How Can Automatic Feedback Help Students Construct Automata?

LORIS D'ANTONI, University of Pennsylvania
DILEEP KINI, University of Illinois at Urbana-Champaign
RAJEEV ALUR, University of Pennsylvania
SUMIT GULWANI, Microsoft Research
MAHESH VISWANATHAN, University of Illinois at Urbana-Champaign
BJÖRN HARTMANN, UC Berkeley

In computer-aided education, the goal of automatic feedback is to provide a meaningful explanation of students' mistakes. We focus on providing feedback for constructing a deterministic finite automaton that accepts strings that match a described pattern. Natural choices for feedback are binary feedback (correct/wrong) and a counterexample of a string that is processed incorrectly. Such feedback is easy to compute but might not provide the student enough help. Our first contribution is a novel way to automatically compute alternative conceptual hints. Our second contribution is a rigorous evaluation of feedback with 377 students. We find that providing either counterexamples or hints is judged as helpful, increases student perseverance, and can improve problem completion time. However, both strategies have particular strengths and weaknesses. Since our feedback is completely automatic, it can be deployed at scale and integrated into existing massive open online courses.

Categories and Subject Descriptors: H.5.m. [**Information Interfaces and Presentation (e.g., HCI)**]: Miscellaneous

General Terms: Human Factors, Experimentation

Additional Key Words and Phrases: Autograding, feedback, automata, A/B study

ACM Reference Format:
Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? ACM Trans. Comput.-Hum. Interact. 22, 2, Article 9 (March 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2723163

1. INTRODUCTION

Both online and offline, student enrollment in computer science courses is rapidly increasing. For example, enrollment in introductory computer science courses has roughly tripled at Berkeley, Stanford, and the University of Washington in the past decade [Patterson 2013]; in addition, computer science is the most frequently taken massive open online course (MOOC) subject online [New York Times 2012; Jordan 2014]. With a thousand students in a lecture hall, or tens of thousands following a MOOC, standard

This research was supported by the NSF Expeditions in Computing award CCF 1138996.
Authors' addresses: L. D'Antoni, 3330 Walnut Street, Levine 565, Philadelphia, PA 19104-6389; email: lorisdan@seas.upenn.edu; D. Kini, 3301 Siebel Center, 201 N Goodwin Ave, Urbana, IL 61801; email: kini2@illinois.edu; R. Alur, 3330 Walnut Street, Levine 609, Philadelphia, PA 19104-6389; email: alur@cis.upenn.edul; S. Gulwani, Microsoft Corporation, One Microsoft Way, Redmond, WA, 98052; email: sumitg@microsoft.com; M. Viswanathan, 201 N Goodwin Ave, Urbana, IL 61801; email: vmahesh@illinois.edu; B. Hartmann, 533 Soda Hall, Berkeley, CA 94720-1776; email: bjoern@eecs.berkeley.edu.

DOI: http://dx.doi.org/10.1145/2723163

approaches to providing individualized feedback do not scale. However, students need appropriate guidance through specific feedback to progress and overcome conceptual difficulties.

Our focus in this article is on the problem of deterministic finite automata (DFA) construction. DFAs are a fundamental topic in computer science education. Besides being part of the standardized computer science curriculum, the concept of DFA is rich in structure and potential applications. It is useful in diverse settings such as control theory, text editors, lexical analyzers, and models of software interfaces. We focus on providing feedback for assignments in which a student is asked to provide a DFA construction corresponding to a regular language description.

One way to provide feedback is to use techniques of statistical machine learning. Using training data, incorrect solutions are categorized into known classes of mistakes. This method is general but requires annotations. For automata, and similar domains, there is an opportunity to use fully automated and potentially more accurate methods based on formal analysis of the solution. This is the approach that we focus on in this work. Thanks to the closure and decidability properties of DFAs, it is simple to check whether a student solution is a correct one and output a binary (correct/incorrect) feedback. Going a step further, if the solution is incorrect, one can produce a short counterexample on which the student solution does not perform correctly (e.g., *"Your DFA accepts the string ab but the correct DFA doesn't"*). Such procedures are used for producing feedback in current state-of-the-art tools [Rodger and Finley 2006]. Although counterexamples are very helpful, they do not suggest how to fix the student's solution, and at times, this can lead the student to fix the solution in a wrong way. In particular, the student might treat the counterexample as a special case on which the solution does not work rather than try to generalize it to the actual mistake. In many cases, the student may benefit from a hint on how to fix the solution rather than a specific counterexample (e.g., *"Check the transitions out of state 3"*).

The first contribution of this article is a new technique for automatically generating high-level (conceptual) hints for student DFA constructions. Our new algorithms for generating feedback build on recent techniques developed for automatic grading [Alur et al. 2013]. These grading algorithms can identify the type of student mistake; we translate them into readable feedback. In some cases, this is as easy as generating a counterexample, although in other cases it requires more complex paraphrasing techniques to go from succinct logical representations to English. For example, using the DFA edit difference proposed in Alur et al. [2013], we can generate feedback of the form *You need to change the acceptance condition of one state*, for a solution in which one final state is missing, or a sentence like *Your solution accepts the language* {*s* |*'ba' appears in s at least twice*} for a solution that accepts a different language from the one in the assignment. These new feedback techniques can be easily adapted to many other problems in theory of computation, including non-DFA and regular expressions.

At a high level, our approach proposes novel techniques for detecting different types of mistakes and translating them into explanatory feedback. We distinguish *semantic* mistakes, which likely are due to a misunderstanding of the problem, from *syntactic* mistakes, which likely are due to incorrectly expressing a semantically correct solution. Accordingly, our system generates different types of hints that either describe conceptual mistakes of the given solution or offer pragmatic feedback on what kind of syntactic edits have to be performed. Similar types of feedback can be applied in many other domains, including programming problems [Singh et al. 2013], and geometric constructions [Gulwani et al. 2011; Itzhaky et al. 2013].

The second contribution of the article is a comprehensive study of the effectiveness of feedback in the process of learning DFA constructions. We compare two state-of-the-art techniques, *binary* (correct/incorrect) and *counterexample*-based feedback, as well

as our new *hint*-based feedback system. These techniques could be combined into a comprehensive policy. We separate them in our study to learn about the utility of each in isolation. Our techniques are embodied in an online tool,[1] where students can submit automata and receive tailored feedback. To understand if feedback techniques help students become proficient in automata construction, we carried out a field experiment with 377 participants from two introductory theory of computation courses. To the best of our knowledge, this is the first study on effectiveness of any kind of feedback in DFA constructions.

Based on our results, we conclude that both counterexample-based feedback and our new hint-based feedback are more effective than binary feedback in students' learning process. Students receiving feedback need less time to solve problems and persevere longer on optional practice problems. We thus believe that feedback is effective in learning DFA concepts. At a high level, we did not observe any significant difference between counterexample-based feedback and our new hinting technique. However, when looking at individual problems, we can observe how both techniques have distinct benefits. Whereas counterexamples are helpful in understanding simple syntactic mistakes, our hints are helpful in identifying conceptual mistakes for which a single counterexample is not sufficient.

Finally, we deployed our tool and made it available online. The tool was used by a third university not involved in our study. Survey responses show that students and the teacher were enthusiastic, and the teacher expressed his interest in using it in future offerings of the course. Other institutes have adopted the tool in their courses. Since our feedback is completely automatic, we believe that our tool is ready to be deployed at scale and integrated into existing MOOCs so that it can reach tens of thousands of students.

Beyond the domain of DFAs, this article provides a classification of different types of feedback along with examples of how this approach may generalize. Our evaluation design, which combines required and optional problems to investigate both student performance and perseverance, can also serve as a template for conducting online experiments on feedback techniques. We hope that these contributions provide helpful guidelines for evaluating the effectiveness of automated personalized feedback, which is increasingly used in a wide variety of large online courses.

2. RELATED WORK

2.1. Intelligent Tutoring Systems

Intelligent tutoring systems (ITSs) aim to emulate the efficacy of personal, one-on-one tutoring, a gold standard of teaching techniques. ITSs often adopt two major functions: first, they guide students through a curriculum by selecting or suggesting which problems a student should work on; second, they provide feedback and assessment on individual problems [VanLehn 2006]. This article focuses on the task of providing personalized feedback and evaluating how different feedback strategies (binary, counterexample, and hint) compare to each other in terms of effectiveness. In the following, we describe how our contributions fit with respect to each component of an ITS.

2.1.1. Task Selection. Task selection uses student models that attempt to capture and describe student knowledge [Koedinger et al. 2006; Mitrovic et al. 2001]. Some ITSs support automatic problem generation to enrich the pool of tasks from which to select. The techniques vary a lot in this setting and are often problem specific [Ahmed et al. 2013; Singh et al. 2012; Andersen et al. 2013].

[1]Tool available at http://automatatutor.com.

Our system does not employ any automatic task selection, and in this work we do not generate problems automatically—students decide which problem they work on. Our tool could in principle generate new problems and solutions, but we decide to concentrate on a set of manually selected problems to better evaluate the feedback techniques without spreading the data over many different problems.

