Automated Grading of DFA Constructions *

Rajeev Alur, Loris D'Antoni
Department of Computer Science
University of Pennsylvania

Sumit Gulwani
Microsoft Research
Redmond

Dileep Kini, Mahesh Viswanathan
Department of Computer Science
Univ. of Illinois at Urbana-Champaign

Abstract

One challenge in making online education more effective is to develop automatic grading software that can provide meaningful feedback. This paper provides a solution to automatic grading of the standard computation-theory problem that asks a student to construct a deterministic finite automaton (DFA) from the given description of its language. We focus on how to assign partial grades for incorrect answers. Each student's answer is compared to the correct DFA using a hybrid of three techniques devised to capture different classes of errors. First, in an attempt to catch syntactic mistakes, we compute the edit distance between the two DFA descriptions. Second, we consider the entropy of the symmetric difference of the languages of the two DFAs, and compute a score that estimates the fraction of the number of strings on which the student answer is wrong. Our third technique is aimed at capturing mistakes in reading of the problem description. For this purpose, we consider a description language MOSEL, which adds syntactic sugar to the classical Monadic Second Order Logic, and allows defining regular languages in a concise and natural way. We provide algorithms, along with optimizations, for transforming MOSEL descriptions into DFAs and vice-versa. These allow us to compute the syntactic edit distance of the incorrect answer from the correct one in terms of their logical representations. We report an experimental study that evaluates hundreds of answers submitted by (real) students by comparing grades/feedback computed by our tool with human graders. Our conclusion is that the tool is able to assign partial grades in a meaningful way, and should be preferred over the human graders for both scalability and consistency.

1 Introduction

There has been a lot of interest recently in offering college-level education to students worldwide via information technology. Several websites such as EdX (https://www.edx.org/), Coursera (https://www.coursera.org/), and Udacity (http://www.udacity.com/) are increasingly providing online courses on numerous topics, from computer science to psychology. Several challenges arise with this new teaching paradigm. Since these courses, often referred to as *massive open online courses* (MOOCs), are typically taken by several thousands of students located around the world, it is particularly hard for the instruction staff to provide useful personalized feedback for practice problem sets and homework assignments.

Our focus in this paper is on the problem of deterministic finite automata (DFA) construction. The importance of DFA in computer science education hardly needs justification. Beside 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 and lexical analyzers, and models of software interfaces. We focus on grading assignments in which a student is asked to provide a DFA construction corresponding to a regular language description. Our main goal is that of automatically measuring how far off the student solution is from the correct answer. This measure can then be used for two purposes: assigning a partial grade, and providing feedback on why the answer is incorrect.

Figure 1 shows five solutions from the ones we collected as part of an experiment involving students at UIUC. The solutions are for the following regular language description:

$$L = \{s \mid s \text{ contains the substring “ab” exactly twice}\}$$

For this problem the alphabet is $\Sigma = \{a, b\}$. Current technologies for this kind of problem [Aut, 2010] simply check whether the DFA proposed by the student is semantically equivalent to the correct one. For this particular example such a technique would only point out that the first solution A_1 is correct, while all the other ones are wrong. Such a feedback, however, does not tell us *how wrong* each solution is.

The four DFAs $\mathrm{A}_2, \mathrm{A}_3, \mathrm{A}_4$, and A_5 in Figure 1 are representative of different mistakes. We first concentrate on A_2. In this attempt the DFA accepts the language

$$L_1 = \{s \mid s \text{ contains the substring “ab” } \textbf{at least} \text{ twice}\}$$

This example shows a common mistake in this type of assignments: the student misunderstood the problem. We need an automated technique that is able to recognize this kind of mistake. The necessary ingredient to address this task is a

*This research is partially supported by NSF Expeditions in Computing awards CCF 1138996, and CCF 1016989.

