Tools for Teaching Theoretical Computer Science

Sam Lowenstein

December 19, 2019
Version: Final Draft
Advisor: John Dougherty

Abstract

Teaching students the concepts and techniques for understanding theoretical computer science is an essential part of any computer science curriculum, yet thus far educators are struggling to help their students master this material. There is still research to be done to identify the causes of these difficulties, but results thus far indicate that they stem from a lack of interest and a lack of basic proficiency in the practice of theory amongst students. Researchers have applied a wide variety of pedagogical methods to attempt improve student performance, including the Moore Method, problem-based learning, and cognitive apprenticeship, but despite some success, improving pedagogy may not be sufficient.

To fully engage students in theoretical computer science, digital tools can be created and used. The theorem prover Coq, originally designed for research, has been repurposed for use in the classroom. Coq shows promise as an educational tool, as students who use it are able to write proofs more easily, but they still struggle when required to write proofs without aid. Several other tools have also been developed, the most noteworthy of which is JFLAP, which helps students visualize the operation of automata. Despite the increased enjoyment that JFLAP fosters, however, it is unclear if it improves student performance. New research suggests that games might be introduced to theoretical computer science classes to help students engage with the material. JFLAP, which already contains elements of gamification, has been further gamified in a project by Fabio Bove. I propose to extend this game with an adaptive hint system to ensure that students receive the assistance they need to learn from it.

Contents

1 Introduction

Theoretical computer science, a discipline that includes formal languages, automata, and complexity theory (Kelley 1995), is an important part of the college computer science curriculum. Though it has fewer direct practical applications than other branches of computer science, it helps students learn to think mathematically and reach a deeper understanding of the concepts behind the other branches (Armoni et al. 2006). However, students often have trouble in theoretical computer science courses (Sigman 2007). Researchers have attempted to determine the causes of this difficulty by examining how students complete coursework in these classes. Observing three groups of students at work, the researchers notice that the students' gaps in understanding extended even to fundamental principles of mathematical reasoning and conclude that the students' struggles are a result of their lack of basic proficiencies. There is evidence to suggest, however, that students are missing these proficiencies as a result of their lack of interest in the material (Knobelsdorf and Frede 2016). Others have suggested a lack of problem-solving skills (Pillay 2009) and the dissimilarity to other areas of computer science (Hamalainen 2004) as potential causes of the problems students are experiencing.

Some researchers propose applying various pedagogical methods to address these challenges. The Moore Method, which replaces lectures with student presentations on their independent work, has been successful in small classes but may be difficult to implement on a large scale (Sigman 2007). Problem-based learning, a hybrid method which combines lectures with student-driven problem-solving sessions, could also be useful, though there was no evidence that it improved students' grades (Hamalainen 2004). Cognitive apprenticeship, which places emphasis on allowing all interactions between students and instructors to contribute to the learning process, has also met with success in reducing student failure rate, though there are still improvements to be made (Knobelsdorf et al. 2014). Pedagogical methods are only part of the picture, however, as the material itself is still seen as prohibitively difficult. Researchers are therefore developing tools to aid students in mastering the material of theoretical computer science courses.

The Coq Proof Assistant, which helps the user prove theorems through an application of type theory, has been suggested as a tool to teach students proficiency in proof-writing. Coq is promising, as it resembles the kind of programming environment that

computer science students are accustomed to. The researchers have tested Coq on a small class and conclude that it is effective at teaching students proof-writing skills. However, the results indicate that, while the students became better at using Coq throughout the course, their ability to write proofs independently was still lacking (Knobelsdorf et al. 2017).

The Java Formal Languages and Automata Package (JFLAP) aims to help students visualize and simulate automata (Procopiuc et al. 1996). It also proposes to aid students in working through important proofs in theoretical computer science (Gramond and Roger 1999). A study across fourteen universities applied JFLAP to college classes, and students indicated that they found abstract concepts easier to understand when using JFLAP. However, their grades were not significantly better than those of a control group (Rodger et al. 2009).

Some researchers have tried to improve theoretical computer science classes by introducing games. One class required students to design games represented by automata, which helped weaker students engage with the material, though stronger students were dissatisfied with the approach (Korte et al. 2007). JFLAP already includes a game using the Pumping Lemma (*Java Formal Languages and Automata Package 7.1*), which may be very helpful for students struggling with the complicated lemma. In his bachelor's thesis, Fabio Bove adds more elements of gamification to JFLAP, introducing a game in which the user must convert between different representations of a formal language (Bove 2015).

However, Bove's hint system may fail to aid students who are struggling to understand the material to begin with. As a result, I suggest improving on the gamified version of JFLAP by adding an adaptive hint system that provides users with enough help to solve each problem while still remaining intellectually challenging.

Literature Review

2

2.1 Teaching Theoretical Computer Science

Though computer science is often associated solely with programming and circuitry, the discipline also has a strong theoretical foundation. College computer science curricula may include one or more courses in theoretical computer science. Such courses often build on the theory of formal languages (sets of strings) and their various representations, such as finite automata and grammars. These topics lead up to discussions of Turing machines, which are automata with finite states but also infinite tapes containing series of characters. Turing machines, which can not only recognize languages but also compute functions, can then be used to introduce complexity theory (Kelley 1995). There exist variations on this formula, but these are the central unifying concepts that will be referred to henceforth as "theoretical computer science."

Theory of computation is a critical topic in the undergraduate computer science curriculum. Researchers Armoni, Rodger, Vardi, and Verma argue that its importance is often understated. Armoni points out the importance of introducing a variety of computational models and a different way of thinking about concepts in computer science. Vardi explains that automata theory has been successfully applied to other areas of computer science for decades, ranging from circuit design to logic program optimization (Armoni et al. 2006).

There are some who argue for reducing the role of theoretical computer science in undergraduate education. In "Computing Theory with Relevance," Brookes describes how industry workers with computer science degrees sometimes believe their time learning theoretical computer science to have been wasted, and students are generally more invested in learning skills that they will directly apply in jobs. As a result, computer science curricula are moving away from theory (Brookes 2004). The authors of "Our Curriculum Has Become Math-Phobic" argue that this reduction of theoretical computer science education deprives students of the chance to learn crucial mathematical reasoning skills. Moreover, as technology advances, they contend, techniques based on today's technology are liable to become obsolete, but the theory behind them will not change so easily. Understanding the theoretical foundations of computer science could allow graduates to adapt along with scientific

progress and prepare them for longer-term investment in the field (Tucker et al. 2001).

2.1.1 The Difficulty of Teaching Theory

Yet teaching theory of computation is no easy task. In "Engaging Students in Formal Language Theory and Theory of Computation," Scott Sigman explains that undergraduate students tend to struggle with the course material due to a number of factors. First, the formal mathematical style of theory is so different from other courses as to be almost unrecognizable as a computer science course to students accustomed to courses on programming and circuit design. Furthermore, it is difficult, especially during the early stages of theory courses, to draw connections between theory of computation and other areas of computer science, which can cause students to question its importance (Sigman 2007).

The authors of "Analyzing Student Practices in Theory of Computation in Light of Distributed Cognition Theory" look deeper into the struggles of undergraduate students in theoretical computer science courses. They challenge the notion that students lack interest in theory, noting that few studies have actually examined this assumption, and they suggest a more evidence-based approach to identifying difficulties. Only then, the authors argue, will researchers be able to create techniques to address these struggles (Knobelsdorf and Frede 2016).