2.1.2. Feedback and Assessment. In a typical ITS, feedback is provided at each step or even substep produced by the student while attempting a solution. In contrast, in computer aided instruction (CAI), feedback is only shown after a complete solution has been submitted, by matching against a set of manually predefined answers and associated correction strategies [Conati 2009].

Providing useful hints and feedback requires models specific to each problem domain. ITSs typically categorize errors beforehand and associate feedback with them—this might be possible to do for some well-studied procedural problems such as addition or subtraction [Ashlock 1986; VanLehn 1992]. CAIs, on the other hand, categorize the set of possible solutions rather than the set of possible mistakes and then provide feedback on this finite set of solutions.

In the case of automata problems (which belong to the conceptual domain, where there is no set procedure to follow), students can submit infinitely many possible solutions, and the categorization of solutions needs to be done at the error level. Although this can be easily done for binary and counterexample feedback, the task is harder when dealing with more conceptual feedback. In this article, we introduce a new automatic feedback technique for the domain of automata construction that is able to propose personalized hints based on the student mistake. Our system has features of both ITSs and CAIs. As in CAIs, feedback is generated at submission time, and it is solely based on the current submission, not on a student's history. On the other hand, the instructor only has to provide one representative solution. Our system is capable of automatically comparing the student's attempt against a larger class of solutions and providing personalized feedback. In this aspect, our system is closer to an ITS.

Policies determine how to progress through different types of feedback—importantly, the overall goal is to get a student to solve problems without aid later on. Giving solutions away does not contribute to this goal. Our tool tries to produce feedback that is helpful but does not give away the solution. In our work, we evaluate different types of feedback independently—binary (correct/incorrect), pointing out a specific problem (counterexample), and hinting at solutions. These techniques could be combined into a comprehensive policy. We separate them in our study to learn about the utility of each in isolation. Similar types of feedback can be applied in many other domains, including programming problems [Singh et al. 2013] and geometric constructions [Gulwani et al. 2011; Itzhaky et al. 2013].

Some prior work has studied the effects of different feedback strategies. For example, in the context of geometry and algebra problems [Anderson et al. 1995; VanLehn 2011], when feedback is delayed to a student's submission, different feedback types do not affect learning in terms of assessment outcomes, but they affect solution speed and engagement in the course. We observe similar results in our study in the DFA domain.

Minimal feedback has been shown to cause student frustration [Razzaq and Heffernan 2006; Gallien and Oomen-Early 2008]. Our study yields a similar result: binary feedback, which is clearly insufficient in indicating why the student's attempt is wrong, was judged to be most confusing.

2.1.3. ITS Evaluation. Several guidelines have been proposed for evaluating the effectiveness of ITSs [Shute and Regian 1993; Mark and Greer 1993], and the evaluation in this article follows them. Typically, the evaluation is divided into internal (regarding the robustness of the tool) and external parts (regarding the effectiveness of the tool

on learning). In terms of internal evaluation, we analyze which feedback the students perceive as less confusing and more helpful. For external evaluation, we measure the average grade of students receiving each type of feedback.

2.1.4. Applications. Many tutoring systems have focused on various aspects of computer science, such as LISP programming [Anderson and Reiser 1985] and SQL queries [Mitrovic 1998]. Recent work on automatic feedback generation for programming assignments in Python [Singh et al. 2013] shares similarities to our own approach in terms of goals and the use of synthesis techniques for implementation. We next review prior work specific to automata education.

2.2. Automata Education

There are several strategies for teaching automata and other formalisms in computer science education. Our system is the first one to provide students with personalized feedback that differs from binary and counterexample feedback.

Alur et al. [2013] provide a method of autograding automata, assigning partial credit based on an analysis of the type of problem the students' DFA exhibits. Grades by this system are comparable to those of expert human graders. However, their system does not provide any rationale for the assigned grade to the student, limiting its utility as a learning tool. We build directly on their work to generate feedback.

The main other existing tools for teaching DFA constructions are JFLAP, FLUTE, and Gradiance. JFLAP [Rodger and Finley 2006] allows students to author and simulate automata—it is widely used in classrooms. Instructors can test student models against a set of input strings and expected results. Recently, JFLAP was equipped with an interface that allows students to test their solution DFA on problems for which they only have an English description of the language [Shekhar et al. 2014]. To do this, the student writes an imperative program that matches the given language description, and if this program is not equivalent to the student's DFA, JFLAP tries to automatically produce a counterexample. FLUTE is an ITS for formal languages [Devedzic and Debenham 1998]. It focuses on guiding students through a set of concepts through an appropriate plan. Gradiance[2] is a learning environment for database, programming, and automata concepts. It focuses on providing tests based on multiple-choice questions. These tools either do not support a way for drawing DFAs or do not have a high-level representation of a problem and can therefore not provide feedback about the conceptual problems with a student's submission.

Other tools are available for problems related to DFA constructions. In ProofChecker [Stallmann et al. 2007], students prove the correctness of a DFA by labelling the language described by each state: given a DFA, the student enters "state conditions" (functions or regular expressions) describing the language of each individual state. The system generates all strings up to length $n + 2$ for an n-state DFA and checks whether the condition of the final state for each string is satisfied. In DeduceIt [Fast et al. 2013], students are asked to solve assignments on logical derivations. DeduceIt is then able to grade such assignments and provide incremental feedback. Visualizations such as animations of algorithms or depictions of transformations between automata and equivalent regular expressions exist [Braune et al. 2001]. These systems focus on different problems that do not support the problem of constructing DFAs corresponding to a given language.

In general, the number of controlled experiments of automata teaching tools is very limited, and to the best of our knowledge, no prior work tackles the effectiveness of feedback in learning automata constructions.

[2]http://www.newgradiance.com/.

3. TYPES OF FEEDBACK IN AUTOMATA CONSTRUCTIONS

Three general strategies exist for presenting feedback [VanLehn 2006]:

(1) Binary feedback that indicates whether a student's solution is correct or not
(2) Error-specific feedback that points out what is wrong
(3) Solution-oriented feedback that suggests strategies for addressing an error.

In this section, we discuss different approaches to operationalize these feedback strategies for the domain of automata construction. Our study later compares the impact of showing feedback associated with these different approaches.

3.1. Binary Feedback

This is a yes/no message that tells the student whether the DFA does or does not meet the input specification (type 1).

3.2. Counterexample Feedback

A counterexample is a specific string on which the student DFA does not meet the input specification. Counterexamples are one instance of pointing out what is wrong (type 2). For example, the first-attempt DFA in Figure 1 accepts the string *ababab,* whereas the correct solution does not. Most DFA learning tools in this case would provide a feedback of the following form: *Your DFA accepts the string 'ababab' but the correct DFA does not.* Counterexamples enable students to trace the behavior of their DFA symbol by symbol on this string and can thus help to identify the origin of a mistake.

3.3. Descriptive Hint Feedback

A hint is a descriptive message that tries to help the student's conceptual understanding of a problem. Hints can either describe *how* a student's provided solution is incorrect (type 2) or may describe strategies that can be applied to correct mistakes (type 3). Based on a review of common teacher feedback in homework assignments from previous automata courses, we developed three new types of hint feedback. Although counterexamples can be considered a particular type of hint, we observed that teachers rarely offer counterexamples during grading. We hypothesize that counterexamples alone are not sufficient for explaining the rationale behind assigning partial credit—they do not provide enough information about the type of a mistake or the root of the student's misconception. Our techniques for generating feedback build on the work of grading automata constructions presented in Alur et al. [2013]. The three hint types are presented next:

—*Problem Syntactic Mistake: Solution accepts a different, related language.* In this type of mistake, the student DFA accepts a language that has a description that is syntactically close but not equal to the correct one. The corresponding feedback is an English description of the language computed by student DFA. An example is the feedback for the first-attempt DFA in Figure 1:

  Your solution accepts the following set of strings: {s | 'ba' appears in s **at least** $twice$.}

  Similar to counterexamples, these hints point out what is wrong (type 2).

—*Problem Semantic Mistake: Solution fails to accept some strings.* In this type of mistake, the student DFA is correct on many inputs but not all the them. The corresponding feedback is an English description of a subset of misclassified strings.