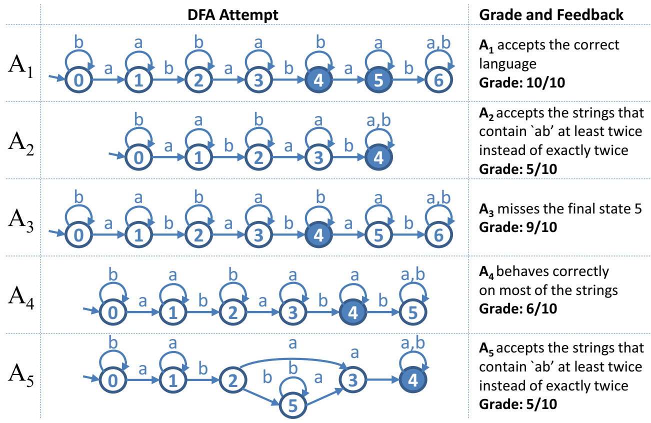

Figure 1: Example of DFA grading. The dark states are final. Column 1 contains the name of the DFA depicted in column 2. Column 3 shows the grade computed by our tool for the DFA with the corresponding feedback.

procedure that, given a DFA A, can *synthesize* a description of the language $L(A)$ accepted by A. Here a question that immediately arises is: what should the description language for $L(A)$ be? Ideally we would like to describe $L(A)$ in English, but such a description cannot be easily subjected to automated analysis. A better option is a logical language that is not only efficient to reason about, but one which also provides a rich set of primitives at a level of abstraction that is close to how language descriptions are normally stated in English. For this purpose, we extend a well-known logic, called *monadic-second order logic* (MSO) [Thomas, 1996; Büchi and Landweber, 1969], that can describe regular languages, and we introduce MOSEL, an MSO-equivalent declarative logic enriched with syntactic sugar. In MOSEL, the languages L and L_1 can be described by the formulas $|\texttt{indOf}\ `ab'| = 2$ and $|\texttt{indOf}\ `ab'| \geq 2$ respectively. Thanks to this formal representation, we can compute how far apart two MOSEL descriptions are from each other and translate such a value into a grade. To compute the distance between two descriptions we use an algorithm for computing the edit distance between trees [Bille, 2005]. We design two algorithms: the first one computes the DFA corresponding to a MOSEL description, and conversely the second one computes the MOSEL description of the language accepted by a DFA. Despite the high computational complexity of such algorithms, through several optimizations, we were able to make them work on examples used to learn automata. We executed the first algorithm on all the DFA assignments appearing in [Hopcroft *et al.*, 2006], achieving running times below 1 second. On the same set of assignments we were able to execute the second algorithm on 95% of the problems, achieving running times below 5 seconds.

The approach presented in the previous paragraph is able to capture a particular class of mistakes. However, several DFAs, such as A_3 in Figure 1, do not fall in this class. A_3 has the full structure of the correct DFA but state 5 is not marked as final. A possible MOSEL description of A_3 is $|\texttt{indOf}\ `ab'| = 2 \wedge \texttt{endWt}\ `b'$ where the second conjunct indicates that all strings must end with a b. This description is syntactically far from the description of L causing the corresponding grade to be too low. This example shows that there should be a metric that tells how far A_2 is from a correct DFA. To address this class of mistakes we introduce a notion of *DFA edit distance* that given a DFA A and a regular language R computes how many states and transitions of A we need to modify in order to obtain a DFA A' that accepts R. Such a computation naturally translates into a grade.

The previous techniques cover two broad classes of mistakes. However, in several cases they are still not enough. The language accepted by the DFA A_4 in Figure 1 has a complicated MOSEL description and the number of operations needed to "fix" A_4 is quite high (more than 5) because we need to add a new state and redirect several edges. However, this solution is on the right track and behaves correctly on most of the strings. The student just did not notice that in state 4 the machine does not necessarily read the symbol a causing strings such as $ababb$ to be rejected. Hence, A_4 correctly rejects all the strings that are not in L, but also rejects "few" more. Following this intuition we introduce a notion of *language density* and we use it to approximate the percentage of strings in Σ^* on which a DFA A misbehaves. Again, such a quantity naturally translates into a grade. We finally combine the three techniques to compute a unique grade.

DFA A_5, despite being syntactically different from A_2, computes exactly the same language as A_2. This similarity might be hard to notice for a human. While our tool, using the same approach as for A_2, assigned the same grade to both the attempts, we observed in our experiments that the same human grader assigned different grades.

We evaluated our tool on DFAs submitted by students at UIUC and compared the grades generated by the tool to those provided by human graders. First, we identified several instances in which two identical DFAs were graded differently by the same grader, while this was not the case for the tool. Second, we observed that the tool produces grades comparable to those produced by humans. In order to check such properties, we used statistical metrics to compare the tool with two human graders, and manually inspected the cases in which there was a discrepancy between the grades assigned by the tool and by the human. The resulting data suggests that the tool grades as well as a human, and we often found that, in case of a discrepancy, the grade of the human was less fair than that of the tool.