The authors base their work on Distributed Cognition Theory (DCog), which examines how cognition is built by interactions amongst individuals or between individuals and tools. In the case of theory of computation, the tools in question that the authors identify are the mathematical notations that represent concepts and help students understand them. To proponents of DCog, then, knowledge of the course content is only part of the picture. Another vital part is knowledge of the tools required to put the course content to use, in this case, the understanding of mathematical notation. Thus the paper looks closely at how students use notation in their work (Knobelsdorf and Frede 2016).

Specifically, the authors looked at three student groups attempting to solve a problem on NP-completeness in a course titled Algorithms and Data Structures (AD). AD, which is taught every year at the University of Hamburg, is generally taken by second-year computer science majors, most of whom have already taken an introductory theory of computation course (the introductory course covers automata, logic, and some complexity theory) (Knobelsdorf and Frede 2016).

Even before the study began, it was already clear that students were having problems in AD, as evidenced by a high dropout rate and student reflections on their own difficulties. The year the study took place, nearly half of the enrolled students dropped out of the course or did not pass (Knobelsdorf and Frede 2016).

The authors decided not to rely on student self-evaluation for their data collection, instead opting to directly observe students at work in their groups. The course instructor, who cooperated with the authors in their study, asked for volunteer study groups to be part of the study. The authors observed one session of each of three groups that volunteered, though they were careful not to interfere (even when the members of these groups asked the observers for help with the assignment). They recorded the sessions and summarized the students' discussions and using a category system to classify the summaries (Knobelsdorf and Frede 2016).

All three groups (labeled A, B, and C) were working during the same week on the same NP-completeness problem, which asked students to show that if P = NP, then every non-empty, non-full language in P is NP-complete. The lectures had introduced a five-step method for proving a language to be NP-complete, and the instructor had taught all of the prerequisite concepts for understanding this method (Knobelsdorf and Frede 2016).

The correct proof can be written in two medium-length paragraphs, and the authors note its simplicity, explaining that it requires little creativity to develop. As such, the problem was only worth 2 points out of a total of 16 for the whole assignment. They remark, however, that the proof uses a few pieces of precise mathematical notation. Moreover, creating and understanding it requires familiarity with concepts related to complexity classes, sets, and functions (Knobelsdorf and Frede 2016).

The category system divides the students' time spent as follows: clarifying the assignment, clarifying concepts and inscriptions, requesting group members to act, clarifying an approach for a potential solution, developing a solution and writing it down, discussing theory of computation courses and related topics, searching for information and support, and other unrelated activities. The authors noticed that students overall spent the majority of their time seeking and providing clarification. The authors take this trend to indicate that the students are missing basic proficiency in theoretical computer science (Knobelsdorf and Frede 2016).

The authors then examine each group's practices in detail. In group A, one of the students brought up the five-step method, but all of the students struggled to figure out how to apply it, eventually resorting to searching the internet for help. Their research only brought up more questions as the students encountered unfamiliar methods and notation. They were uncertain about the notation that the

five-step method used and had to look on the internet for that as well. They put together arguments to prove that L, the arbitrary language they were dealing with, was nonempty and non-full. This was not what they were intended to prove but rather one of the explicitly stated assumptions of the problem. However, they were unable to formalize their arguments with mathematical notation and so decided to write down natural-language versions of them. Throughout, they ran into further clarification issues and had to seek help from outside sources. Moreover, two of the four group members barely participated in the discussion at all. In the end, group A scored 0.5/2 points for the problem (Knobelsdorf and Frede 2016).

Group B was more successful, with all four members actively engaged in the discussion. However, they struggled at first to find the five-step method, attempting to use material from the previous course such as Turing machines instead. They eventually found the five-step method and were able to apply it correctly, though they too were unable to write their solution down formally during the session. Group B scored 2/2 points for the problem (Knobelsdorf and Frede 2016).

Group C contained two members who had not taken the previous theory of computation course, and so the other two students spent much of their time explaining concepts to them. The group made some errors, attempting to prove a universal statement by example and trying to apply the concept of decidability to the problem. The members of group C were not aware of the five-step method, but they did not end up finding it at all and decided instead to figure out their own method using Turing machines. Group C's solution scored 0/2 points (Knobelsdorf and Frede 2016).

The authors discuss these results, noting that none of the groups was able to stick to one activity for very long, frequently switching from writing to clarifying to engaging in non-academic pursuits. As a result of this disorganized approach, the authors conjecture that the students likely struggled with other assignments as well, though they provide no data to confirm this conclusion (Knobelsdorf and Frede 2016).

The authors also argue that none of the three groups completely understood the proof they were trying to write, or even in some cases the basic principles of creating a proof in general. Group A even tried to prove an assumption of the problem, which means that they were also unable to effectively decipher a problem statement. Even group B only followed the instructions in the five-step method without ever comprehending how the five-step method worked, the authors claim. The authors were surprised, especially given that most of these students had already taken a theory of computation course (Knobelsdorf and Frede 2016).

Two main research questions were used to frame the results: 1) How do students use notation? 2) How do students know if their answers are right? The authors concluded that the students had little mastery over the notation. Though they were sometimes able to figure out the meaning behind notation given to them in lecture or on the internet, they could only rarely turn their own thoughts and arguments into effective notation. During discussion, the students agreed multiple times that they should write their solutions down, but admitted that they did not know how. The authors point out that students also missed the opportunity to write down their ideas as they were working to further their own understanding and that of the other members of their groups (Knobelsdorf and Frede 2016).

Students often tried to verify their solutions by searching on the internet. As with group A, this approach led to further confusion. Had they instead used material given to them in lecture, they would not have needed so much external guidance, but as the authors note, the students clearly did not have mastery over the lecture material (Knobelsdorf and Frede 2016).

To see if the problems experienced by these three groups were representative of the larger class, the authors examined the final scores of 52 student groups in total. Half of these groups scored below 1/2 points for the problem, demonstrating that the difficulties were widespread. The authors conclude by suggesting that the primary issue in teaching theory of computation is the lack not of student interest (as is widely believed), but rather of student proficiency with the methods and tools needed to practice theoretical computer science (Knobelsdorf and Frede 2016).

This paper makes an intriguing argument for reexamining the common wisdom about the education of theoretical computer science, and the authors are correct in identifying the need for more evidence in this area. This very identification, however, makes it somewhat surprising that the authors are so quick to jump from the small amount of data collected to conclusions about the field in general. They do acknowledge their intent to verify this data on a larger sample size, though the implication seems to be that this larger sample size will consist of more students studying at the University of Hamburg (Knobelsdorf and Frede 2016). This could be an issue, since difficulties experienced by students at a particular school might be the fault of that school's curriculum in particular and have little bearing on the field as a whole.

More importantly, while the authors are correct in stating that the students are missing proficiency with the methods of theoretical computer science, they fail to examine the underlying cause. The authors note briefly that students from all groups spent a significant amount of their time engaged in pursuits unrelated to the task at hand, but draw no conclusions (Knobelsdorf and Frede 2016). This practice,

however, is a clear indicator of a lack of interest, and it raises the question of why students are missing these basic proficiencies to begin with. It has already been established that students are not only failing to understand the lecture material but also either failing to pay attention or failing to take notes, because the instructor explicitly introduced the five-step method in lecture as an approach to demonstrate NP-completeness, and yet two out of three groups started work unaware it even existed, while the third needed to use the internet to look up the definitions of two of the most basic pieces of notation in formal language theory. It is therefore very possible that the reason students are lacking proficiency is because they find the material so uninteresting that they do not put in the effort to learn it.