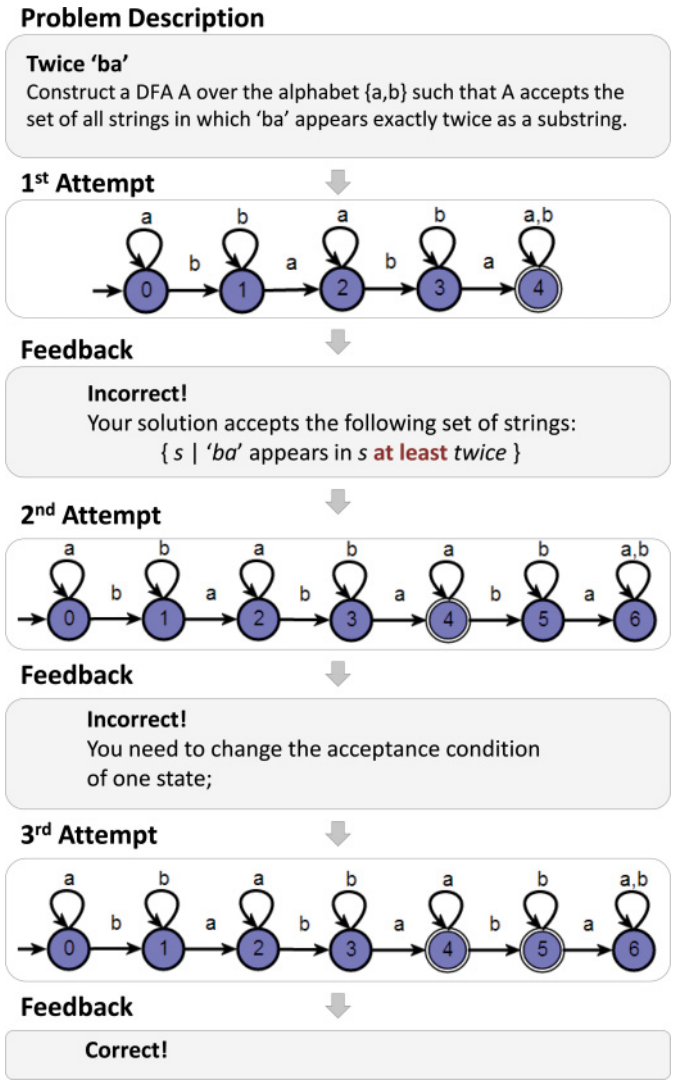


Fig. 1. When solving DFA constructions, a student provides an attempt solution for the target language. The student receives personalized feedback and keeps refining the solution until finding a correct one.

For example, a possible feedback for the first-attempt DFA in Figure 1 would be:

> Your DFA is incorrect on the following set of strings: {s | 'ba' appears in s more than $twice$.}

This feedback is a generalization of a particular counterexample—by presenting a description of a set of incorrect strings, we hypothesize that students can think more holistically about the behavior of their DFA. These hints again point out what is wrong (type 2).

—*Solution Syntactic Mistake: DFA has some structural errors.* In this type of mistake, the student DFA is syntactically close but not equal to a correct DFA. This means that a small number of structural changes (e.g., changing a transition or adding or

removing a state) can transform the DFA into a correct solution. The corresponding feedback is an English description of what states and transitions of the DFA should be changed to "fix" it. An example is the feedback of the second-attempt DFA in Figure 1:

You need to change the acceptance condition of one state.

This type of hint suggests a strategy for addressing an error (type 3).

3.4. An Example Student Session

We will now describe a typical session of a student solving a DFA construction problem in our setting: the student, given an English description of a regular language, draws a solution attempt and receives the corresponding feedback. After receiving the feedback, the student can modify the solution and submit again. This iterative cycle continues until the student submits a correct DFA. An example of this interaction is shown in Figure 1. For this particular example, we illustrate the new type of feedback introduced in this article. However, the interaction is the same for different types of feedback provided by the tool.

The problem description. The tool shows an English description of the language for which the student has to construct the corresponding DFA. Here the student is asked to draw a DFA that accepts all strings containing the substring *'ba'* exactly twice.

A first solution. The student draws a first-attempt DFA that she believes accepts the target language. The student can draw and erase parts of the DFA and hit submit when satisfied with the attempt.

Personalized feedback. The submitted solution is compared against the problem specification, producing a personalized feedback. In this case, the feedback alerts the student that her solution accepts a different but related language of strings containing the substring *'ba' at least* twice (instead of *exactly* twice—a Problem Syntactic Mistake). As we will describe later, this feedback is produced using formal methods algorithms. The feedback is shown to the student, suggesting that the problem has been misunderstood.

A revised solution. The student can now revise the solution according to the feedback and add two extra states to deal with the case where the input string contains more than two occurrences of *'ba'*. The new attempt is submitted.

Personalized feedback. The second student attempt is very close to a correct solution but misses a final state (a Solution Syntactic Mistake). The corresponding feedback message is shown to the student.

Correct solution. Finally, the student draws a correct DFA and after submitting receives a confirmation that the problem has been solved.

3.5. Generalizing Counterexamples and Hints to Other Domains

Although it is clear that a binary feedback is applicable to many domains besides automata constructions, in this section we show how counterexamples and descriptive hints generalize to other settings. Table I shows some examples.

The feedback for a problem syntactic mistake, which in our case is an English description of the language accepted by the student DFA, points out what is wrong by characterizing what problem the student solved instead of the specified one.

The feedback for a problem semantic mistake, for which our hint is an English description of misclassified strings, points out what is wrong by describing the inputs on which the solution is incorrect. Counterexamples fall under this category and are in general easy to produce; however, we have a hypothesis that going beyond a single counterexample to sets of misclassified inputs is useful. For example, when looking at Table I, in the case of geometric constructions, telling the student that the solution is

Table I. Types of Feedback in Domains beyond Automata Constructions

Problem Syntactic Mistake	
Regular expressions	Your solution accepts ab^* instead of $(ab)^*$.
Geometric constructions	You might have missed that the angles have to be identical.
Algebra problems	You might have missed the minus in front of the x.
Programming	The description required the list to be in descending order.
Problem Semantic Mistake	
Regular expressions	Your solution is incorrect on all strings of even length.
Geometric construction	Your solution is incorrect when AB and CD are parallel.
Algebra problems	Your solution is incorrect when $x > 0$.
Programming	Your solution is incorrect on lists containing a negative numbers.
Solution Syntactic Mistake	
Regular expressions	If you replace a $*$ with a $+$, your solution will be correct.
Geometric construction	The angle ABC is not equal to the angle BCD.
Algebra problems	You didn't flip the sign of y when changing the side of the $<$.
Programming	The inequality in the for-loop should be strict.
Counterexample	
Regular expressions	Your solution is incorrect on the empty string.
Geometric construction	Your solution is incorrect on the rectangle shown in the figure.
Algebra problems	Your solution is incorrect on the value $x = 0$.
Programming	Your solution is incorrect on the empty list.

incorrect when two segments are parallel is more informative than showing a particular rectangle as a counterexample.

The feedback for a solution syntactic mistake, which in our case is a hint on how to fix the DFA, explains what steps are necessary to remedy the error. However, the steps are not a recipe—they tell the student where to look rather than what to do exactly.

Although problem syntactic and problem semantic mistakes might have similar characteristics, they refer to two different types issues with the student solution. The former captures the case in which the student misunderstood the problem and solved a variation of it, whereas the latter captures the case in which the student misclassified a few "corner cases." If we look again at Table I, in the case of algebra problems, if the student missed a sign in the problem description, it is more helpful to point the student to the problem description rather than provide incorrect inputs.

In summary, counterexamples and feedback for problem syntactic and semantic mistakes are similar in that they focus on the notion of pointing out what is wrong. The feedback for solution syntactic mistakes is quite different because it focuses on how to change the solution.

We believe that by using our mistake classification, one can create a general guideline for designing feedback systems. In particular, for each type of mistake, one needs to provide the following two components:

(1) A technique that given a solution identifies whether it contains this particular type of mistake
(2) A technique that given a representation of the mistake produces a hint in English.

For example, Singh et al. [2013] show how to develop these two components for solution syntactic mistakes (but not for problem syntactic/semantic mistakes) in the context of introductory programming assignments. Their technique first finds small edits that can transform the student solution into a correct program and then prompts the student with a hint on how to find such a set of edits.

4. IMPLEMENTING FEEDBACK FOR DFA CONSTRUCTIONS

4.1. Background on DFAs

A DFA over an alphabet Σ is a tuple $A = (Q, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \mapsto Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. Given $q, q' \in Q$ and $a \in \Sigma$, if $\delta(q, a) = q'$, we say that A has a transition $t = (q, a, q')$ and call q the source state of t, a the label of t, and q' the target state of t. We define the transitive closure of δ as, for all $a \in \Sigma$, $s \in \Sigma^*$, $\delta^*(q, as) = \delta^*(q', s)$, if $\delta(q, a) = q'$, and $\delta^*(q, \varepsilon) = q$. The language accepted by A is $L(A) = \{s \mid \delta^*(q_0, s) \in F\}$. For example, the first DFA in Figure 1 accepts the language of strings over the alphabet $\{a, b\}$ in which the string ab appears at least twice as a substring.

4.2. Binary Feedback and Counterexamples

Using DFA closure properties, one can check whether the student solution is equivalent to the target DFA and produce the corresponding binary feedback. If the two DFAs are not equivalent, one can compute a DFA accepting their symmetric difference and produce a counterexample by finding a string accepted by such a DFA. Binary feedback and counterexamples are the two state-of-the-art techniques for providing feedback for DFA constructions and are found in many existing tools [Rodger and Finley 2006].