2 MOSEL: Declarative Descriptions of Regular Languages

This section provides a preliminary background on DFAs, defines the language MOSEL, and presents algorithms for transforming MOSEL descriptions into DFAs and vice-versa.

2.1 Background on DFAs

A deterministic finite automaton (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. 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 $L(A) = \{s \mid \delta^*(q_0, s) \in F\}$.

2.2 The Language MOSEL

MOSEL was designed with the goal to be (a) *expressive* enough to describe problems that arise in common assignments, (b) *simple* enough to have a close correspondence to natural language descriptions so that the syntactic distance between MOSEL descriptions reflects the distance between their English language descriptions, and (c) *succinct* enough to have small descriptions for common DFA assignments.

The syntax and semantics of MOSEL can be found in the full version of this paper [Alur *et al.*,]. We illustrate features of this language through some examples. The 5 languages described in the first column of Table 1 can be described by the following MOSEL formulas:

$\mathrm{L}_1 = \texttt{begWt}$ 'a' $\wedge$ $|\texttt{indOf}$ 'ab'$|\% \, 2 = 1$**:** strings that start with an a and have and odd number of ab substrings;

$\mathrm{L}_2 = |\texttt{indOf}$ 'a'$| \geq 2 \vee |\texttt{indOf}$ 'b'$| \geq 2$**:** strings that contain at least two a's or at least two b's;

$\mathrm{L}_3 = a @ \{x \mid |\texttt{psLe}\ x| \% \, 2 = 1\}$**:** strings where every odd position is labeled with an a;

$\mathrm{L}_4 = \texttt{begWt}$ 'ab' $\wedge$ $|\texttt{all}| \% \, 3 \neq 0$**:** strings that start with ab and with length not divisible by 3;

$\mathrm{L}_5 = |\texttt{indOf}$ 'ab'$| = 2$**:** strings that contain the substring ab exactly twice; and

$\mathrm{L}_6 = |\texttt{indOf}$ 'aa'$| \geq 1 \wedge \texttt{endWt}$ 'ab'**:** strings that contain the substring aa at least once and end with ab.

2.3 From MOSEL to DFAs

Next, we describe how we transform a MOSEL formula ϕ over an alphabet Σ into the corresponding DFA A_ϕ, such that A_ϕ describes the same language as ϕ. Since MOSEL only adds syntactic sugar to Monadic Second Order Logic (MSO) over strings, MOSEL formulas can be transformed into equivalent MSO formulas. In the first step of our transformation we inductively transform ϕ into an MSO formula ϕ'. Next, we use standard techniques to transform an MSO formula into the corresponding DFA [Henriksen *et al.*, 1995]. Such techniques inductively generate the DFAs corresponding to each sub-formula of ϕ' and then combine such DFAs using automata operations. In the transformation from MSO to DFA, the alphabet is enriched with bitvectors that represent the values of the quantified variables, causing the alphabet to grow exponentially in the number of nested quantifiers.

We implemented the transformation using the Automata library [Veanes and Bjørner, 2012]. This library relies on BDDs to succinctly represent large alphabets, making our transformation efficient in practice. During the inductive transformation we always keep the minimized DFA in order to avoid a blow-up in the number of states. For every exercise E appearing in [Hopcroft *et al.*, 2006], our tool generated the DFA from the corresponding MOSEL description of E in less than 1 second.

2.4 From DFAs to MOSEL

While it is well known that every DFA can be transformed into an equivalent MSO formula, in the standard transformation from DFA to MSO, the distance between MSO representations is not meaningful as it directly reflects the DFA structure rather than the accepted language. Since our goal is to use MOSEL descriptions to capture the syntactic difference between two languages, we use a different approach. Given a DFA A, we use an iterative deepening search to enumerate all possible MOSEL formulas and find the one that describes $L(A)$. As we showed in the previous subsection, MOSEL descriptions of common DFA assignments are succinct. Thanks to such succinctness the brute force approach works adequately for our purpose.