While it is good practice to avoid speculation, the authors ignore evidence that has the potential to undermine their own conclusion. It may be misleading to suggest that education researchers should shift their focus, since without addressing the lack of student engagement, it could prove impossible to develop those missing proficiencies. Until further research can further clarify the cause of the lack of student engagement, it would be wise to take both of these factors (lack of interest and lack of proficiency) into account when designing new approaches to teaching theoretical computer science.

In "Learning Difficulties Experienced by Students in a Course on Formal Languages and Automata Theory," Pillay also delves into the causes behind students' dissatisfaction with theoretical computer science, using a third-year theory class as a case study. Data was collected in the form of students' answers to weekly problems as well as three tests over the course of the semester-long class. Pillay observed difficulties experienced by the students throughout the class. Students struggled especially with converting between different representations of the same language and applying the more mathematical theorems such as the pumping lemma (Pillay 2009).

Pillay attributes the bulk of these issues to a lack of problem-solving skills and suggests several constructive solutions, such as a greater emphasis on group work and additional stages of feedback from instructors, but she does not explain how she has reached this conclusion (Pillay 2009). Examining the submitted answers to tests makes it clear where students are having difficulties, but it does not necessarily demonstrate the causes of these difficulties, so it is unclear where Pillay is getting this information from. Once again, therefore, other explanations are plausible as well. Nonetheless, Pillay's assertion that students lack proficiency in problem-solving ties into the conclusions of (Knobelsdorf and Frede 2016), highlighting the improvement proficiency as a clear area for further exploration, though not the only one.

In "Problem-Based Learning of Theoretical Computer Science," Hamalainen concurs with other researchers that theoretical computer science is particularly challenging

to teach, pointing out the striking difference in the style of reasoning required when compared to other computer science course material: where students are taught algorithmic thinking in other courses, theory of computation requires abstract, logical thinking, which is more characteristic of mathematics courses. However, as Hamalainen remarks, most students are uninterested in math (Hamalainen 2004), which makes it difficult for students to acquire the necessary proficiency in mathematical reasoning. Thus, as these papers have shown, students are entering theoretical computer science courses missing the proficiencies required to succeed in these courses and already prejudiced against the course material due to its mathematical foundations and its perceived lack of practical applications. But critically, the courses themselves are often failing to address these issues. New methods and tools must be developed if computer science departments are to overcome the inherent difficulties of teaching theoretical computer science.

2.2 Pedagogical Methods

Sigman lays out some of the ideas that have been proposed to confront these challenges. Some approaches attempt to place emphasis on the applications of the automata to provide students with motivation for the course's content. Others attempt to tackle what students view as the overly abstract mathematical approach to theory of computation by using various tools that can simulate theoretical machines or otherwise make the material more concrete for the students (Sigman 2007). It is worth noting that some, such as Armoni and Verma, advocate for retaining the abstraction, arguing that it is essential for students to learn the course material in the mathematical context in which it is currently taught (Armoni et al. 2006). Brookes mentions an indirect approach to teaching theory by incorporating concepts from theoretical computer science into a more practical course built around learning a language such as XML (Extensible Markup Language) (Brookes 2004).

2.2.1 The Moore Method

Some approaches for fixing unresolved issues with theoretical computer science education deal primarily with altering pedagogical techniques rather than the course content. Sigman recommends the Moore Method, which is the collective term for a set of related practices focusing on allowing students to drive the learning process (Sigman 2007). Theory of computation is typically taught in lecture format, in which the instructors explain the material at the board, handing out problem sets every week for the students to demonstrate their understanding of the lecture content (Knobelsdorf et al. 2017). Yet the authors of "Active Learning Increases Student Performance in Science, Engineering, and Mathematics" find that the traditional

lecture format of many college classes can be ineffective, resulting in high rates of failure among the students (Freeman et al. 2014). In contrast, with the Moore Method, students first learn the course material by working on the problem sets in collaboration with other students, and the course meetings are reserved for student presentations about the problems they've solved (Sigman 2007).

Sigman advocates for the Moore Method in the context of theoretical computer science because it has been shown to be effective helping students learn how to develop proofs (though he does not identify where or by whom this has been shown). To test this proposal, Sigman organized a course on theoretical computer science using the Moore Method at Drury University. He evaluates the course by studying the test results of the four enrolled students and appraising the quality of their presentations (Sigman 2007).

Overall, he declares the course a success. The presentations were particularly effective, and they only improved throughout the class. Even the students who entered the course with limited experience in higher-level mathematics were by the end able to create and demonstrate complicated proofs for their presentations. Sigman also argues that the positive effects of the Moore Method went beyond the easily quantifiable. He observes that the students in the class developed more confidence as the course went on, even gaining the mature attitude that it was okay to present incorrect solutions so long as they learned something from the process (Sigman 2007).

The results of Sigman's study are impressive. Not only did students achieve proficiency with formal mathematics, but at least one of them became interested in the subject as well. Sigman notes that one of the four students decided during the course that he wished to become a mathematics minor despite already being a junior, and filled his schedule for his senior year with as many math courses as he could even after learning that it was impossible to complete the minor so late (Sigman 2007). The Moore Method therefore has the potential to address both of the main difficulties with teaching theoretical computer science identified above.

Sigman acknowledges the limited scope of his study. Not only does the small class size make the results anecdotal at best, but it is also particularly relevant given the style in which the course was taught (Sigman 2007). When student participation forms the bulk of the class, a change in the class size will drastically affect the student experience. Sigman asserts that this format could work with up to fifteen or twenty students, and he suggests modifications that could allow it to work with even more (Sigman 2007), though these modifications have yet to be tested. Sigman's application of the Moore Method to theoretical computer science is promising, but it

is clear that further testing must be done before it can be successfully generalized to work with most theoretical computer science classes, if that is even possible at all.

2.2.2 Problem-Based Learning

Hamalainen argues for a similar approach. Problem-based learning is comparable to the Moore Method in that students are primarily responsible for their own learning, working on problems with limited prior guidance on the material at hand from the instructor. The version Hamalainen uses, however, is more of a hybrid, as it includes both student problem-solving sessions and lectures, and it lacks the in-class presentation component of the Moore Method (Hamalainen 2004).

Hamalainen tested out problem-based learning in the context of theoretical computer science in a course at the University of Joensuu. Over the ten weeks of the course, the students were to learn automata and formal languages by a seven-step method of figuring out the material by solving the problems given to them, though there was also a much smaller control group for a more traditional approach to the class. Overall, the problem-based learners had a much lower dropout rate (Hamalainen 2004). This suggests, perhaps, that problem-based learning helps build student interest by letting them engage more fully with the material. However, there was no significant difference between the grades of the control group and those of the experimental group (Hamalainen 2004), which implies that further testing and possibly refinement are needed to demonstrate a link between problem-based learning and improved student proficiency. As Hamalainen notes, though, the control group was too small to draw conclusions with any certainty (Hamalainen 2004), so the issue may lie in the experimental design rather than the method being tested.