4.3. Descriptive Hints

In this section, we briefly recall the techniques introduced in Alur et al. [2013] for grading DFA constructions and show how they can be used to generate the three previously described types of hint feedback. Given a target language L_T and a student DFA A_s, grading the student solution corresponds to finding a metric that tells us how far A_s is from a L_T. In Alur et al. [2013], the authors identify three classes of mistakes and investigate three approaches that try to address each class. The following list summarizes the three techniques used to compute the grade and the corresponding feedback from the previous section:

—*Problem Syntactic Mistake.* The student gives a solution for a different language for which the description is syntactically close to the description of L_T.
Technique: Synthesize a logic description of A_s and L_T and use tree edit distance [Gao et al. 2010] to compute the difference between two formulas.
Feedback: Use logic description of A_s to produce an English description of A_s.

—*Problem Semantic Mistake.* The solution is wrong on a small fraction of strings.
Technique: Use regular language density to compute the size of the difference L_D between L_T and $L(A_s)$ when viewed as sets.
Feedback: English description of the set L_D of misclassified strings, or English description of a subset $L'_D \subseteq L_D$ of misclassified strings, or counterexample.

—*Solution Syntactic Mistake.* The student DFA is syntactically close to a correct one.
Technique: Use notion of DFA edit distance to capture the smallest number of edits (edit script) necessary to transform A_s into a correct DFA
Feedback: In addition, use the edit script tell the student how to fix A_s.

Next, we formally describe the metrics used for each type of mistake and how each feedback is computed.

Background on **MOSEL** In the following, we refer to the logic MOSEL introduced in Alur et al. [2013] without formally defining it. Informally, MOSEL predicates can describe all (and only) regular languages. The logic contains many high-level constructs that make MOSEL predicates succinct, human readable, and very close to natural English descriptions of the regular language. For example, the language *Twice 'ba'* of Figure 1

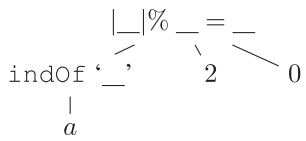


Fig. 2. Parse tree for $\phi = |\texttt{indOf } \text{`}a\text{'}|\% \ 2 = 0$.

is captured by the MOSEL predicate $|\texttt{indOf } \text{`}ba\text{'}| = 2$. This predicate can be read as the size ($|\_|$) of the set of positions containing the string '*ba*' (`indOf` '*ba*') is 2. We illustrate the main features of MOSEL through some examples of languages:

- `begWt` '*a*' $\wedge$ |`indOf` '*ab*'|% 2 = 1. Strings that start with an *a* and have and odd number of occurrences of the substring *ab*.
- |`indOf` '*a*'| $\geq 2 \vee$ |`indOf` '*b*'| ≥ 2. Strings that contain at least two *a*'s or at least two *b*'s.
- *a*@{*x*| |`posLe` *x*|% 2 = 1}. Strings where every odd position ({*x*| |`posLe` *x*|% 2 = 1}) is labeled with an *a*.
- `begWt` '*ab*' $\wedge$ |`all`|% 3 $\neq$ 0. Strings that start with *ab* and with length not divisible by 3.
- |`indOf` '*ab*'| = 2. Strings that contain the substring *ab* exactly twice.
- |`indOf` '*aa*'| $\geq 1 \wedge$ `endWt` '*ab*'. Strings that contain the substring *aa* at least once and end with *ab*.

In Alur et al. [2013], it is shown that given a MOSEL predicate φ, it is possible to compute a DFA A_φ accepting the language described by φ, and given a DFA A, it is possible to compute a MOSEL formula φ_A describing the language accepted by A. However, whereas the first operation is efficient in practice, the second one is shown to be slower.

4.3.1. Problem Syntactic Mistake. The following metric captures the case in which the MOSEL description of the language $L(A_s)$ corresponding to the student DFA A_s is close to the MOSEL description of the target language L_T. This metric computes how syntactically close two MOSEL descriptions are. We consider MOSEL formulas as the ordered trees induced by their parse trees. Figure 2 presents an example of a parse tree. Given a MOSEL formula ϕ, let T_ϕ be its parse tree. Given two ordered trees t_1 and t_2, their tree edit distance $\text{TED}(t_1, t_2)$ is defined as the minimum number of *edits* that can transform t_1 into t_2. Given a tree t, an edit is one of the following operations:

—*Relabel.* Change the value of a node n.
—*Node deletion.* Given a node n with parent n', (1) remove n and (2) place the children of n as children of n', inserting them in the "place" left by n.
—*Node insertion.* Given a node n, (1) replace a consecutive subsequence C of children of n with a new node n' and (2) let C be the children of n'.

The distance TED can be computed using the algorithm in Gao et al. [2010]. The distance $\text{D}(\phi_1, \phi_2)$ between two formulas ϕ_1 and ϕ_2 is then defined as $\text{TED}(\text{T}_{\phi_1}, \text{T}_{\phi_2})$. Finally, such distance is weighted to the size of the target formula $\text{WTED}(\phi_1, \phi_2) \stackrel{\text{def}}{=} \frac{\text{D}(\phi_1, \phi_2)}{|\text{T}_{\phi_2}|}$, where $|\text{T}|$ is the number of nodes of a tree T. In this way, for the same number of edits, fewer points are deducted for languages with a bigger description.

The feedback corresponding to this kind of mistake is the English description of the student DFA A_s. Such a description is computed in two steps. First, we use the synthesis algorithm in Alur et al. [2013] to find a MOSEL formula ϕ_{A_s} corresponding to A_s. Next, we inductively produce an English description of the formula ϕ_{A_s}. To avoid a "robotic" English, we use deep pattern matching. Classical pattern matching consists

of reading one node of the tree at a time in a top-down manner. On the other hand, deep pattern matching consists of reading multiple adjacent nodes in a single step, always in a top-down manner. This technique is often used in translations for natural language processing [Maletti 2008].

For example, consider the formula $\phi \stackrel{\text{def}}{=} |\texttt{indOf}\ \text{`}ab\text{'}|\%\ 2 = 1$. The following are possible rules for classical pattern matching of $\phi$: (1) $t(|\phi'|\%\ n = m) =$ *when dividing by n the size of* $t(\phi')$ *we obtain remainder m* and (2) $t(\texttt{indOf}\ \text{`}\alpha\text{'}) =$ *the set of* 'α' *in 's.'* A mechanical translation of ϕ using the preceding rules would yield a description such as *when dividing by 2 the size of the set of occurrences of 'ab' in 's,' we obtain remainder 1*. Deep pattern matching, on the other hand, would use the rule

$$t(|\texttt{indOf}\ \text{`}\alpha\text{'}|\%\ 2 = 1) = \text{`s' contains an odd number of `}\alpha\text{'}$$

and produce the description *'s' contains an odd number of 'ab.'*

Considering that for each language there exist infinitely many MOSEL formulas describing it, we set a time-out in the synthesis procedure and only consider the formulas discovered in such a time span. We then output the formula describing $L(A_s)$ that has smallest tree edit distance from the smallest formula describing L_T. Using the information in the tree edit script, we can also highlight the difference between the descriptions of $L(A_s)$ and L_T. An example of such a technique is shown in the first feedback of Figure 1 where the words *at least* are highlighted.

4.3.2. Problem Semantic Mistake. The following metric captures the case in which the DFA A_s behaves correctly on most inputs, by computing what percentage of the input strings is correctly accepted/rejected by A_s. Given two languages L_1 and L_2, their *density difference* is defined as

$$\text{DEN-DIF}(L_1, L_2) \stackrel{\text{def}}{=} \lim_{n \to +\infty} \frac{|((L_1 \setminus L_2) \cup (L_2 \setminus L_1)) \cap \Sigma^n|}{max(|L_2 \cap \Sigma^n|, 1)},$$

where Σ^n denotes the set of strings in Σ^* of length n. Informally, for every n, the expression $E(n)$ inside the limit computes the number of strings of length n that are misclassified by L_1 divided by the number of strings of length n in L_2. The *max* in the denominator is used to avoid divisions by 0. Unfortunately, as shown in Alur et al. [2013], the density difference is not always defined, as the limit may not exist; therefore, in practice, this quantity is approximated to a finite value of n.

Similar notions of density have been proposed in the literature [Kozik 2005]. The density $\text{DEN}(L)$ of a regular language L over the alphabet Σ is defined as the limit

$$\text{DEN}(L) \stackrel{\text{def}}{=} \lim_{n \to +\infty} \frac{|L \cap \Sigma^n|}{|\Sigma^n|}.$$

When this limit is defined, it is also computable [Bodirsky et al. 2004]. The conditional language density $\text{DEN}(L_1|L_2)$ of a given language L_1 in a given language L_2, such that $L_1 \subseteq L_2$, is the limit

$$\text{DEN}(L_1|L_2) \stackrel{\text{def}}{=} \lim_{n \to +\infty} \frac{|L_1 \cap \Sigma^n|}{|L_2 \cap \Sigma^n|}.$$

Again, there are languages for which these densities are not defined, but when they are, they can also be computed [Kozik 2005]. These definitions have good theoretical foundations, but unlike the metric presented in Alur et al. [2013], they are undefined for most DFAs.