Since typical formulas have small size/width, we use an iterative deepening search on the width of a formula. We next describe some optimizations that make this approach feasible in practice. In order to check whether a formula ϕ is equivalent to the target language, the simplest approach would be that of using the algorithm of Section 2.3 to generate the automaton A_ϕ corresponding to ϕ and then run a DFA equivalence algorithm. Such a procedure does not scale in practice. In our implementation we first check whether ϕ behaves correctly on some selected inputs and, only if it passes this test, we compute the DFA for ϕ and perform the equivalence test.

Next, we describe the procedure for generating the input test set. Given the input DFA $A = (Q, q_0, \delta, F)$, we compute two sets P and N of positive and negative examples respectively, that is, $P \subseteq L(A)$, and $N \cap L(A) = \emptyset$. For every two states $q_1, q_2 \in Q$, we add the string $s_{q_1} p_{q_1}$ to P, and the string $s_{q_2} n_{q_2}$ to N, where $\delta^*(q_0, s_{q_i}) = q_i$ and $\delta^*(q_i, p_{q_i}) \in F$ and $\delta^*(q_i, n_{q_i}) \notin F$. Similarly, for every two states $q_1, q_2 \in Q$, and for every $a \in \Sigma$, we add the string $s_{q_1} a p_{\delta(q_1,a)}$ to P and the string $s_{q_2} a n_{\delta(q_2,a)}$ to N. The sets P and N contain at most $|Q|^2|\Sigma|$ strings. Finally, for every DFA $B = (Q_B, q_0^B, , \delta_B, F_B)$ such that $|Q_B| \leq |Q_A|$, it is enough to test Q_B on the test sets P and N in order to check whether $L(A) \neq L(B)$. Therefore, if the minimal DFA corresponding to ϕ has fewer states than A, we detect the inequivalence of ϕ and A by simply testing the formula on P and N. In our experiments this technique yields a 300X speed-up.

Next, we optimized the algorithm for common cases in which the target formula is a disjunction or a conjunction. When we come across formulas ϕ_1, ϕ_2 which pass the negative test set N during the enumeration, we check if $\phi_1 \vee \phi_2$ passes the positive test set P and follow it with an equivalence check if needed. Conjunctions are handled in a similar manner. This optimization permits to identify formulas of big width early in the search. For example, the tool was able to synthesize the formula describing the language in Figure 2 in less than a second. Without this optimization it would have taken more than 5 minutes to reach the corresponding formula in the enumeration.

Finally, we implemented several pruning techniques. First, using syntactic properties of the input DFA we can avoid enumerating some formulas. For example, if the DFA has only loops of size 1, and 2, the formula $|\_\_| \% \, 3 = \_\_$ can be statically ruled out. Secondly, we statically remove several terms that are equivalent to other terms of smaller width ($\neg\neg\phi$). These pruning techniques caused a 500X speed-up on our test set of 30 examples. We tested the algorithm on the exercises in [Hopcroft *et al.*, 2006] for which the alphabet only contained two elements.. For each exercise, given the corresponding DFA solution A, we were able to generate a MOSEL description of A in less than 5 seconds for 95% of the problems. In few cases, the MOSEL description was too big and the tool was not able to generate it.

3 An Algorithm for Grading DFA Constructions

We next address the problem of grading a student attempt. Given a target language L_T, and a student solution A_s, we need a metric that tells us how far A_s is from a correct solution. Based on our experience related to teaching and grading DFA constructions, we identified three classes of mistakes:

Problem Syntactic Mistake: the student gives a solution for a different problem (see (2) and (5) in Figure 1);

Solution Syntactic Mistake: the student omits a transition or a final state (see (3) in Figure 1); and

Problem Semantic Mistake: the solution is wrong on a small fraction of the strings (see (4) in Figure 1).

We investigated three approaches that try to address each class. First, we use the classic notion of tree edit distance [Gao *et al.*, 2010] to compute the difference between two MOSEL formulas. Secondly, we introduce a notion of DFA edit distance to capture the distance between DFAs. Last, we use the concept of regular language density to compute the difference between two languages when viewed as sets.

3.1 Problem Syntactic Distance

The following metric captures the case in which the MOSEL description of the language corresponding to 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. Given a MOSEL formula ϕ, we call T_ϕ its parse tree. Given two ordered trees t_1 and t_2, their tree edit distance $\mathrm{TED}(t_1, t_2)$ 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 label of a node n;

node deletion: given a node n with parent n', 1) remove n, 2) place the children of n as children of n', inserting them in the "place" left by n; and

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'.