2.2.3 Cognitive Apprenticeship

The authors of "Teaching Theoretical Computer Science Using a Cognitive Apprenticeship Approach" emphasize the importance of the instructor-student relationship. The practice of cognitive apprenticeship focuses on various opportunities for the student and the instructor to collaborate: modeling and articulation, where the teacher provides exposition on the course material; coaching, where the teacher assists the students in putting the material to use; and scaffolding, where the teacher creates methods and structures to aid the students in applying the material. The authors modified this framework to better suit theoretical computer science classes, but the overall pedagogy remains much the same. (Knobelsdorf et al. 2014).

The authors used a cognitive apprenticeship approach in an introductory theory course usually taken by 150 to 300 students at a time. The authors believe that coaching was already accomplished by feedback on homework and exams. To facilitate modeling and articulation, the authors added weekly class sections for students to ask questions and observe the instructors working through the problem-solving process. To help with scaffolding, the authors created exercises for the students to work on together in order to prepare them for similar questions on the homework assignments. The course in question had a high rate of failure prior to the authors' alterations, reaching as high as 59.5% in 2011. After introducing cognitive apprenticeship, however, the authors observed failure rates below 10%. The paper concludes that the study has demonstrated the efficacy of the cognitive apprenticeship approach in theoretical computer science education (Knobelsdorf et al. 2014). While other variables may have played a role in the reduction of the failure rate, the changes were so dramatic that it is clear that the focus on cognitive apprenticeship has been beneficial.

Clearly, pedagogical methods can improve student experiences, and in some cases, grades, in theoretical computer science classes, but the widely acknowledged difficulty of the material itself remains a barrier. Therefore, changes in pedagogy alone may be insufficient to overcome the obstacles to teaching theory. It is critical, therefore, to consider whether there are computational tools that can assist students in such courses.

2.3 Theorem Provers

The authors of "Theorem Provers as a Learning Tool in Theory of Computation" also affirm the importance of theoretical computer science. They too recognize the difficulty of teaching it, pointing out the high dropout rates and low grades that afflict such courses. They propose, however, that an important reason for this difficulty lies in the proof-based nature of the course (Knobelsdorf et al. 2017).

2.3.1 The Coq Proof Assistant

The authors therefore put forward a new application for Coq, a mathematical theorem prover. Used primarily to check complex proofs for use in papers or to aid students in upper-level graduate classes, theorem provers have been around since the 1960's. Coq, created in France in the 1980's, is based on type theory. Type theory is an area of formal logic reminiscent of strongly typed programming languages in that, similar to variables in a language like Java, mathematical terms are assigned types (such as *natural number* or *function from the natural numbers to the natural*

numbers). Type theory is especially useful in the context of a theorem prover, as a proposition (a logical statement to be proven) can be seen as a type, with all proofs of that proposition acting as terms of that type (*NLab*). This makes checking proofs for correctness a problem of checking types, algorithms for which already exist for other purposes, such as compilers for programming languages (*Inside Coq's Type System*). Coq's integrated development environment (IDE) looks similar to those used for other programming languages. The user can create proofs by adding tactics to a sequence. Tactics are Coq's system for backward reasoning, each tactic taking what needs to be shown and subdividing it into lesser propositions such that proving each of these lesser propositions is sufficient to prove the original statement (Paulin-Mohring 2011). Eventually, this process produces a completed proof that the theorem prover then verifies (see Fig. 2.1). Additionally, Coq aids the user during the proof's construction, showing information about what remains to be shown and checking each step for correctness along the way. The authors posit that these features make using Coq akin to programming (Knobelsdorf et al. 2017).

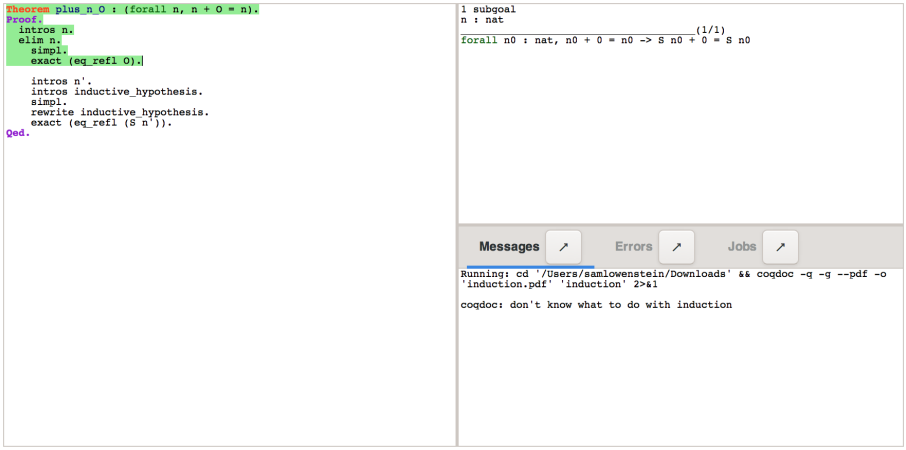

Fig. 2.1: An inductive proof that for all n, n + 0 = n in Coq, just after executing the base case (*The Coq Proof Assistant: A Tutorial*). The upper-right pane lists all current variables and their corresponding types (*The Coq Proof Assistant*).

As the dissimilarity to type of coursework that undergraduate students are accustomed to could be a significant cause of the lack of interest in theory, this resemblance is reason to believe that Coq may be a useful tool in education. Moreover, constantly checking for correctness means that students can learn immediately if there are problems in their work, giving them time to fix issues before turning work in. The paper also asserts that Coq's formalization is more consistent than that usually taught in proof-based classes and therefore helps students understand what is required of them (Knobelsdorf et al. 2017). Coq could therefore serve to aid students in filling in the gaps in their mathematical proficiency.

However, like many other theorem provers, Coq was created for experts to use, not undergraduate students. Thus the authors had to make certain adaptations. The

most important adjustment was based on the principle of information hiding. By restricting the functionality of the theorem prover to a limited set of tactics that work like building blocks, the authors hoped to make Coq more accessible and the type theory underlying the tool nonessential to its use (Knobelsdorf et al. 2017).

2.3.2 Testing Coq

To test the newly adapted tool, the authors set up an undergraduate course on formal logic and data structures at the University of Hamburg in Germany. The course was full-time and lasted two weeks. Students were asked to use the authors' version of Coq to complete some of the proofs on homework assignments and eventually the final exam (Knobelsdorf et al. 2017).

The authors used a category system to analyze the kinds of problems students encountered while working on their assignments and their ability to tackle these problems. Chief among these categories were T-problems, which students were only able to resolve with help from the instructors or tutors, and S-problems, which they were able to resolve on their own. Using this category system, the authors observed that while the students initially experienced difficulties using Coq and required additional help from the instructors, by the end of the course they had become proficient with the theorem prover and were able to work past these difficulties on their own. Additionally, they initially struggled to create proofs but by week two were experiencing only a few problems in that area (Knobelsdorf et al. 2017).

All 14 students who took the final exam received passing grades on it. The problems on the final exam included both proofs assisted by Coq and proofs that could only be solved with paper and pencil. Students overall performed well on problems for which they could use Coq. However, they received lower scores on problems that did not permit the use of the theorem prover, exhibiting the same kinds of errors that they had run into during the early days of the course (Knobelsdorf et al. 2017).

By running this course, the authors intended to answer three questions about Coq. 1) What problems do students have using Coq? 2) Do students like using Coq? 3) Does using Coq help students write proofs? In addition to the work the students turned in, the results of the exam, and the category distribution of the problems students encountered, the authors evaluated the course with three surveys, of which two dealt with the students' opinions on using Coq (Knobelsdorf et al. 2017).

