Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 248 (2009) 47–56
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Learning Algorithm for Deterministic Finite Automata using JFLAP [1]

Mikel Alecha and Montserrat Hermo

Dpto. de Lenguajes y Sistemas Informáticos.
Facultad de Informática. Universidad del País Vasco.
Paseo Manuel de Lardizbal, 1
20018-San Sebastián, Spain.

Abstract

The JFLAP package is a free, interactive visualization, and teaching tool for formal languages. JFLAP is based on the principle that a picture of a concept can be easier to understand than a textual representation. With the help of this package, we implement Dana Angluin's algorithm which is able to learn Deterministic Finite Automata. The use of JFLAP allows users to visualize each step in the process of learning. The protocol used by the algorithm is called exact learning from membership and equivalence queries. This protocol was also introduced by Dana Angluin, who showed that her learning algorithm discovers the unique minimum automaton coherent with the queries in an efficient running time.

Keywords: Exact Learning Model, Deterministic Finite Automata, JFLAP

1 Introduction

Dana Angluin [2] considers the problem of learning a representation class $\mathcal{R}$ for a concept class $\mathcal{C}$ by allowing the learning algorithm to make specific kinds of queries about the unknown target concept $c \in \mathcal{C}$. For example, if $\mathcal{R}$ is the class of Deterministic Finite Automata DFA over alphabet Σ, then the concept class is the set of Regular Languages over Σ, and a target concept is a particular regular language $L \subseteq \Sigma^*$.

Among the types of queries Dana Angluin considers are the following.

– *Membership:* The input to a membership query is an element w whose answer will be YES if $w \in c$ or NO if $w \notin c$. In our example $w \in \Sigma^*$ and the answer will be YES as long as $w \in L$.

– *Equivalence:* The input is a hypothesis $h \in \mathcal{R}$ and the output is YES if $c_h \equiv c$, where c_h is the concept represented by h, and NO otherwise. If the answer is NO

[1] This work has been partially supported by Spanish project TIN2007-66523

doi:10.1016/j.entcs.2009.07.058

an element x in the symmetric difference of c_h and c is returned. If $x \in c - c_h$ then x is called a *positive* counterexample. Otherwise, it is a *negative* counterexample. In our example the hypothesis is a particular deterministic finite automaton $M \in$ DFA. If $L(M) \not\equiv L$, then the counterexample must be a string of Σ^*.

The particular selection of a counterexample is assumed to be arbitrary, that is, a successful learning algorithm must work no matter what counterexample is provided. Moreover, the algorithm must output a representation of a hypothesis equivalent to the target concept c, and has to do so in time polynomial w.r.t. the size of the *minimum* representation for c, and the length of the largest counterexample received. When this occurs, it is said that $\mathcal{R}$ *is efficiently exactly learnable with membership and equivalence queries.*

In our example, the learning algorithm must output a particular $M \in$ DFA, such that $L(M) = L$ and it must work in time polynomial w.r.t. the size of the minimum automaton that recognizes L and the length of the largest string $w \in \Sigma^*$ given as counterexample.

In [1] it was showed that the DFA class is efficiently exactly learnable with membership and equivalence queries. Moreover, the algorithm finds out the unique minimum automaton equivalent to the target language.

In this paper, we present an implementation of the learning algorithm for DFA following the ideas in [3,5]. What is new is the use of the JFLAP tool [4] for visualizing the steps of the algorithm. Our algorithm asks the user for strings that must be answered depending on whether they are in the target language (obviously, the precondition of the algorithm is that the target language must be regular). Additionally, when the algorithm asks the user for a particular automaton, it provides a visual picture of it through the JFLAP tool. This is very useful because it makes the task of finding a counterexample if the automaton given is not equivalent to the target language easier.

The application is publicly available on `http://www.sc.ehu.es/jiwhehum2/DFA/dfa.jar/` and the readers can execute it as long as they have a JAVA virtual machine.

The rest of the paper is organized as follows. In the next section we explain how the algorithm is able to learn DFA using a particular example. Two other examples are presented in section 3. Section 4 describes some implementation details. Finally, section 5 concludes.

2 The Learning Algorithm Trough an Example

Since Dana Angluin's algorithm and its different versions are sufficiently known in this field and have been also referenced in the bibliography of this paper, we are not going to give an in-depth analysis of the structure of the used algorithm and the way it works. Rather than that, we are going to give a brief summary of its method of operation and then we will proceed to show the basics of its execution in our application through an example and a series of captions.

