors in a proof, and there are levels of complexity that can be added from that base. The first level of complexity is asking students to explain why the errors they have identified are incorrect. A further level of complexity can be added by having students explain how to fix those errors. The highest level of complexity for a spoof problem is to ask students if the claim the spoof attempts to prove is true; if it is, write a corrected version of the proof, and if it is not, provide a reason or counterexample showing that it is not. An alternative presentation of a spoof problem (that I did not use) could present students with a claim and a potential proof of that claim, and ask students to identify whether or not that proof is correct. If it is not, they can follow the same steps laid out for the previous kinds of spoof problems: identify, explain, and correct errors in the incorrect proof.

All of the spoof problems used were written by me for Spoof-It. In order to make using Spoof-It as straightforward as possible, all problems were designed around the basic spoof problem format to have exactly 1 step that is objectively incorrect for students to identify. The rest of the steps were designed to be correct in that they logically followed from givens and previous steps; by nature of how a proof is written, any steps that followed after the 'incorrect' step would likely also be untrue, but for the purposes of problem construction and solving those steps were considered to be 'correct' in that they logically followed from givens and previous statements.

4.3 Student Participants

In total, student participants were given 4 "assignments" to complete in Spoof-It across 4 different kinds of proofs:

- *Formal Inductive Proofs*: 2-column proofs with numbered steps about basic propositional and/or predicate logic.
- *English Number Theory Proofs*: Proofs written in plain English about properties of numbers, numerical series, and number theory.
- *English Set Theory Proofs*: Proofs written in plain English about properties of sets.
- *Induction Proofs*: Proofs written in plain English using mathematical induction.

These assignments were given to students as soon as they learned the associated material, with the intent that they would complete these assignments prior to being assessed on their understanding of that material in assignments in their discrete mathematics course (CSE 311). There were 19 total student volunteers from CSE 390Z. 10 completed Assignment 1 (Formal Inductive Proofs), 4 completed assignments 2 and 3 (English Number Theory Proofs and English Set Theory Proofs, respectively), and 2 completed Assignment 4 (Induction Proofs). Due to the different levels of participation across assignments, I chose to focus on the 4 participants who completed assignments 1-3, and did not use assignment 4 or its related problems in my data.

4.4 Evaluation

The assignments in the student participants' main discrete mathematics course were used as proxies for measuring their understanding of proof-writing skills to avoid assigning extra time-consuming work to students; this also provided a baseline for how students not using Spoof-It were doing with these skills in comparison. These assignments contained 'spoof' problems similar to the kinds of problems students would experience in Spoof-It, as well as problems that had them write full proofs from scratch. These problems were categorized into four types; Identifying Errors, Explaining Errors, Correcting Proofs, and Writing Proofs from Scratch.

- *Identifying Errors.* These problems present students with an incorrect attempt at a proof of a claim, and ask students to identify "significant errors". Here, a "significant error" was defined as one that could lead to a false conclusion, or which requires inserting multiple steps to correct. An insignificant error was defined as something that could be fixed quickly, such as using the wrong name for an inference rule or stylistic issues.
- *Explaining Errors.* These problems usually involve identifying an error in a proof, like the previous category, but additionally require students to explain the significant error, and potentially explain how the proof could be fixed (or *if* it could be fixed, and why; some spoofs came to entirely incorrect conclusions).
- *Correcting Proofs.* These problems require students to write the correct(ed) version of an incorrect proof. This category of problem is usually built on problems of the previous two types but was sometimes presented on its own.
- *Writing Proofs from Scratch.* These problems require students to write a proof for a given claim from scratch, with no given spoof or scaffolding. These are generally the traditional type of problems used for assessment in a proof-writing class.

Table 1. The number of each type of problem student scores were collected from

Problem Type	Number of Problems
Identifying Errors	2
Explaining Errors	6
Correcting Proofs	5
Writing Proofs from Scratch	11

Table 2. Avg. Percentage Scores on different categories of problems for participants, control, and class avg

	Identifying Errors	Explaining Errors	Correcting Proofs	Writing Proofs from Scratch
Participant 1	100	89.6	100	81.4
Participant 2	100	95.8	91.3	93.0
Participant 3	68.8	100	100	96.4
Participant 4	100	83.3	100	81.5
Control Avg	96.8	93.9	95.1	88.7
Class Avg	92.5	93.6	91.4	81.6

A larger group of 28 students from the same workshop class (CSE 390Z) were used as a control group to compare participants to. For each problem used as a measure of student understanding, I collected the percent grade each participant and control student received on the problem, as well as the overall discrete mathematics course (CSE 311) class average for that problem.