To answer question 1, the authors conclude that their version of Coq was easy to learn and did not cause significant problems for students. To answer question 2, the authors determine based on the surveys and their observations of the students

that they found more satisfaction in using Coq than they did without it. To answer question 3, they conclude from the decline in T-problems and S-problems over the course of two weeks that Coq did in fact help. Limiting the students' options when writing proofs gave them a chance to get better at particular approaches and prevented confusion. The paper ends with the assertion that the new version of Coq can be an effective tool in the education of theoretical computer science (Knobelsdorf et al. 2017).

While these conclusions are encouraging, the data may not fully support them. Even though the students performed well when they were allowed to use Coq, their lack of progress in proving theorems without its aid suggests that the answer to question 3 may not be quite so satisfying. The authors note the disparity between the students' desire to use Coq (and their higher grades when doing so) and their reluctance to write proofs without Coq's aid (and their comparatively poor performance) (Knobelsdorf et al. 2017). This disparity raises further questions about what the students were actually learning by using a theorem prover.

Yes, the students must have been learning something, as their ability to prove theorems with Coq improved, but perhaps what they were learning was simply how to use Coq well. They may have been training themselves to rely on Coq to guide them, developing bad habits and failing to learn the meaning behind the proof tactics Coq supplied for them. The authors at least claim (without displaying the relevant evidence) that pure trial and error would have been insufficient for students to complete the Coq assignments given to them (Knobelsdorf et al. 2017). Even if that is true, though, it is one thing to learn the strategies for employing a tool and quite another to learn from that tool how to succeed without using it.

The authors suggest that perhaps they did not focus enough on how to facilitate the transfer of skills between using a theorem prover and writing proofs with paper and pencil. They plan on looking into cultivating this transfer in future experiments, though the specifics are not made clear in their paper (Knobelsdorf et al. 2017). What is clear is that there needs to be some regulator in place to monitor students' progress with Coq and prevent them from becoming dependent on the use of a theorem prover.

It is also important to remember that this experiment has not yet demonstrated the use of Coq in an actual theory of computation course. Though the authors believe future work can demonstrate Coq's efficacy in that context, their course in this paper only got as far as logic and data structures (Knobelsdorf et al. 2017). These areas cover many of the basics of proof-writing, so they are good choices for an initial investigation. However, formal languages and computational complexity would introduce new challenges for anyone attempting to adapt Coq (or a similar program)

for use in education, since proofs would have to center around the operation of various automata. Not only would this require additional work from the adapters, but it could also potentially cause different issues for students attempting to learn and use the theorem prover.

Furthermore, only sixteen students attended the entirety of the class (Knobelsdorf et al. 2017), which is a small sample size, especially given that the course lasted only two weeks. Though there were 30 hours of class time each week (Knobelsdorf et al. 2017), the short time span provides little evidence with regards to students' retention of the material in the long term. Most of these students had also previously taken an introductory course covering at least some formal logic (Knobelsdorf et al. 2017). This paper therefore does not show whether it is viable to use Coq in a course for students who have no experience in logic, which could present problems for colleges that put students straight into the theoretical computer science track.

2.4 JFLAP

The authors of "Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP" argue that an important reason for the lack of student proficiency in theoretical computer science is the difficulty of visualizing the concepts and constructions used in these classes. The standard notations for automata can be difficult to create and parse, which makes it even more challenging to interact with the automata and simulate their operation (Procopiuc et al. 1996).

2.4.1 The Java Formal Languages and Automata Package

The authors propose the Java Formal Languages and Automata Package (JFLAP), one of the leading tools in the education of theoretical computer science, as a solution to this problem. JFLAP allows the user to create easily comprehensible visual representations of automata and run them on various inputs (see Figs. 2.2, 2.3). Working through an example of a potential application for JFLAP in the classroom, the authors suggest that JFLAP will help students understand the meanings of abstract representations in automata theory. However, the paper's proposals are entirely theoretical, with no evaluation of the efficacy of the tool. (Procopiuc et al. 1996).

Three years later, the authors of "Using JFLAP to Interact with Theorems in Automata Theory" (including Susan H. Rodger, one of the authors of "Visualization and Interaction") present a set of modifications to JFLAP that allow the user to work through proofs of conversion between representations of languages. For instance,

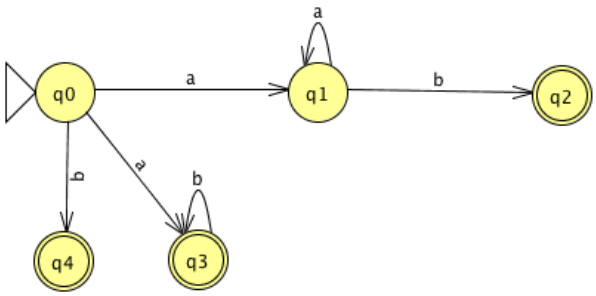

Fig. 2.2: The NFA recognizing $a^*b \cup ab^*$ (Kelley 1995), created in JFLAP (*Java Formal Languages and Automata Package 7.1*).

after creating an NFA using JFLAP's interface, one can command the tool to turn that NFA into an equivalent DFA, watching the process unfold step by step (see Figs. 2.4, 2.5). Similar transformations are supplied for other representations of regular languages as well as those for context-free languages. The authors again recommend the use of JFLAP in education, claiming that it has been an effective tool in classes at Duke University during the preceding years. However, they still do not provide a study or other formal evaluation to corroborate these findings (Gramond and Roger 1999).

After similar explanations of the functionality of JFLAP, the authors of "A Visual and Interactive Automata Theory Course with JFLAP 4.0" (including Susan H. Rodger) provide some insight as to the nature of one course at Duke using this tool. The curriculum puts JFLAP front and center, with in-class demonstrations of its mechanics and in-class automaton building in collaboration between the students and the instructors. The class also puts the modifications made by (Gramond and Roger 1999) to use, helping students visualize the ideas behind proofs by having JFLAP work through the steps of conversions. Students are also allowed to use JFLAP outside of the classroom to solve or format problems for homework assignments. The authors also provide some measure of evaluation, giving the results of a survey distributed to the students. Overall, students found JFLAP easy to use, and most preferred using JFLAP to solving problems with only paper and pencil (Cavalcante et al. 2004). It remains unclear, however, if using JFLAP is improving their performance in the course.