Let us begin with a general overview of what the algorithm does. As previously

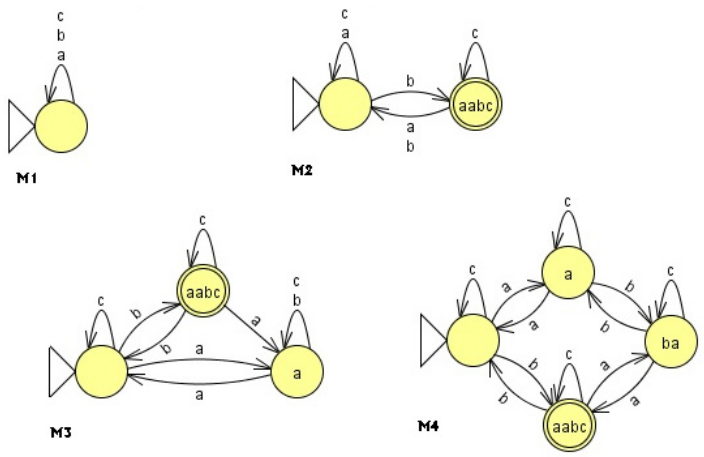
said, the whole process involves requesting specific information from an external source (in our case, the user) and processing it to expand the knowledge of the concept. More specifically, the algorithm starts by gathering basic information of the target (a particular regular language). Once it has done this, it enters in a loop consisting of two phases: first, with guidance from the user, who has to answer a set of membership queries, it builds an hypothesis automaton; once it has done this, it shows the hypothesis to the user and queries whether that hypothesis exactly recognizes the target or not. If it does, the process ends; otherwise, it asks the user for information on why its hypothesis is different from the target, processes it and begins the loop again.

Now, the example we are going to use to see a more detailed view is the automaton built when the target is the language

$$L = \{w \in \{a, b, c\}^* : |w|_a \bmod 2 \ = 0 \ \wedge \ |w|_b \bmod 2 \ = 1\}.$$

That means L contains the strings over alphabet $\{a, b, c\}$ that have an even number of a's and an odd number of b's.

The first step is to define the alphabet for the application, something it will request before starting the execution of the learning algorithm. Once a working alphabet has been entered, the algorithm starts by asking whether the empty string is accepted or not. In other words, whether this string is inside of the target language. The responses to membership queries are put in through a simple interface consisting of *Yes*/*No* buttons. The application stores the answers given for later use, so that the user is not asked the same membership query more than once. Additionally, all given answers are also accessible by the user, to be checked if necessary.



In our case the answer to the first query must be *No*. Then the hypothesis automaton M_1 (see figures above) is built and displayed so that the user can compare it with L. At this point the algorithm needs to know whether $L(M_1) = L$. If that is not the case the algorithm requires a counterexample to justify this statement,

which the user has to input through an interface for equivalence queries. In our example, a valid counterexample for M_1 could be the string *aabc*.

Next, the algorithm needs to process the information gathered from this counterexample into its main data structure: *the classification tree*. Another series of membership queries will be made to the user for this purpose in the same manner as before, also storing the results for further reusability later on. Once the algorithm considers it has gathered enough information in the classification tree, it is ready to once again start the process of building an hypothesis automaton M_2. M_2 has one more state than M_1.

In our case, before constructing M_2, the strings the algorithm asks for and their respective answers are:

a	*b*	*c*	*aabca*	*aabcb*	*aabcc*
No	*Yes*	*No*	*No*	*No*	*Yes*

As $L(M_2) \neq L$, the user has to provide another counterexample. For instance the string *ab*. With this information the algorithm proceeds to ask all these membership queries:

cb	*aabcab*	*aabcbb*	*aa*	*aab*	*abb*	*ac*	*acb*
Yes	*No*	*Yes*	*No*	*Yes*	*No*	*No*	*No*

Next, the algorithm produces M_3. If the user decides to give the string *baa* as counterexample, then the new set of membership queries is as follows:

ba	*bab*	*aabcaa*	*aba*	*aca*	*babb*	*baba*	*bac*	*bacb*	*baca*
No	*No*	*Yes*	*Yes*	*No*	*No*	*No*	*No*	*No*	*Yes*

Now, the hypothesis M_4 verifies that $L(M_4) = L$ and the process can finish if the user confirms it. As we have said before, the algorithm finds out the unique minimum automaton equivalent to the target language.