5 RESULTS

Summarized results of the Spoof-It participants' performance on the different categories are detailed in Table 2, along with the performance of the workshop class (CSE 390Z) student control group, and the overall discrete mathematics course (CSE 311) class average. In general, both Spoof-It participants and the control group performed better than the main discrete mathematics class average across all assignments and problem types. Both groups performed similarly on spoof problems; this may be because those problems were relatively easier and involved less nuance, and most students did very well on them. When it came to actually writing proofs, both groups did better than the class average. There was no clear consensus on one group doing better than the other, however. The Spoof-It participants ranged from doing slightly (a few percent) worse to moderately (~10%) better than the control group on each problem. However, the Spoof-It participants did perform the best on average on every assessment on problems that involved writing the corrected versions of given incorrect proofs, which could indicate a stronger understanding of how to fix errors in proofs once they were found.

Due to the limited number of participants, I chose to focus on analyzing individual participants' experiences and progression rather than a group-level analysis. All of the participants completed assignments regularly throughout the course term, and some reported that they felt spoof-style practice problems were helpful for them. There did not appear to be any noticeable trend across the term for any of the participants; the lack of pattern suggests the format of having students regularly complete these problems in addition to the rest of their coursework at the very least did not have a negative impact on their learning. However, I would hesitate to say that doing spoof problems had no benefit at all for students, as all of my participants did do exceptionally well on the problems asking them to complete incorrect proofs. This could suggest that these problems can be useful if accompanied by training on how to transfer those skills beyond straightforward spoof problems.

The spoof problems seemed to have the largest impact on each of the participant's ability to correct incorrect proofs, which was the category of problem each participant did the best in, both objectively and in comparison to the control and class averages. However, that impact did not seem to translate to students' ability to write correct proofs from scratch, as those grades tended to not show much sign of improvement over the course of the term and were less distinguished from the rest of the class. It is possible that although students were getting a better understanding of how to identify and correct errors in proofs when asked, they were not applying those new skills to examining the proofs they themselves were writing. Interventions pushing students to self-examine their own proofs, such as was studied by Hodds et. al. [10], could help them translate gained proof-examining skills to the proofs that they write, and result in more correct proofs written by students.

6 DISCUSSION

6.1 Limitations

My work was heavily limited in scope and statistical significance by the small number of participants. I chose to assess efficacy based on student grades on existing assessments in the discrete mathematics class to avoid putting extra work on student participants. However, this choice also limited the specificity of the kinds of data that I could collect and the kinds of problems I could present student participants. I were also limited in distinguishing between the effects of spoof problems and the effects of the Spoof-It tool; my only method of measuring the impact of the spoof problems was tied to those problems being presented in Spoof-It, so the efficacy of these problems on their own may be different. There were also spoof-type problems on their main course assessments, which means that all students got some exposure to these kinds of problems through summative assessments; my participants only got unique additional exposure to these kinds of problems in formative assessments through Spoof-It. This may have further limited my ability to distinguish the impacts of these problems. I also was limited by only being able to offer these participants practice with Spoof-It in addition to their regular course work rather than as a replacement. This work was run over the course of one 10-week quarter and I was thus limited to running this study with one class for only one term, and having a limited amount of time to try different approaches with students. I noted previously that it was important for spoof problems to focus on "significant" errors, however, there were some problems that I wrote for Spoof-It that had "insignificant" main errors (small, easily fixable syntactic issues).

6.2 Future Work

Future work on spoof problems should include an evaluation of spoof problems presented on their own without any associated tool/technology. I only used spoof problems as formative practice problems, but a good next step would be an empirical study of whether spoof-type problems are able to assess students' proof-writing abilities as well as traditional proof-writing problems can. It would also be beneficial to empirically study the effects of replacing traditional proof-writing formative assessments with spoof problems, as compared to being used in addition to traditional proof-writing problems (as spoof problems were in my study). This could be useful to determine if students can learn to write proofs as well as they would traditionally. There are also a few potential impacts of using spoof problems that I was not able to look at that merit future study, namely assessing if and how doing spoof problems impacts students' ability to identify mistakes in reasoning and ability to determine whether a given proof is correct or incorrect. As mentioned in Section 5, I noticed an improvement in students' ability to correct proofs that was not accompanied by an improvement in their ability to write proofs from scratch. For this reason, it seems like another avenue for future work could be to study the impact of spoof problems paired with the kind of self-examination training studied by Hodds et al. [10].