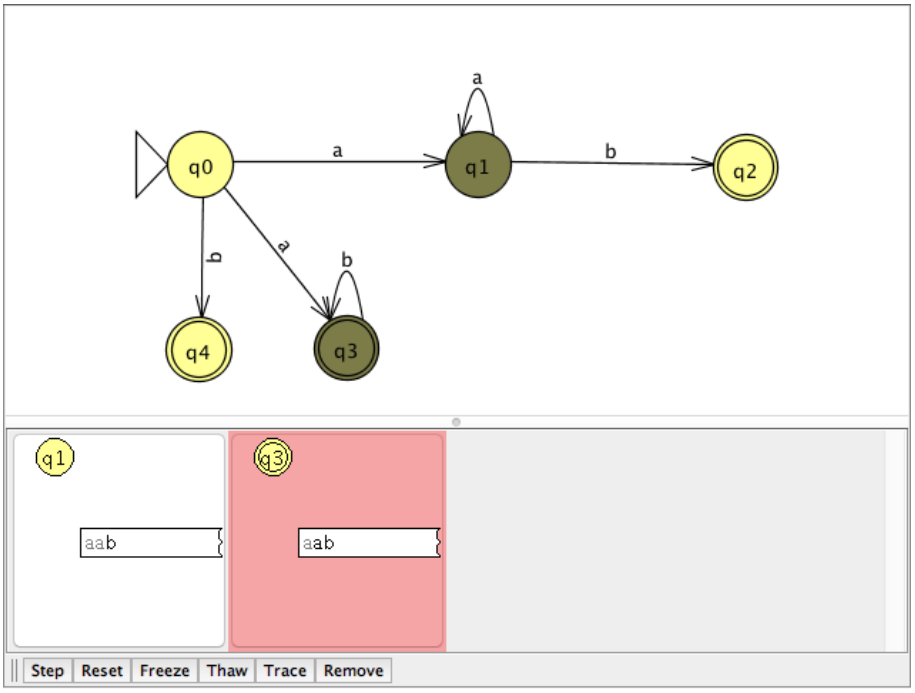

Fig. 2.3: Simulating the NFA on input *aab*. Two paths can be seen, one leading through q1 that is still running, and one that has terminated at q3 (*Java Formal Languages and Automata Package 7.1*).

2.4.2 Testing JFLAP

In 2009, however, Rodger attempts to answer that question with a two-year study in "Increasing Engagement in Automata Theory with JFLAP." Fourteen universities participated in the study, which applied JFLAP to undergraduate theoretical computer science coursework. This time, the authors collected extensive information to evaluate the results of the experiment, using tests, surveys, and interviews. After the first year of the study, a workshop was held to discuss the results of the first year and identify steps for improvement. The feedback from students and faculty casts some light onto the efficacy of JFLAP as an educational tool (Rodger et al. 2009).

Most of the students who used JFLAP did not feel that it improved their grade significantly, with only 36% of respondents agreeing that they would not have done as well without JFLAP. However, the authors note that many of the students who did use JFLAP used it in less than half of the course, which could mean that they did not use JFLAP enough for it to make a noticeable difference. But even those who did not find that JFLAP improved their grade generally did not find it to be detrimental. Students expressed almost unanimously that JFLAP did not take time away from other study activities. Results from the workshop indicate overall satisfaction amongst faculty, and the broad range of participants highlighted the varied potential uses of JFLAP in the classroom. The tests administered to the

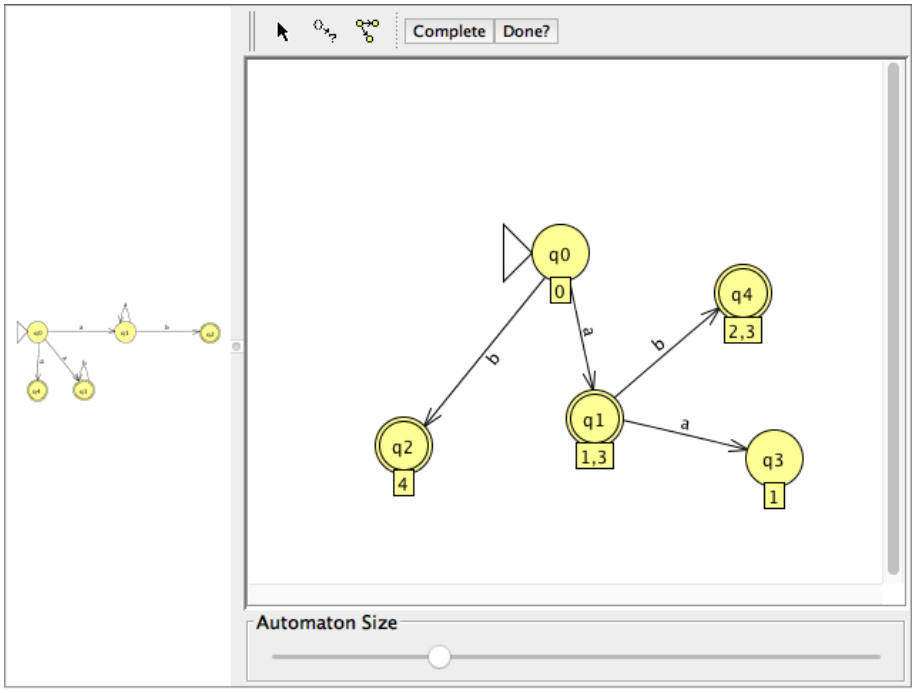

Fig. 2.4: An intermediate stage in converting the NFA to a DFA in the right pane, with the original NFA still showing in the left pane (*Java Formal Languages and Automata Package 7.1*).

students demonstrate that the students were learning, though they do not show whether that learning can be at all attributed to JFLAP (Rodger et al. 2009).

For the second year, the authors implemented some suggestions from the workshop to change the evaluation metrics and added a control group in which JFLAP was not used. In the survey at the end of the course, most students believed that JFLAP made the course more engaging and enjoyable, and that it helped them understand course concepts more easily (Rodger et al. 2009). However, the methodology of the paper necessarily introduces uncertainty into its findings. Students' perceptions of coursework are not dependent solely on the tools used in their classes. Additional variables, including the course structure, the teaching style of the instructors, and even the students' commitment and motivation, will vary from class to class, which makes it difficult to identify with any certainty the causes of the study's results.

Furthermore, the control group's performance was not significantly different from that of the other groups (Rodger et al. 2009). This result casts doubt on the value of the survey responses. While the authors conclude the paper on an optimistic note, stressing the positive results of the experiment, it is unclear if JFLAP actually did help students perform better in the course. The increased enjoyment is an accomplishment in and of itself, but further improvements may need to be made if JFLAP is meant to help students develop proficiency with theoretical computer science.

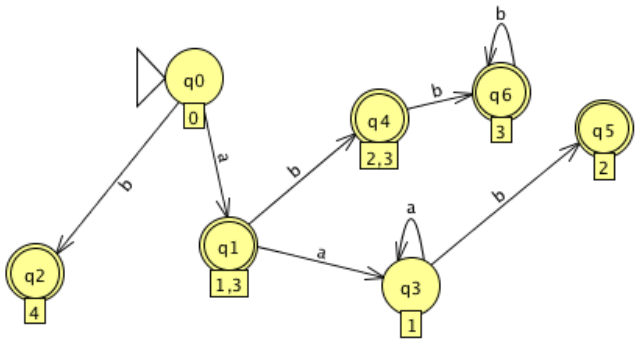

Fig. 2.5: The NFA fully converted to a DFA (*Java Formal Languages and Automata Package 7.1*).

2.4.3 Other Visualization Tools

A few other visualization tools to support theoretical computer science education are worth noting. Agreeing that a primary obstacle to understanding the material is the difficulty of visualizing it, Chuda has developed animations to show students how automata function in detail. While these animations are helpful, Chuda proposes that they are best suited for an online course (Chuda 2007). After all, they are simply a more time-efficient version of simulating at a blackboard, and furthermore, they cannot easily be altered to answer student questions or test out additional inputs. Lesser-known tools similar to JFLAP also exist, including THOTH (Garcia-Osorio et al. 2008) and AtoCC (Hielscher and Wagenknecht 2006), but these tools fill the same niche as JFLAP and have been tested less.

2.5 Gamification

While the developers of some tools, such as Coq and JFLAP, seek to bolster the traditional aspects of theoretical computer science classes, other researchers suggest a new method of piquing student interest: integrating games into the curriculum.