We use the algorithm in [Gao *et al.*, 2010] to compute TED. Next, we compute the distance $\mathrm{D}(\phi_1, \phi_2)$ between two formulas ϕ_1 and ϕ_2 as $\mathrm{TED}(\mathrm{T}_{\phi_1}, \mathrm{T}_{\phi_2})$. Finally, we compute $\mathrm{WTED}(\phi_1, \phi_2) \stackrel{\text{def}}{=} \mathrm{D}(\phi_1, \phi_2)/|\mathrm{T}_{\phi_2}|$, where $|\mathrm{T}|$ is the number of nodes in T. In this way, for the same number of edits, less points are deducted for languages with a bigger description. Since we are ultimately interested in grading DFAs, given a DFA A_s we use the procedure proposed in § 2.4 to compute the formula ϕ_{A_s} corresponding to A_s.

Example 1 Consider the language L corresponding to $\phi \stackrel{\text{def}}{=} |\texttt{indOf}\ `ab\text{'}|\%\ 2 = 1 \wedge \texttt{begWt}\ `a\text{'}$ over the alphabet $\Sigma = \{a, b\}$. Let's assume the student provides the DFA A' that implements the language $\phi' \stackrel{\text{def}}{=} |\texttt{indOf}\ `ab\text{'}|\%\ 2 = 1 \vee \texttt{begWt}\ `a\text{'}$, where $\wedge$ has been replaced by $\vee$. The problem syntactic distance will yield the following values: $\mathrm{TED}(\phi', \phi) = 1$, and $\mathrm{WTED}(\phi', \phi) = 1/9$. In this case applying one node relabeling is enough to "fix" $\mathrm{T}_{\phi'}$. We omit the parse tree of ϕ which contains 9 nodes. □

3.2 Solution Syntactic Difference

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. We define the notion of DFA edit distance. Given two DFAs A_1, A_2, we say that the difference between A_1 and A_2, $\mathrm{DFA\text{-}D}(A_1, A_2)$ 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 $a \in \Sigma$; and

state relabeling: given a state q, add it or remove it from the set of final states.

To take into consideration the severity of a mistake based on the difficulty of the problem, we compute the quantity $\mathrm{WDFA\text{-}D}(A_1, A_2) \stackrel{\text{def}}{=} \mathrm{DFA\text{-}D}(A_1, A_2)/k + t$, where k and t are, respectively, the number of states and transitions of A_2.

Example 2 Consider the DFA A_3 in Figure 1 where state 5 is mistakenly marked as non-final. A_1 is the correct solution for the problem. In this case

$\mathrm{DFA\text{-}D}(A_3, A_1) = 1 \quad \mathrm{WDFA\text{-}D}(A_3, A_1) = 1/12{+}6 = 1/18$

since applying one state relabeling will "fix" A_3. □

In the tool we compute this metric by trying all the possible edits and checking for equivalence with a technique similar to the one presented in Section 2.3.

A similar distance notion *graph edit distance* [Bille, 2005]. However this metric does not take into account the language accepted by the DFA.

3.3 Problem Semantic Difference

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, we define *density difference* to be

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

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, the density difference is not always defined, as the limit may not exist.

Example 3 Consider the languages L_A corresponding to $|\texttt{all}|\%\ 2 = 0$ and L_B corresponding to $\texttt{true}$ (i.e. Σ^*) over the alphabet $\Sigma = \{a, b\}$. The limit DEN-DIF(L_A, L_B) is not defined since it keeps oscillating between 0 and 1. □

In practice we compute the approximated density A-DEN-DIF$(L_1, L_2) \stackrel{\text{def}}{=} (\sum_{n=0}^{2k} E(n))/2k+1$, where k is the number of states of the minimum DFA representing L_2. This approximation is not precise, but it is very helpful for capturing the cases in which the student forgot a finite number of strings in its solution (for example only ε).

Example 4 Consider the DFAs A_1 and A_4 in Figure 1 and their respective languages $L(A_1)$ and $L(A_4)$. In this case A-DEN-DIF$(L(A_4), L(A_1)) = 0.09$. This value is the one used to compute the grade shown in Figure 1. □

Similar notions of density have been proposed in the literature [Bodirsky *et al.*, 2004; Kozik, 2005]. These definitions have good theoretical foundations, but, unlike our metric, they are undefined for most DFAs.