The feedback corresponding to this type of mistake aims at describing the set $L_D = (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$ of misclassified strings. The feedback is produced using the same

technique that we presented for the problem syntactic mistake. The algorithm for computing the feedback proceeds as follows:

(1) Try to synthesize a MOSEL description ϕ of L_D; if a description is found within 2 seconds, output the English description of ϕ; otherwise,
(2) try to synthesize a MOSEL description ϕ' of some subset of $L_{D'} \subseteq L_D$; if a description is found within 2 seconds, output the English description of ϕ'; otherwise,
(3) output a counterexample $\alpha \in D$.

This algorithm outputs the English description of the language difference L_D. However, when failing, the algorithm tries to first compute an underapproximation L'_D of L_D, and if it still fails, it simply outputs a counterexample.

The underapproximation ϕ' is computed using a similar synthesis technique as the one in Alur et al. [2013]. The synthesis algorithm in Alur et al. [2013] simply enumerates all possible MOSEL predicates and checks for equivalence with the target language. The enumeration is sped up using static techniques and approximate equivalence. When looking for an underapproximation, instead of checking for equivalence, the synthesis procedure will check for language containment. To speed up this procedure, before checking for containment, each formula is tested on a set of negative examples (not accepted by D) that are generated using the algorithm for approximate equivalence shown in Section 2.4 of Alur et al. [2013].

Of all underapproximations generated in a given time-out, we output the English description of the smallest one. Finally, if no formula is found, we simply output the shortest counterexample.

4.3.3. Solution Syntactic Mistake. The following metric captures the case in which the student DFA A_s is syntactically close to a correct one, by computing how many edits are needed to transform A_s to make it accept the correct language L_T. The following notion of DFA edit difference is defined in Alur et al. [2013]. Given two DFAs A_1, A_2, the edit difference $\text{DFA-D}(A_1, A_2)$ is the minimum number of *edits* that can transform A_1 into some DFA A'_1 such that $L(A'_1) = L(A_2)$. Given a DFA A, an edit is one of the following operations:

—*Transition redirection.* Given a state q and a symbol $a \in \Sigma$, update $\delta(q, a) = q'$ to $\delta(q, a) = q''$ where $q' \neq q''$.
—*State insertion.* Insert a new disconnected state q, with $\delta(q, a) = q$ for every $a \in \Sigma$.
—*State relabeling.* Given a state q, add it or remove it from the set of final states.

Notice that since the final goal is to find a DFA that is language equivalent instead of syntactic equivalent to A_2, the operation of node deletion is not necessary for two automata to always admit a finite edit difference. For example, a DFA A_1 may be language equivalent to a DFA A_2, but it may contain an extra state that is unreachable. To take into consideration the severity of a mistake based on the difficulty of the problem, one can use the weighted metric

$$\text{WDFA-D}(A_1, A_2) \stackrel{\text{def}}{=} \frac{\text{DFA-D}(A_1, A_2)}{k + t},$$

where k and t respectively are the number of states and transitions of A_2. A similar distance notion is *graph edit distance* [Bille 2005]. However, this metric does not take into account the language accepted by the DFA.

In this case, the corresponding feedback is a hint that uses the edit script to help the student to fix the DFA. An edit script produces the following outputs:

n state insertions. *"You need to add n states."*
1 *state relabeling*. *"You need to change the acceptance condition of one state."*

$n > 1$ *state relabelings*. *"You need to change the set of final states."*
transition redirection. For each state q such that there exists a transition redirection from q, output *"Check the transitions out of state q."*

It might happen that there exist more than one minimal edit script for a particular DFA. When this is the case, our enumeration technique simply prompts the student with the description of the first edit script encountered in the search.

4.3.4. Selecting the Feedback. The aforementioned approaches need to be combined to compute the final feedback. The first decision to take into account is *How much feedback do we want to provide?* Since we do not want the student to be overwhelmed by the amount of feedback, we decide to only output one feedback statement. The next question is *Which statement do we pick?* Again, we decide to opt for Occam's razor and select the statement corresponding to the metric that captures the highest similarity between the student DFAs and the solution. The intuition behind this choice is that such a metric identifies the simplest explanation of the student's mistake. In few instances, this approach will cause the tool to output a feedback that is not necessarily the most natural one.

5. EVALUATION

Our evaluation of feedback has two major goals: (1) to determine if feedback can improve how students learn DFA constructions and (2) to identify particular strengths and shortcomings of different feedback types. We compare the following three types of feedback:

—*Binary feedback.* Shows if the solution is correct or not.
—*Counterexample.* Shows if the solution is correct or not and in the latter case provides a counterexample.
—*Hints.* Shows if the solution is correct or not and in the latter case shows different kinds of descriptive hints based on the type of mistake.

Binary feedback acts as a baseline for our study; counterexamples are used by other tools (e.g., JFLAP) [Rodger and Finley 2006], but the effectiveness has not been evaluated; and hints are novel to our work.

5.1. Participants

We recruited 377 students attending two undergraduate introductory theory of computation courses at two large universities in the United States. The students ages ranged from 18 to 22 years; 309 were male, and 68 were female.

5.2. Method

All participants used the tool Web site as part of a 1 week long homework assignment. The homework was divided into two parts[3]:

—*Part A.* This was a set containing the following four problems that students could attempt as many times as they wish and in the order they prefer while receiving feedback on incorrect solutions:
*A*1. $\{s | $ the number of a's in s is not divisible by $3\}$
*A*2. $\{s | s$ contains exactly two occurrences of 'ba'$\}$
*A*3. $\{s | s$ contains at least 2 a's or at most 2 b's$\}$
*A*4. $\{s | s$ does not end with 'b' if it starts with 'a'$\}$

[3]The questions reflect prior questions assigned in past offerings of the course.

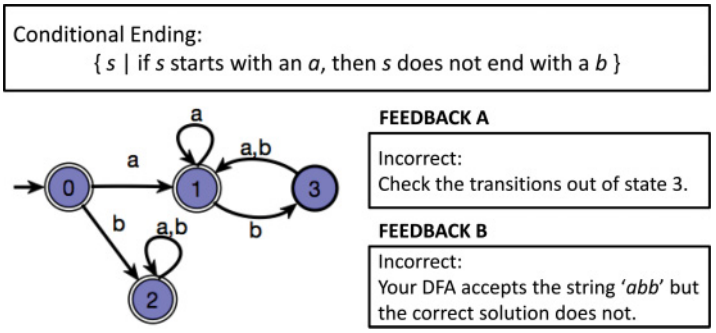


Fig. 3. Concrete example from survey on feedback preference.

—*Part B.* A single DFA construction for which each student has only one attempt. It contained the following problem:
B.{$s|s$ contains at least 2 a's and it ends with 'ab'}

Participants in this between-subjects experiment were randomly assigned to one of three feedback conditions, in which they either receive *Binary* feedback, *Counterexamples*, or *Hints* (which may include counterexamples). Feedback was shown for problems in part A, as well as for 18 practice problems that were available in the system but were not included in the homework assignment.

After submitting the problem set, participants also completed an online survey that collects data about the quality of the interface, the usability of the tool, and the effectiveness of the feedback received. Participants in the *Hint* condition also received an additional survey that presents two concrete examples of incorrect DFAs with counterexample and hint feedback (like the one shown in Figure 3). The survey elicited feedback preference as well as a rationale for their choice. We distributed these questions only to the students in the *Hint* condition since they saw both kinds of feedback during their homework.

5.3. Measures

Our dependent variables aim to capture student performance and learning over the course of the assignment. Our *problem-specific analysis* measures performance of the entire cohort for particular problems. We also perform a *submission-specific analysis* that groups students who submitted a common incorrect solution to a particular problem and traces their ultimate outcomes.

5.3.1. Problem-Specific Analysis. For each feedback group, we measure

—*number of attempts* before solving part A problems;
—*time taken* after the first submission until a problem in part A is solved;
—*improvement* in grade[4] for part A problems after the initial feedback is received;
—the *average final grade* on part B of the homework; and
—*the number of students giving up* before solving a homework or practice problem.

5.3.2. Submission-Specific Analysis. Let a *popular* solution be an incorrect DFA construction that has been drawn by more than 20 students. For each popular solution P, we call S_P^i the set of immediately subsequent student attempts (the solutions drawn right after receiving the feedback) from group i. For each popular solution P and for each group i, we measure

[4]The grades we consider are those generated by the technique proposed in Alur et al. [2013].

—*average grade* in S_P^i;
—*percentage of students giving up* in S_P^i (students leaving the homework page); and
—*percentage of correct solutions* in S_P^i.