7 CONCLUSION

Spoof problems appear to fill an important gap in the kinds of problems discrete mathematics students were solving. The Spoof-It tool was one approach I took for presenting those problems, but it is certainly not the only possible approach. Spoof-It was only built as a tool to collect specific data for the purposes of this study. The most important part of using spoof problems is careful problem writing; the errors written into spoofs should be significant and unambiguous, and represent the kinds of errors you most care about students avoiding. Spoof problems should be focused on students' ability to recognize logical inconsistencies, not mechanical ones, and so inserted logical inconsistencies should be as mechanically consistent with good proof-writing as possible.

My experience using ”spoof” problems is limited, but shows promise. These kinds of problems are not a replacement for other methods of learning proof-writing, but they do appear to fill an important niche in the kinds of problems presented to discrete mathematics students that is not currently well addressed. With careful problem construction and thoughtful problem presentation, these problems have the potential to be valuable tools in teaching students to write proofs well.

ACKNOWLEDGMENTS

I'd like to thank Robbie Weber for his support in this study as the professor of the main discrete mathematics course and as my advisor, as well as Lauren Bricker for her support as my advisor. I'd also like to thank Adam Blank for introducing UW to the term ”spoof”, and the students of CSE 390Z 22sp for participating in this study.

REFERENCES

[1] Mark Ardis, David Budgen, Gregory W Hislop, Jeff Offutt, Mark Sebern, and Willem Visser. 2015. SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering. *Computer* 48, 11 (2015), 106–109.

[2] 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.

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

[4] John D Bransford, Ann L Brown, Rodney R Cocking, et al. 2000. *How people learn*. Vol. 11. Washington, DC: National academy press.

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

[6] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating coding ability without writing any code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 123–128.

[7] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[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] 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] 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.

[11] Matthew Inglis and Lara Alcock. 2012. Expert and novice approaches to reading mathematical proofs. *Journal for Research in Mathematics Education* 43, 4 (2012), 358–390.

[12] 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.

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

[14] Yi-Yin Ko and Eric J Knuth. 2013. Validating proofs and counterexamples across content domains: Practices of importance for mathematics majors. *The Journal of Mathematical Behavior* 32, 1 (2013), 20–35.

[15] 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.

[16] Mark McCartin-Lim, Beverly Woolf, and Andrew McGregor. 2018. Connect the dots to prove it: A novel way to learn proof construction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* 533–538.

[17] Paul G. Allen School of Computer Science & Engineering. 2022. Allen School Demographics. https://www.cs.washington.edu/diversity/demographics

[18] University of Washington College of Engineering. 2020. Washington STARS In Engineering Impact Report 2013-2019. https://www.engr.washington.edu/sites/engr/files/curr_students/docs/STARS-Impact-Report.pdf

[19] Seth Poulsen, Mahesh Viswanathan, Geoffrey L Herman, and Matthew West. 2022. Evaluating proof blocks problems as exam questions. *ACM Inroads* 13, 1 (2022), 41–51.

[20] Annie Selden and John Selden. 1987. Errors and misconceptions in college level theorem proving. In *Proceedings of the second international seminar on misconceptions and educational strategies in science and mathematics*, Vol. 3. ERIC, 457–470.

[21] Annie Selden and John Selden. 2003. Validations of proofs considered as texts: Can undergraduates tell whether an argument proves a theorem? *Journal for research in mathematics education* 34, 1 (2003), 4–36.

[22] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.

[23] Keith Weber. 2001. Student difficulty in constructing proofs: The need for strategic knowledge. *Educational studies in mathematics* 48, 1 (2001), 101–119.

[24] Keith Weber. 2004. Traditional instruction in advanced mathematics courses: A case study of one professor's lectures and proofs in an introductory real analysis course. *The Journal of Mathematical Behavior* 23, 2 (2004), 115–133.

[25] Keith Weber. 2010. Mathematics majors' perceptions of conviction, validity, and proof. *Mathematical thinking and learning* 12, 4 (2010), 306–336.

[26] Dylan Wiliam. 2011. What is assessment for learning? *Studies in educational evaluation* 37, 1 (2011), 3–14.