3.4 Combining the Approaches

The aforementioned approaches need to be combined in order to compute the final grade. We are aware of the many machine learning techniques that could be used for combining the three features, but instead, we decide to use a simple combination function for the following reason : 1) in the future we would like to extract feedback information from the computed grade, and 2) in general, only one of the three feature succeeds in computing a positive grade.

Next, we provide the general schema of the combining function. First, each deduction v, which ranges between 0 and 1, is scaled to a new value v' using a formula of the form $v' := (v + c)^2 - c^2$ where c is a constant. We used a training set of 60 manually graded attempts to identify the constants c for the combining function. Finally, we pick the metric which awarded the highest score.

4 Experimental Evaluation

The aim of our experiment is to evaluate to what extent the grades given by our tool and those given by human instructors agree. To do so we collected around 800 attempts at DFA construction questions by students taking a theory of computation course for the first time. For each problem we had two instructors and our tool grade each attempt separately. In order to see how well the tool does we compare statistics that reveal variation between human graders and variation between a human grader and our tool. To measure the extent of agreement between two graders we employ Pearson's correlation coefficient. The correlation coefficient is a number between -1 and 1. A value of 1 indicates that the paired points are linearly related with a positive slope. When this quantity is closer to 1 it indicates that the two measurements being compared tend to vary together in the same direction.

In order to obtain a basis for comparing the correlation coefficients we also see how a naive grader would perform with respect to human graders. There could be many ways to define a naive grader. A simple one that we consider uses the following grading scheme: (i) it awards near maximum (9 or 10) marks to the correct solutions, and (ii) for incorrect solutions it deducts marks based on the number of states that

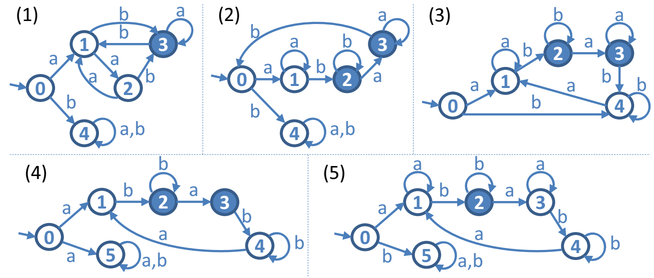

Figure 2: Selected attempts for the language $L_1 = \{s \mid s$ starts with a and has odd number of ab substrings$\}$.

are lacking or in excess and adds a small random noise. We summarize the resulting calculations in Table 1.

Detailed Analysis In the following we only consider the first problem where the language is $L_1 = \{s \mid s$ starts with a and has odd number of ab substrings$\}$. The first column in the averages reads 0.99 for H_1-H_2 meaning that H_1 has awarded, on average, 1 point more than H_2. The next two columns show that H_1 is on average closer to the naive grader N and to the tool T than it is to H_2. However, the standard deviation for H_1-N (2.62) is greater than that for H_1-T (1.99), which means that the grades given by our tool show a lot less variation, and are in fact closer to H_1 more often than N. The Pearson correlation coefficients shows that the degree of correlation between the tool T and H_1 (0.83) is clearly better than that between N and H_1 (0.65), and at the same time comes very close to the degree of correlation between two human graders H_1 and H_2 (0.87).

We say that two graders agree on an attempt with a threshold of t if the grades given by the two graders do not differ by more than t. The plot on the right shows 3 curves. Each curve compares two graders and displays how the percentage of problems on which they agree increases with the threshold varying from 0 to 10. The three curves compare T, H_2 and N against H_1, and it is easy to see that our tool T comes to an agreement much faster than N. More surprisingly, the tool also comes to an agreement faster than H_2.