5.3.3. Surveys. For the posttest survey, we use Likert scales to measure participant perception if the feedback is overall useful; if it is helpful in understanding mistakes; if it is helpful for getting to the correct DFA; and, conversely, if the feedback is confusing. In addition, we measure whether students in the hint feedback group prefer the counterexample feedback or the hint feedback.

5.4. Hypotheses

We have the following expected outcomes:

—*H1: Attempts.* In multiple-submission problems (part A), students in the *Binary* condition will need more attempts than students in *Counterexample* or *Hints*, and students in *Counterexample* will need more attempts than students in *Hints*.
—*H2: Time.* In multiple-submission problems (part A), students in the *Binary* condition will need more time than students in *Counterexample* or *Hints*.
—*H3: Grade Improvement.* In multiple-submission problems (part A), students in the *Binary* condition will improve less after receiving feedback than students in *Counterexample* or *Hints*.
—*H4: Single-Submission Grade.* In the single-submission problem (part B), students in the *Binary* condition will receive lower scores than students in *Counterexample* or *Hints*.
—*H5: Drop-Outs.* Students in the *Binary* condition will abandon more problems before correctly solving them than students in *Counterexample* or *Hints*.
—*H6: Preference.* Students will prefer hint feedback to counterexample feedback, and both to binary feedback.
—*H7 Specificity.* There will be cases in which the counterexample-based feedback will be considerably more effective than the hint feedback and vice versa.

5.5. Results

In aggregate, after removing the cases in which students solved a problem in a single attempt (and received no feedback), students submitted 3,085 solution attempts for the four problems in part A (an average of 3.53 per student per problem). Students performed well overall, receiving an average grade of 9.78/10 points for the homework. Surprisingly, many students also attempted additional, unscored practice problems: we received 4,209 such submissions by 293 students.

5.5.1. Attempts. Students took a mean of 3.3 attempts to correctly solve a problem in *Binary*, 3.7 in *Counterexample*, and 3.5 in *Hint* (Figure 4, top). Using a two-way ANOVA with problem and feedback type as independent variables, we find a significant main effect for problem ($F(3,862) = 14.57$, $p < 0.001$) but not for feedback type. We did not find a significant interaction effect. Therefore, *H1 is not supported*.

5.5.2. Time. Students were slower in the *Binary* condition (mean: 394 seconds) than in *Hint* (mean: 264 seconds) and *Counterexample* (mean: 245 seconds) (Figure 4, middle). Using a two-way ANOVA, we find significant main effects for feedback type ($F(2,820) = 4.17$, $p < 0.05$) and problem ($F(3,820) = 12.71$, $p < 0.001$). We did not find a significant interaction effect. Pairwise comparisons with Welch two sample t-tests show a significant difference between *Binary* and *Counterexample* ($t(401.9) = 2.26$, $p < 0.05$) and between *Binary* and *Hint* ($t(364.4) = 2.04$, $p < 0.05$), but not between *Counterexample* and *Hint*. *H2 is supported—Binary* is significantly slower.

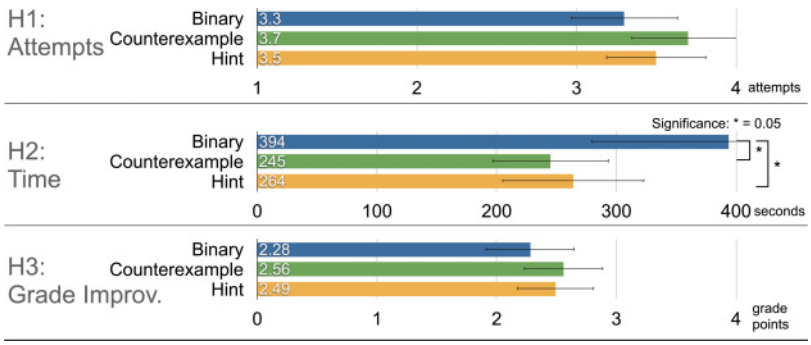


Fig. 4. Quantitative data for number of attempts, time taken, and grade improvement in our study. Error bars show 95% confidence interval.

5.5.3. Grade Improvement. Figure 4 (bottom) shows student grade improvement after a first incorrect submission. Students had a mean grade improvement of 2.28 points after a first incorrect submission in *Binary*, 2.56 points in *Counterexample*, and 2.49 points in *Hint*. Using a two-way ANOVA, we find a significant main effect for problem ($F(3,853) = 30.64$, $p < 0.001$) but not for feedback type. We did not find a significant interaction effect. *H3 is thus not supported*.

5.5.4. Single-Submission Grade. Almost all students received perfect scores on part B (means: 9.67, 9.71, and 9.67 out of 10). This question was likely too easy, or students achieved mastery in part A. *H4 is not supported*.

5.5.5. Drop-Outs. Although almost all students successfully completed all assigned homework problems, we observed interesting differences in their perseverance on practice problems. In the *Binary condition*, students gave up on 44% of attempted practice problems without solving them successfully, whereas they gave up on only 27% of problems in *Counterexample* and 33% in *Hint*. The difference between *Binary* and any of the other two feedback types is significant ($X^2(2, N = 2246) = 48.43$, $p < 0.001$). So students persevere more when provided with richer feedback, and *H5 is supported*.

5.5.6. Preference. Posttest surveys were submitted by 122 students (33 from the *Binary* condition, 46 from *Counterexample*, and 43 from *Feedback*). Respondents found binary feedback less useful than other feedback. The median usefulness of feedback on a five-point Likert scale was $Md = 3$ (neutral) for *Binary*, 5 (strongly agree) for *Counterexample*, and 4 for *Hint*. Although Likert data is best understood as ordinal, we also show means in Figure 5. We found a significant effect of condition for pairs *Binary* and *Counterexample* (Wilcoxon rank-sum $Z = -4.49$, $p < 0.001$) and *Binary* and *Hint* ($Z = -3.62$, $p < 0.001$), but not *Counterexample* and *Hint*.

Students found counterexamples most helpful for understanding mistakes ($Md = 4$), followed by hints ($Md = 4$) and then binary ($Md = 2$) feedback. We found significant effects of condition for pairs *Binary–Counterexample* ($Z = -6.15$, $p < 0.001$) and *Binary–Hint* ($Z = -5.43$, $p < 0.001$) and *Counterexample–Hint* ($Z = 2.05$, $p = 0.04$).

Students found binary feedback less helpful for understanding how to correct their DFA ($Md = 2$) compared to $Md = 4$ for *Counterexample* and $Md = 4$ for *Hint*. We found a significant effect of condition for pairs *Binary–Counterexample* ($Z = -5.08$, $p < 0.001$) and *Binary–Hint* ($Z = -4.85$, $p < 0.001$).

Finally, students found binary feedback most confusing ($Md = 3$), and hints ($Md = 2$) more confusing than counterexamples ($Md = 2$). We found significant effects of

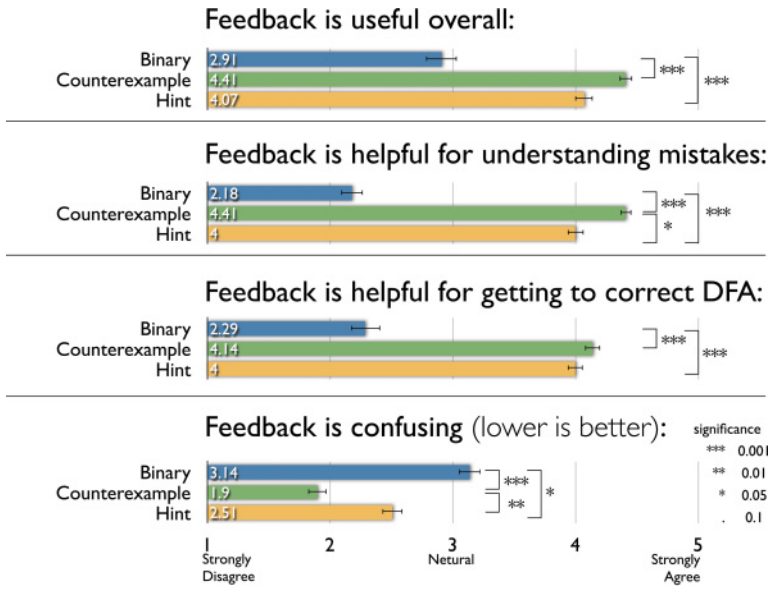


Fig. 5. Likert scale ratings from the posttest survey.

condition for pairs *Binary–Counterexample* ($Z = 4.61$, $p < 0.001$), *Binary–Hint* ($Z = 2.44$, $p < 0.05$), and *Counterexample–Hint* ($Z = -2.82$, $p < 0.01$).

Twelve additional participants from the hint feedback group responded to our questions about preference between counterexamples and hints. The results were evenly split: 5 preferred counterexamples, 5 preferred hints, and 2 were undecided.

In aggregate, the quantitative survey responses show that binary feedback is least preferred along all dimensions, but that hints are not preferred over counterexamples. Therefore, *H6 is only partially supported*.