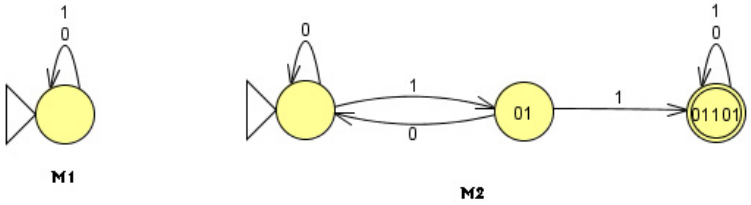
To finish this section, it is worthwhile to note that the number of times the main loop of the learning algorithm is executed is exactly $size(M_4)$. That is, the number of states in M_4. This is because the algorithm always starts by considering a one-state automaton and increases the state count by one during each loop. In addition, each execution of the main loop requires to update the classification tree with the help of a single counterexample. If the length of this counterexample is m, then this process requires at most m operations. Therefore, we have $size(M_4)$ main loop executions, each of which requires $O(size(M_4) + n)$ operations, where n is the length of the longest counterexample.

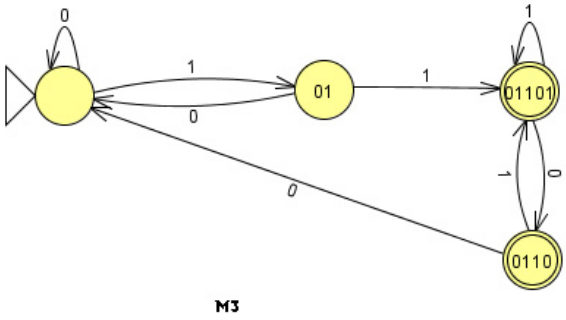
3 Two other examples

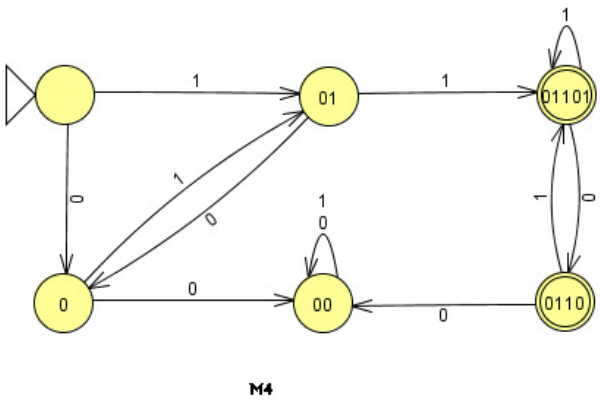
The available application [2] keeps all the answers supplied by the user through the execution. Therefore, as we have said before, the user is not asked the same membership query more than once. Moreover, the algorithm always builds a new hypothesis automaton which is coherent with previous counterexamples.

For instance, let L be the language over alphabet $\{0,1\}$ whose strings contain the substring 11 but do not contain the substring 00. Formally

$$L = \{w \in \{0,1\}^* : 11 \subseteq w \wedge 00 \not\subseteq w\}$$







[2] http://www.sc.ehu.es/jiwhehum2/DFA/dfa.jar

The algorithm starts asking whether the empty string is in L or not. In this case the answer is *No* and the hypothesis built by the algorithm is M_1.

A valid counterexample for M_1 is the string 01101. If this is the counterexample provided by the user, then the algorithm begins to ask a sequence of membership queries.

0	1	011010	011011	01	011	11	010	0101
No	*No*	*Yes*	*Yes*	*No*	*Yes*	*Yes*	*No*	*No*