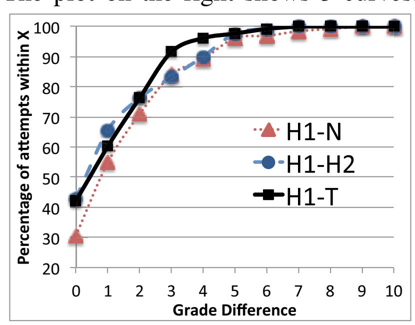

Figure 2 shows five cases in which either the human graders and the tool have a discrepancy. In case (1) the computed language L'_1 is described by the formula $|\texttt{indOf}$ 'b'$|\% \ 2 = 1 \land \texttt{begWt}$ 'a'. Since the MOSEL descriptions of L_1 and L'_1 are similar, the tool gives a high grade for this attempt. However, L'_1 has an easier construction than L_1. We attenuate these "false high grades" by deducting extra points when the size of the minimal DFA corresponding to the student solution has less states than the target DFA. Case (2) shows a DFA for which the language density is low, causing the tool to award this attempt with 7 points. One can argue that this grade is too high for such a DFA. However, the same

Problem	Attempts Tot.	Attempts Dis.	Average H_1-H_2	Average H_1-T	Average H_1-N	Standard Deviation H_1-H_2	Standard Deviation H_1-T	Standard Deviation H_1-N	Pearson Correlation H_1-H_2	Pearson Correlation H_1-T	Pearson Correlation H_1-N
$L_1 = \{s \mid s$ starts with a and has odd number of ab substrings$\}$	131	108	0.99	0.54	0.22	2.06	1.99	2.62	0.87	0.83	0.65
$L_2 = \{s \mid s$ has more than 2 a's or more than 2 b's$\}$	110	100	-0.66	0.85	0.26	1.80	2.44	2.71	0.90	0.80	0.75
$L_3 = \{s \mid s$ where all odd positions contain the symbol $a\}$	96	75	-0.52	0.86	-1.38	1.61	2.67	3.84	0.90	0.74	0.31
$L_4 = \{s \mid s$ begins with ab and $\|s\|$ is not divisible by 3$\}$	92	68	0.40	1.32	0.36	1.68	2.78	2.48	0.81	0.71	0.61
$L_5 = \{s \mid s$ contains the substring ab exactly twice$\}$	52	46	0.02	0.19	0.29	2.01	1.88	3.23	0.71	0.79	0.49
$L_6 = \{s \mid s$ contains the substring aa and ends with $ab\}$	38	31	-0.50	-1.34	-1.5	2.42	2.90	3.70	0.76	0.63	0.34

Table 1: Comparing grades given by humans and tool. The grades were between 0 and 10. H_1 and H_2 denote the two human graders, T the tool, and N the naive grader. A-B denotes the difference between the graders A and B when grading each individual attempt. For each problem L_i the table shows in the order: the number of student attempts, the number of distinct attempts, the average difference, the standard deviation, and the Pearson's correlation between single attempt grades.

DFA as (2) was submitted by multiple students, and, while the tool always awarded 7 points, both human graders were inconsistent: H_1 graded five identical attempts with 4,5,7,7, and 7 points, while H_2 awarded 1,1,2,3, and 8 points. Case (3) shows a DFA for which the DFA edit distance yields a too "generous" grade. For this DFA it is enough to remove the transition $\delta(0, b)$ in order to obtain a DFA that accepts L_1. However, in this case the mistake is deeper than a simple typo. Case (4) shows a DFA for which the human awarded too high a score. Even though this DFA has several syntactic mistakes, H_1 awarded the same grade as for attempt (5), where only one final state is missing. For case (5), where state 3 was mistakenly marked as non-final, both H_1 and H_2 lacked in consistency. H_1 awarded different grades from 8 to 10 for 7 identical attempts.

Strengths of the tool Inspecting the data for the 131 attempts for the language L_1 we observed the following: 1) in 6 cases one of the human graders mistakenly assigned the maximum score to an incorrect attempt; 2) in more than 20 out of 34 cases in which T and H_1 were disagreeing by at least 3 points, H_1, after reviewing the attempt, agreed with the grade computed of T; and 3) in more than 20 cases at least one of the human graders was inconsistent: i.e. two syntactically equivalent attempts were graded differently by the same grader.

Limitations The tool suffers two types of limitations: behavioral and structural. The former type concerns the failure of the grading techniques on some particular examples. An example is the attempt (3) of Figure 2 where a small DFA edit distance did not reflect the severity of the mistake. As for the structural limitations, our techniques are crafted to perform well on problems appearing in theory of computation books [Sipser, 1996; Hopcroft *et al.*, 2006]. Such techniques do not scale for DFAs with large alphabets or many states. Moreover the MOSEL edit distance fails when the language does not admit a succinct description. We do not believe these to be actual limitations, because such situations typically do not arise in undergraduate level DFA constructions. However, we plan on extending our tool to deal with these cases.