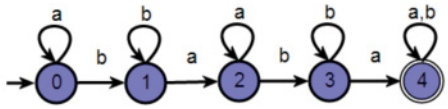
Submission-specific analysis. Figure 6 shows two representative examples of the analyses we performed on particular problems. The example at the top shows a case in which the hint feedback performs better than the counterexample one, whereas the example on the right shows the opposite situation. For each group, the figure shows the percentage of give-ups (students leaving the page), the percentage of full score submissions, and the average score in the immediately subsequent submission.

In the instance at the top, we can see how students who receive the hint feedback when submitting a new attempt are less likely to give up (4% with hint feedback vs. 22% and 10% with binary and counterexample feedback, respectively), receive better scores, and are more likely to find the correct DFA (5% more correct solutions than with binary and counterexample and feedback).

On the other hand, in the instance at the bottom of the figure, we can see how students who receive the counterexample feedback when submitting a new attempt are less likely to give up (9% with counterexample feedback vs. 21% and 27% with binary and hint feedback, respectively), receive better scores (average score of 9 with counterexample feedback vs. 6.32 and 6.62 with binary and hint feedback, respectively), and are more likely to find the correct DFA (12% more correct solutions than with hint feedback and 18% more correct solutions than with binary feedback). Thus, different feedback strategies appear to work for different problems, which we interpret as *support for H7*. We attempt to characterize this difference through our analysis of qualitative survey results.

Descriptive Feedback Better

PROBLEM: strings that contain '*ba*' exactly twice



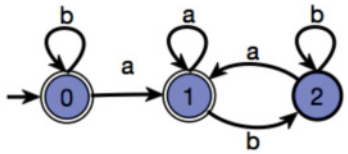
Group	Tot. Students	% Giveup	% Full Score	% Avg Score
Correct/Incorrect	27	22%	45%	7.77
Counterexample	22	10%	45%	7.20
Hint	24	4%	50%	7.82

COUNTEREXAMPLE FEEDBACK:
Your DFA accepts the string '*bababa*' but the correct answer does not.

HINT FEEDBACK:
Your DFA accepts the following set of strings:
{ *s* | '*ba*' appears in *s* **at least** twice }

Counterexample Feedback Better

PROBLEM: strings that don't end with *b* if they start with *a*



Group	Tot. Students	% Giveup	% Full Score	% Avg Score
Correct/Incorrect	28	21%	25%	6.32
Counterexample	11	9%	81%	9.00
Hint	11	27%	27%	6.62

COUNTEREXAMPLE FEEDBACK:
Your DFA rejects the string '*bab*' but the correct answer does not.

HINT FEEDBACK:
Your DFA accepts the following set of strings:
{ *s* | every symbol after the last occurrence of *a* is an *a* }

Fig. 6. Two examples demonstrate that hints and counterexamples are useful for different types of problems.

Qualitative results. In open-ended questions, we elicited reasons why students liked or disliked the feedback they received. Few students in the binary feedback condition responded.

For counterexample students, the main perceived benefit was that the counterexample served as a specific test case that they could examine step-by-step in their DFA: *"I could trace the path of the string which was mentioned in the feedback on my DFA and correct it"*; this was sufficient for some students, as it *"pretty much diagnosed the problem in the DFA perfectly."* However, several students wished for multiple strings to diagnose bigger problems.

Students who saw hints appreciated that they received *"a general idea of what was wrong,"* which *"provided a launching point to think about the problem differently."* In particular, it was helpful to see a characterization of the incorrect solution that they provided in the language of the problem statement: *"I liked when it told me what my DFA did instead of what I thought it did."* However, at times the generated hints confused students because they were either too vague or too complicated: *"Sometimes the languages described were so confusing and convoluted that I couldn't draw any connections from them."*

In the additional questions sent to *Hint* participants, we explicitly asked for a comparison between counterexamples and other types of hints. Students noted that a counterexample may be more concrete than a hint: *"It proves indisputably that the [student's DFA] is wrong."* In comparison, hints offered *"a clearer explanation than [counterexamples]"* and *"explain the broader problem"* by offer information *"about the specific mistake that you are making."* By staying on a conceptual level, hints leave students with more work to translate the type of problem into concrete solution steps. Some students disliked this aspect—*"By keeping it on the bigger scale, it's harder to target the state that's causing the problem"*; whereas others appreciated that they had to reason through the problem: *"While [a hint] is more ambiguous and difficult to interpret, it does a better job of guiding you towards the solution rather than just giving it away."*

5.6. Deployment in the Wild

After our study concluded, an instructor of a discrete mathematics course in Iceland asked us to use AutomataTutor. Using the lessons learned from our user study, we improved the tool's hint feedback. In particular, during our study, we observed that in some cases the tool provided overly complex hints, such as:

—*Your solution accepts the following set of strings:*$\{s \mid$ *if* s *ends with an* a*, the symbol before the last symbol is a* $b\}$.
—*Check the transitions out of states 2 and 4, and you need to change the acceptance condition of one state*.

After manually inspecting numerous hints, we defined a hint to be *too complex* if

—it contains an English description with a parse tree that contains more than seven nodes, or
—it is an edit script proposing more than two edits to the student DFA.

We revised the tool to simply output a counterexample whenever a hint was too complex.

As part of the course, students were given an optional homework assignment in AutomataTutor where they were required to solve eight DFA construction problems. Each student had unlimited number of attempts to solve each problem, and the final grade for the assignment was the sum of the maximum score on each of the eight problems. All students were given the upgraded hint feedback.

The course had 252 students, and 204 completed the homework. A total of 10,710 attempts were submitted, and 8,166 were incorrect solutions for which the student received feedback. On average, each student required five incorrect solutions before solving each problem. The professor reported that in the subsequent two optional homework assignments, only 136 and 120 students, respectively, completed the assignments. The homework was pen-and-paper assignments and did not involve the use of AutomataTutor.

The students were asked to complete the same posttest survey that we used for our main experiment, and 35 students completed it. The feedback was overall considered useful, not confusing, and helped students understand their mistakes. The values of the survey data were in line with those observed for counterexample feedback in our main study.

Question	Mean	Median	Std. Error
Feedback is useful overall	4.40	5	0.16
Feedback is helpful for understanding feedback	4.31	5	0.15
Feedback is helpful for getting to correct DFA	4.34	5	0.15
Feedback is confusing	2.09	2	0.18

In the open-ended questions, students showed great enthusiasm for the tool, and when asked what they liked about the feedback, they explicitly mentioned both counterexamples and hints. Only a couple of students found the feedback at times verbose and believed that it was too informative, almost giving away the solution. Students appreciated the overall value of the tool for learning the concepts (*"I didn't understand Automata when I started these exercises. I do now. Hands on experience like this is way way better than my Discreet Math textbook and/or any lectures."*)

In fall 2014, AutomataTutor was used during undergraduate-level courses at the University of Pennsylvania, the École polytechnique fédérale de Lausanne, and the University of California at San Diego.

6. DISCUSSION

The quantitative results show that, on average, students receiving feedback are able to solve DFA constructions 35% faster than their nonfeedback peers. We were not able to show that students receiving feedback need fewer attempts to solve a problem, and we believe that this is because students who do not receive feedback need to spend more time thinking about the problem without necessarily querying the tool. In terms of scoring, we did not observe significant differences among different groups, and we believe that this is partially because the homework problems were not hard. One of the most interesting results of the quantitative analysis is that students with feedback are much less likely to give up on a practice problem than students who do not receive feedback. This shows how feedback causes students to be more engaged in learning DFA concepts.

The general outcome of the quantitative analysis is that both counterexample and hint feedback help students learning DFA construction faster and with more perseverance than without feedback.

Overall, counter examples slightly outperform feedback comprised of hints; however, we can observe that both types of feedback are useful in practice for different types of problems (see Figure 6). In particular, in some cases, the hint feedback is able to help the students clearly identify their mistake.

Both surveys confirm how counterexample and hint feedback are appreciated by students. An interesting result from the survey is how students are split evenly when asked which feedback they prefer. This result together with that of Figure 6 reinforces our hypothesis that both types of feedback are useful, and perhaps, for the same problem, different students learn better with different types of feedback.

We can also observe that counterexample feedback is a more robust technique and that hint feedback might be confusing in some cases. We believe that the confusion in the hint feedback is mainly because this was a first implementation and, despite the fact that the tool was tested intensively before our experiment, the deployment to hundreds of students with thousands of submissions exercised corner cases that we had not anticipated previously. The posttest survey of our deployment in the wild shows that we were able to identify and address most of these sources of confusion.

We also observed that some types of hints are easier to understand than others. We thus far focused on developing techniques that can generate such hints in the first place, but we have not yet systematically investigated what kind of hints are readily understood and what kind of hints might be confusing (and why). Although we have taken a first step at iteratively improving our hints, a more detailed study of hint types would be valuable future work.