2.5.1 Game-Building

The authors of "Learning by Game-Building" propose that having students create games as class assignments improves both their motivation and the quality of their work. They begin by establishing that there is a clear demand for assignments based around game design, offering students an assignment with a choice between

replicating an existing game and designing their own in a preliminary study. Of the students participating, 55% chose to design their own game. To test the extent of these students' motivation, the authors allowed them to submit their newly designed games to a competition, and half of them did so (Korte et al. 2007).

The main thrust of the paper applies game-building to a theoretical computer science course. During the part of the course focusing on FSA's, each student was asked to build a game in which an FSA kept track of information about the game state, allowing the game to respond to players' decisions. Certain requirements were put in place to make sure that the FSA's created were sufficiently complex to assess the students' understanding of the subject material. As the course moved on to Turing machines, students were then asked to design more advanced games in which a five-tape Turing machine kept track of the game state and allowed for greater complexity (Korte et al. 2007).

The authors note that 87% of the students finished their games and that the weaker students were particularly engaged with the assignments. Some of the stronger students were less engaged and wished for a more traditional approach to the course (Korte et al. 2007), which has also been observed in Kurt Squire's attempt to use games in education (Squire 2005). The authors conclude that game-building is a promising method for teaching theoretical computer science, while remaining careful to acknowledge that it may not work for all students (Korte et al. 2007).

While it is important to ensure that new methods work for all students involved, it is appropriate for the authors to focus primarily on improving the experiences of students who are struggling with the material, and thus introducing games to theoretical computer science education may be a fruitful area of investigation. However, it should be noted that a full 45% of students in the preliminary study preferred not to have to create their own games (Korte et al. 2007). Game design can be challenging, and it requires skills that not all computer science students will necessarily have.

2.5.2 The Pumping Lemma Game

Incorporating games into the curriculum could still be beneficial for struggling students, though. JFLAP, for instance, is already partially gamified. JFLAP's Regular Pumping Lemma utility takes the user to a game in which the user and the computer go back and forth, each attempting to attack or defend a language's regularity by using the pumping lemma. The user may select the language and then choose whether to attack or defend it (*Java Formal Languages and Automata Package 7.1*).

The Regular Pumping Lemma. If L is an infinite regular language, then there exists a constant n with the property that, if w is any string in L of length greater than or equal to n, then we may write $w = uvx$ in such a way that $uv^ix \in L$ for all $i \geq 0$, with $|v| \geq 1$ and $|uv| \leq n$ (Kelley 1995).

Since the failing to meet the conditions of the pumping lemma means that a language is not regular, the attacking player is trying to find an example for which the condition does not hold, while the defending player is trying to find conditions that prevent the attacking player from doing so. Since disproving a universal statement requires only a single counterexample, the attacking player is allowed to fix all variables with universal quantifiers in the pumping lemma. However, proving an existential statement requires only a single example, which means that the defending player is allowed to fix all variables with existential quantifiers in the pumping lemma. If the resulting decomposition fails the conditions of the pumping lemma, then the attacking player wins (see Fig. 2.6). If, however, the resulting decomposition can be pumped, the defending player wins (*Java Formal Languages and Automata Package 7.1*). Note that a victory for either player does not constitute a proof that the language is regular or nonregular, as a universal statement cannot be proven by example and an existential statement cannot be disproven by counterexample.

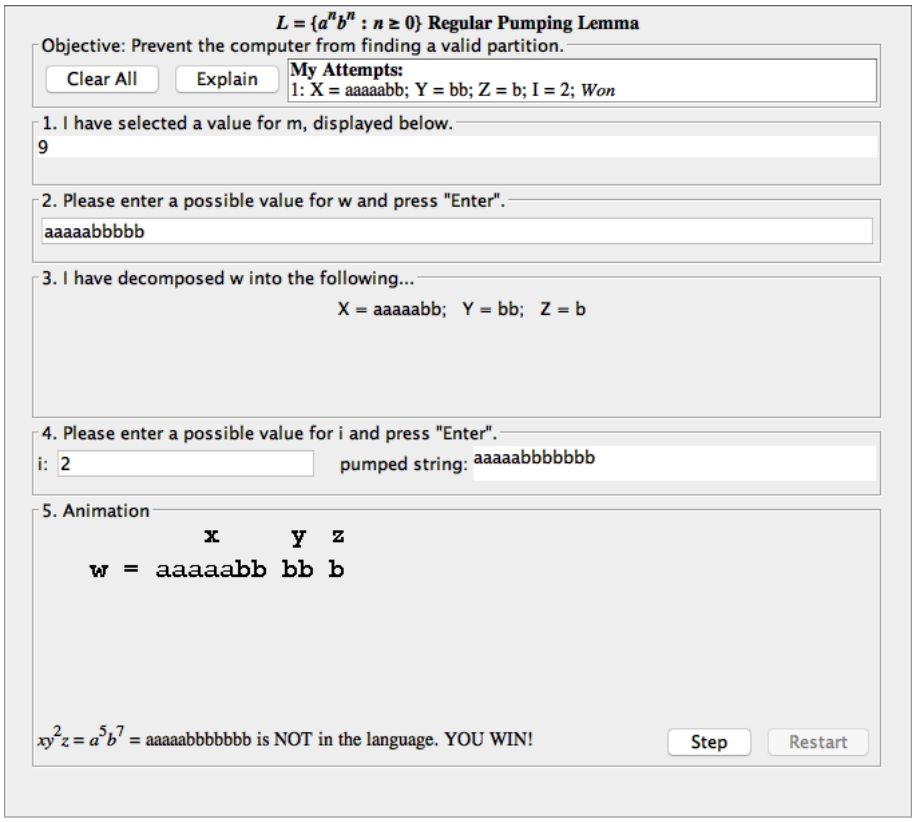

Fig. 2.6: A victory for the attacking human player in the Regular Pumping Lemma game (*Java Formal Languages and Automata Package 7.1*).

This game may present a step in the right direction. Complex theorems with multiple quantifiers can be challenging and uninteresting for students to parse, and students may struggle to figure out what is required to prove or disprove that these quantifiers hold. Turning the lemma into an adversarial game not only engages the students but also helps them comprehend which "side" the burden of proof lies on for each variable in the lemma. Even though playing the game does not formally prove anything about a given language, it allows students to interact with the mechanism of the lemma, potentially preparing them to move on to complete proofs. JFLAP has a similar version of the same game for the Context Free Pumping Lemma as well (*Java Formal Languages and Automata Package 7.1*).

2.5.3 Gamifying JFLAP

The Pumping Lemma games raise the question of whether JFLAP can be further gamified, which Fabio Bove attempts to answer in his bachelor's thesis. Bove builds and advocates for a game implemented using JFLAP. The game gives the user a formal language represented in one way and asks the user to write the language using a different representation. For instance, the user might be given an NFA and asked to construct an equivalent DFA. The game then internally converts the two representations into automata and compares them, providing the user with a message if the conversion was successful or a word accepted by one representation but not the other if the conversion was unsuccessful. This comparison is accomplished by the Graphical Tool for Omega-Automata and Logics (GOAL) (Bove 2015).

The game allows the user to pick the types of representation used for both the initial state and their eventual conversion. The user may then select a level (a single instance of a conversion to be made). The levels are ordered, with each level accessible if and only if either it is the first level in the set or the user has already solved the previous level (Bove 2015). This gives the game a sense of progression, rewarding the user's work with further options. However, it is important to ensure that this progression is meaningful such that each new level introduces new concepts or expands on concepts introduced in previous levels. If the levels are ordered randomly or without planning, some may become frustratingly difficult or too easy, and either would hamper the user's enjoyment and ability to learn. Bove does not address this issue in his thesis, focusing primarily on the mechanics of the game rather than the design.