5 Related Work

JFLAP [Rodger and Finley, 2006] is a mature system used widely for teaching automata and formal language theory. To the best of our knowledge JFLAP does not offer an automatic grading of DFA constructions.

Benedikt et.al have proposed a notion of regular language repairing [Benedikt *et al.*, 2011]. Their approach could also be used for defining a grading metric, however, we believe that such metric would not be a good fit for our purposes since it cannot be associated with any natural feedback.

Singh et.al. have proposed an automatic grading framework for programming problems [Singh *et al.*, 2013]. Given a error model in the form of expression rewrite rules, their system uses SAT based techniques to find the minimal number of corrections that can fix the student solution. Our syntactic edit distance approach is similar to this proposal.

There has been a lot of work in the AI community for building automated tutors for helping novice programmers learn programming by providing feedback about semantic errors [Adam and Laurent, 1980; Murray, 1987]. Such technologies share similar ideas with our tool in the way they measure distance from a correct solution.

6 Conclusion

We investigated the problem of grading DFA constructions. First, we introduced MOSEL, a declarative logic able to provide succinct and natural descriptions of regular languages appearing in automata theory textbooks. Second, we provided algorithms for transforming MOSEL descriptions into DFAs and vice-versa. Last, we presented three grading techniques based on three different classes of mistakes.

We evaluated our tool on DFAs submitted by real students and compared the grades generated by the tool to those provided by human graders. The results are encouraging and show that the tool grades as well as a human. In fact we plan on further engineering our tool, deploy it, and soon use it in real courses. We also plan on storing problem solutions in a database in order to speed up the grading and fix the few cases in which the tool does not assign a fair grade. We believe our techniques can provide foundations for generating automated feedback for students, and can be also adapted for other types of constructions. In particular we are working on automatically generating an inductive proof that a given DFA accepts a given language. Finally, this tool will not replace the need for teaching assistants (TAs), but it automates a task that often consumes several TA-hours. Although we only address a small problem, once a satisfactory grading tool for this problem is constructed, it will be used for a long time, since the concept of DFA will be always taught as it is now.

References

[Adam and Laurent, 1980] Anne Adam and Jean-Pierre H. Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1-2):75–122, 1980.

[Alur *et al.*,] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions, full version. http://www.cis.upenn.edu/ lorisdan/papers/ijcai13long.pdf.

[Aut, 2010] Automata Tutor. http://pub.ist.ac.at/automata_tutor/, 2010.

[Benedikt *et al.*, 2011] Michael Benedikt, Gabriele Puppis, and Cristian Riveros. Regular repair of specifications. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 335–344, Washington, DC, USA, 2011. IEEE Computer Society.

[Bille, 2005] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.

[Bodirsky *et al.*, 2004] Manuel Bodirsky, Tobias Grtner, Timo von Oertzen, and Jan Schwinghammer. Effciently computing the density of regular languages. In Martin Farach-Colton, editor, *In Proceedings of the 6th Latin American Symposium, Buenos Aires, Argentina, 2004*, volume 2976 of *Lecture Notes in Computer Science*, pages 262–270. Springer, 2004.

[Büchi and Landweber, 1969] J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *Journal of Symbolic Logic*, 34(2):166–170, 1969.

[Gao *et al.*, 2010] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, January 2010.

[Henriksen *et al.*, 1995] Jesper G. Henriksen, Ole J.L. Jensen, Michael E. Jrgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, LNCS 1019*. Springer-Verlag, 1995.

[Hopcroft *et al.*, 2006] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Kozik, 2005] Jakub Kozik. Conditional densities of regular languages. *Electronic Notes in Theoretical Computer Science*, 140:67–79, November 2005.

[Murray, 1987] William R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3:1–16, 1987.

[Rodger and Finley, 2006] Susan H. Rodger and Thomas Finley. *JFLAP - An Interactive Formal Languages and Automata Package*. Jones and Bartlett, 2006.

[Singh *et al.*, 2013] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. 2013.

[Sipser, 1996] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[Thomas, 1996] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[Veanes and Bjørner, 2012] Margus Veanes and Nikolaj Bjørner. Symbolic automata: the toolkit. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 472–477, Berlin, Heidelberg, 2012. Springer-Verlag.