Instructor's perspective. We summarize the key (subjective) observations by the three instructors who have taught theory of computation courses multiple times before and have used our tutoring software for the experiment and subsequent deployment de-

scribed in this article. First, although students did not use interactive tools such as JFLAP in earlier years despite encouragement, the requirement that the homework had to be submitted using the tutoring tool ensured students' participation. Once students started interacting with the software, they were very much engaged with the course material. This positive experience during the first week of the course is extremely valuable for pedagogical reasons. Second, providing even the basic binary (correct vs. incorrect) feedback seems valuable. The average number of practice problems (not required for the homework) solved by the students is surprisingly high. Third, based on personal conversations with the students, they seemed to appreciate both the counterexamples and hints. A valuable side effect was that several (academically stronger) students wanted to know how the feedback was computed, contributing to their increased interest in the course. Fourth, the average grade on the homework turned out to be above 9.7 (out of 10) for all three groups. The average grade on similar homework in the previous year was 8.1 out of 10. Finally, the teaching assistants were very happy that the tool did the grading for them. One concern about the feedback tool is that a confusing feedback should be avoided, and this should be fixed in the future revision of the tool. In summary, all instructors agreed that the tool adds value to teaching and that they will use it in future offerings of the course.

7. CONCLUSION AND FUTURE WORK

This article studies the effectiveness of feedback in the process of learning DFA constructions. We analyze two state-of-the-art techniques—binary and counterexample-based feedback—and our new hint feedback system introduced in this work. To the best of our knowledge, this is the first study on effectiveness of any kind of feedback in DFA constructions. Based on our results, we are able to conclude that both counterexample-based feedback and hint feedback are more effective than binary feedback in students' learning process. Moreover, when looking at individual problems, we can observe how both techniques have their benefits. This was the first run of the feedback tool, and we believe that we now know how to better combine the features of counterexample and hint feedback to improve learning and avoid student confusion. In particular, the main sources of confusion were long language descriptions and badly worded hints on how to fix DFAs. Long descriptions can be replaced by counterexamples, whereas the wording of the hints can be improved. We applied these changes, and based on student feedback we observed that the upgraded tool is generally liked by its users. Moreover, the tool is being used to teach finite automata in many institutes in both Europe and the United States.

ACKNOWLEDGMENTS

We thank Luca Aceto for using our tool in his course and helping us collecting feedback from his students.

REFERENCES

Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. 2013. Automatically generating problems and solutions for natural deduction. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*. 1968–1975. http://dl.acm.org/citation.cfm?id=2540128.2540411.

Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*. 1976–1982. http://dl.acm.org/citation.cfm?id=2540128.2540412

Erik Andersen, Sumit Gulwani, and Zoran Popovic. 2013. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'13)*. ACM, New York, NY, 773–782. DOI: http://dx.doi.org/10.1145/2470654.2470764

John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences* 4, 2, 167–207.

John R. Anderson and Brian J. Reiser. 1985. The LISP tutor: It approaches the effectiveness of a human tutor. *BYTE* 10, 4, 159–175. http://dl.acm.org/citation.cfm?id=3351.3354

Robert B. Ashlock. 1986. *Error Patterns in Computation: A Semi-Programmed Approach*. Merrill Publishing Company.

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1–3, 217–239. DOI:http://dx.doi.org/10.1016/j.tcs.2004.12.030

Manuel Bodirsky, Tobias Gartner, Timo von Oertzen, and Jan Schwinghammer. 2004. Efficiently computing the density of regular languages. In *LATIN 2004: Theoretical Informatics*. Lecture Notes in Computer Science, Vol. 2976. Springer, 262–270. DOI:http://dx.doi.org/openurl.asp?genre=article&issn=0302-9743&volume=2976&spage=262

Beatrix Braune, Stephan Diehl, Andreas Kerren, and Reinhard Wilhelm. 2001. Animation of the generation and computation of finite automata for learning software. In *Automata Implementation*Lecture Notes in Computer Science, Vol. 2214. Springer, 39–47.

Cristina Conati. 2009. Intelligent tutoring systems: New challenges and directions. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*. 2–7. http://dl.acm.org/citation.cfm?id=1661445.1661447

Vladan Devedzic and John Debenham. 1998. An intelligent tutoring system for teaching formal languages. In *Intelligent Tutoring Systems*. Lecture Notes in Computer Science, Vol. 1452. Springer, 514–523. DOI:http://dx.doi.org/10.1007/3-540-68716-5_57

Ethan Fast, Colleen Lee, Alex Aiken, Michael Bernstein, Daphne Koller, and Eric Smith. 2013. Crowd-scale interactive formal reasoning and analytics. In *Proceedings of the 26th Annual Symposium on User Interface Software and Technology (UIST'13)*. 363–372.

Tara Gallien and Jody Oomen-Early. 2008. Personalized versus collective instructor feedback in the online courseroom: Does type of feedback affect student satisfaction, academic performance and perceived connectedness with the instructor? *International Journal on E-Learning* 7, 3, 463–476. http://www.editlib.org/p/23582

Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and Applications* 13, 1, 113–129.

Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing geometry constructions. *SIGPLAN Notices* 46, 6, 50–61. DOI:http://dx.doi.org/10.1145/1993316.1993505

Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2013. Solving geometry problems using a combination of symbolic and numerical reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science, Vol. 8312. Springer, 457–472. DOI:http://dx.doi.org/10.1007/978-3-642-45221-5_31

Katy Jordan. 2014. MOOC Completion Rates: The Data. Retrieved February 4, 2015, from http://www.katyjordan.com/MOOCproject.html.

Kenneth R. Koedinger and Albert T. Corbett. 2006. Cognitive tutors: Technology bringing learning science to the classroom. In *The Cambridge Handbook of the Learning Sciences,* K. Sawyer (Ed.). Cambridge University Press, 61–78.

Jakub Kozik. 2005. Conditional densities of regular languages. *Electronic Notes in Theoretical Computer Science* 140, 67–79. DOI:http://dx.doi.org/10.1016/j.entcs.2005.06.023

Andreas Maletti. 2008. The power of extended top-down tree transducers. *Information and Computation* 206, 9–10, 1187–1196.

Mary A. Mark and Jim E. Greer. 1993. Evaluation methodologies for intelligent tutoring systems. *Journal of Artificial Intelligence in Education* 4, 129–153.

Antonija Mitrovic. 1998. Learning SQL with a computerized tutor. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'98)*. ACM, New York, NY, 307–311. DOI:http://dx.doi.org/10.1145/273133.274318

Antonija Mitrovic, Michael Mayo, Pramuditha Suraweera, and Brent Martin. 2001. Constraint-based tutors: A success story. In *Engineering of Intelligent Systems*. Lecture Notes in Computer Science, Vol. 2070. Springer, 931–940. http://dl.acm.org/citation.cfm?id=646863.707788

David Patterson. 2013. Why Are English Majors Studying Computer Science? Retrieved February 4, 2015, from http://blogs.berkeley.edu/2013/11/26/why-are-english-majors-studying-comp uter-science/.

Leena Razzaq and Neil T. Heffernan. 2006. Scaffolding vs. Hints in the assistment system. In *Proceedings of the 8th International Conference on Intelligent Tutoring Systems*. 635–644.

Susan H. Rodger and Thomas Finley. 2006. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett.

Vinay S. Shekhar, Anant Agarwalla, Akshay Agarwal, Nitish B, and Viraj Kumar. 2014. Enhancing JFLAP with automata construction problems and automated feedback. In *Proceedings of the 7th International Conference on Contemporary Computing (IC3)*. 19–23. DOI:http://dx.doi.org/10.1109/IC3.2014.6897141

Valerie J. Shute and J. Wesley Regian. 1993. Principles for evaluating intelligent tutoring systems. *Journal of Artificial Intelligence in Education* 4, 2–3, 245–271.

Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. 2012. Automatically generating algebra problems. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI-12)*.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 15–26. DOI:http://dx.doi.org/10.1145/2462156.2462195

Matthias F. Stallmann, Suzanne P. Balik, Robert D. Rodman, Sina Bahram, Michael C. Grace, and Susan D. High. 2007. ProofChecker: An accessible environment for automata theory correctness proofs. *ACM SIGCSE Bulletin* 39, 3, 48–52. DOI:http://dx.doi.org/10.1145/1269900.1268801

New York Times. 2012. Instruction for Masses Knocks Down Campus Walls. Retrieved February 4, 2015, from http://www.nytimes.com/2012/03/05/education/moocs-large-courses-open-to-all-topple-campus-walls.html?pagewanted=all

Kurt VanLehn. 1992. Mind Bugs: The origins of procedural misconceptions. *Artificial Intelligence* 52, 3, 329–340.

Kurt VanLehn. 2006. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education* 16, 3, 227–265. http://dl.acm.org/citation.cfm?id=1435351.1435353

Kurt VanLehn. 2011. The relative effectiveness of human tutoring, Intelligent tutoring systems, and other tutoring systems. *Educational Psychologist* 46, 4, 197–221. DOI:http://dx.doi.org/10.1080/00461520.2011.611369

Received May 2014; revised November 2014; accepted January 2015