Additionally, it is unclear if the game's feedback system is sufficient to help struggling students. While producing a string that distinguishes between two different languages demonstrates decisively that there is a problem in the user's conversion, it may give little indication of how to solve it or even where the issue is. As demon-

strated by (Knobelsdorf and Frede 2016), students in theoretical computer science classes often miss key elements of the course material altogether, so a student tasked, for instance, with converting an NFA to a DFA, may be unaware of the powerset construction entirely. However, if the student fails in attempting the conversion, the game's feedback system may eventually guide them to a solution to the level in question, but no number of distinguishing strings can teach the powerset construction itself. An ideal version of Bove's game should be able to do just that, providing hints that are more constructive than simply disproving the user's incorrect answer.

3 Conclusion and Future Work

New tools must be used to address the issues students are experiencing in theoretical computer science courses. Some existing tools show promise, such as the theorem prover Coq and the visualization tool JFLAP, but these tools are still incomplete. In the case of Coq, for instance, this problem is the lack of transfer to unguided proof composition. JFLAP helps students visualize concepts in automata theory, making it easier for students to understand them, but it can still be improved to heighten student engagement with the material. The technique of gamification provides a new avenue for the improvement of these tools.

In particular, I propose a project similar to the gamification of JFLAP in (Bove 2015), but with an adaptive hint system. As the game should be able to help students learn concepts they do not yet understand, the hint system should include three levels of hints: those that already exist (i.e. strings that break the conversion), hints that suggest at an approach for the problem, and in-depth hints that provide a brief tutorial on a similar problem. The latter two levels can still potentially be automatically generated, as these hints can be very similar or even the same across similar types of problems. For instance, because the powerset construction can solve any NFA to DFA conversion problem, hints for all problems in that category can simply point towards the powerset construction, with the second level providing an idea or a basic framework and the third level providing a step-by-step guide. This will not pose a significant obstacle, as JFLAP already contains tools for working through conversion proofs. Additional hints tailored to the challenges of a specific problem can be manually coded in as needed. The conversion game already presents the option for the user to create their own levels, and it would be straightforward to add to that feature the ability to write custom hints for new levels.

The adaptation should consist of one or more systems for determining which level of hint to provide. As a representation could be substantially very close to the language it is intended to represent yet still reject all strings in that language (the accept state in the wrong place, for example), the hint system should not use similarity to the intended language as a criterion. Instead, possible criteria include: amount of time spent on the problem, performance on similar problems, and similarity to other users' answers to the same problem, using a machine learning model. An alternative system could instead allow users to unlock more informative hints by

spending larger quantities of some in-game resource, such as a score for the level. Machine learning can also be used in cases where there are different hints of the same level, making decisions about which hints will be most valuable to the current player based on the performance of past players who have used those hints.

Moreover, this extension should use a modern platform such as a smartphone. Time permitting, the program should be made into a full game, with a system of incentive or progression to reward students who continue using it.

Bibliography

Armoni, Michal, Susan Rodger, Moshe Vardi, and Rakesh Verma (2006). “Panel Proposal: Automata Theory: Its Relevance to Computer Science Students and Course Contents”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer science Education*.

Bove, Fabio (2015). “A System for Checking Equivalence of Representations of Regular Languages”. MA thesis. Technical University of Munich.

Brookes, Wayne (2004). “Computing Theory with Relevance”. In: *Proceedings of the Sixth Australasian Conference on Computing Education*.

Cavalcante, Ryan, Thomas Finley, and Susan H. Rodger (2004). “A Visual and Interactive Automata Theory Course with JFLAP 4.0”. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*.

Chuda, Daniela (2007). “Visualization in Education of Theoretical Computer Science”. In: *Proceedings of the 2007 International Conference on Computer Systems and Technologies*.

Freeman, Scott, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth (2014). “Active Learning Increases Student Performance in Science, Engineering, and Mathematics”. In: *Proceedings of the National Academy of Sciences*.

Garcia-Osorio, Cesar, Inigo Mediavilla-Saiz, Javier Jimeno-Visitacion, and Nicolas Garcia-Pedrajas (2008). “Teaching Push-Down Automata and Turing Machines”. In: *ACM SIGCSE Bulletin*.

Gramond, Eric and Susan H. Roger (1999). “Using JFLAP to Interact with Theorems in Automata Theory”. In: *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*.

Hamalainen, Wilhelmiina (2004). “Problem-Based Learning of Theoretical Computer Science”. In: *34th Annual Frontiers in Education*.

Hielscher, Michael and Christian Wagenknecht (2006). “AtoCC: Learning Environment for Teaching Theory of Automata and Formal Languages”. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*.

Kelley, Dean (1995). *Automata and Formal Languages: An Introduction*. Prentice-Hall.

Knobelsdorf, Maria and Christiane Frede (2016). “Analyzing Student Practices in Theory of Computation in Light of Distributed Cognition Theory”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*.

Knobelsdorf, Maria, Christoph Kreitz, and Sebastian Bohne (2014). “Teaching Theoretical Computer Science Using a Cognitive Apprenticeship Approach”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*.

Knobelsdorf, Maria, Christiane Frede, Sebastian Bohne, and Christoph Kreitz (2017). “Theorem Provers as a Learning Tool in Theory of Computation”. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*.

Korte, Laura, Stuart Anderson, Helen Pain, and Judith Good (2007). “Learning by Game-Building: A Novel Approach to Theoretical Computer Science Education”. In: *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*.

Paulin-Mohring, Christine (2011). “Introduction to the Coq Proof-Assistant for Practical Software Verification”. In: *LASER*.

Pillay, Nelisha (2009). “Learning Difficulties Experienced by Students in a Course on Formal Languages and Automata Theory”. In: *ACM SIGCSE Bulletin*.

Procopiuc, Magdalena, Octavian Procopiuc, and Susan H. Roger (1996). “Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP”. In: *Technology-Based Re-Engineering Engineering Education Proceedings of Frontiers in Education FIE’96 26th Annual Conference*.

Rodger, Susan H. *Java Formal Languages and Automata Package 7.1*.

Rodger, Susan H., Eric Wiebe, Kyung Min Lee, Chris Morgan, Kareem Omar, and Jonathan Su (2009). “Increasing Engagement in Automata Theory with JFLAP”. In: *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*.

Sigman, Scott (2007). “Engaging Students in Formal Language Theory and Theory of Computation”. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*.

Squire, Kurt (2005). “Changing the Game: What Happens when Video Games Enter the Classroom”. In: *Innovate: Journal of Online Education*.

The Coq Proof Assistant.

Tucker, Allen B., Charles F. Kelemen, and Kim B. Bruce (2001). “Our Curriculum Has Become Math-Phobic!” In: *ACM SIGCSE Bulletin*.

Websites

Casteran, Pierre. *Inside Coq’s Type System*. URL: `https://www.labri.fr/perso/casteran/CoqArt/Tsinghua/C5.pdf`.

Nahas, Mike. *The Coq Proof Assistant: A Tutorial*. URL: `https://coq.inria.fr/tutorial-nahas`.

NLab. URL: `https://ncatlab.org/nlab/show/type+theory`.