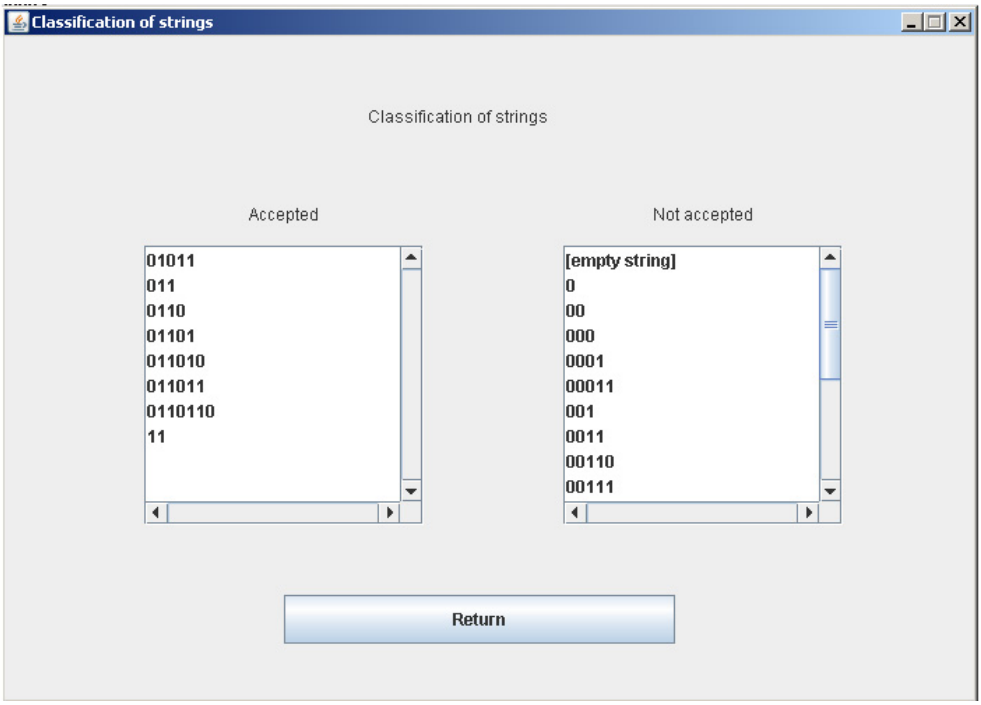
Next the algorithm builds M_2. If the user decides to give the string 011001 as counterexample, then the new sequence of membership queries is

0110	01100	0110100	010110
Yes	*No*	*No*	*Yes*

Once the algorithm shows the hypothesis M_3, if the user gives as counterexample the string 00110, then the membership queries asked by the algorithm are the following

00	001	0011	01011	010011	000	0001	00011	00111	010011
No	*No*	*No*	*Yes*	*No*	*No*	*No*	*No*	*No*	*No*

At this moment, the algorithm shows the hypothesis M_4 which is the minimum automaton recognizing L.



Regarding the fact that all the answers provided by the user are stored during the process, it should be noted that the user has the possibility of displaying all this information at anytime.

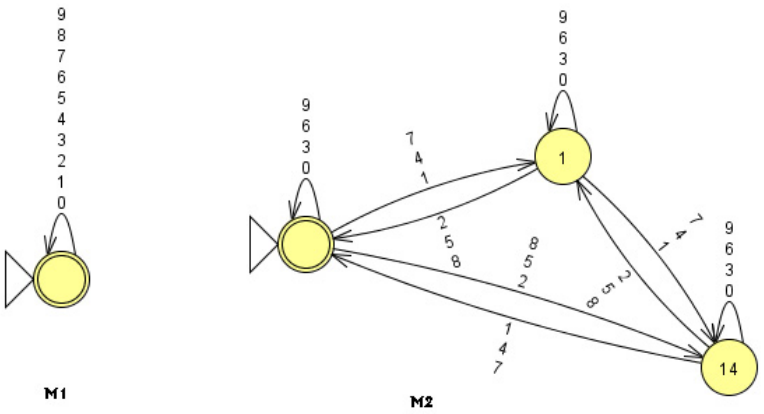
For instance, in the case of this example, the above figure shows the information provided by the algorithm when the user asks for the classification of strings.

A third example finishes this section. The target language is the set of natural numbers that are multiple of 3. Remember that to find out if a number is divisible by 3, we must add up all the digits in the number and check if the sum is divisible by 3. For example: the sum of the digits of 12123 is $1+2+1+2+3=9$, As 9 is divisible by 3, therefore 12123 is too. The target language can be defined as follows

$$L = \{n \in \{0,1,2,3,4,5,6,7,8,9\}^* : n \bmod 3 = 0\}$$

According to this definition the string 000360 represents the natural 360 and belongs to L. We consider 0 (and consequently any string of the form 0^i with $i \geq 1$) in the target language, as well as the empty string.

The application shows M_1 as hypothesis and, once the string 14 is given as counterexample, and after making a sequence of membership queries, it shows M_2 that recognizes L.



Between M_1 and M_2, the sequence of membership queries is as follows

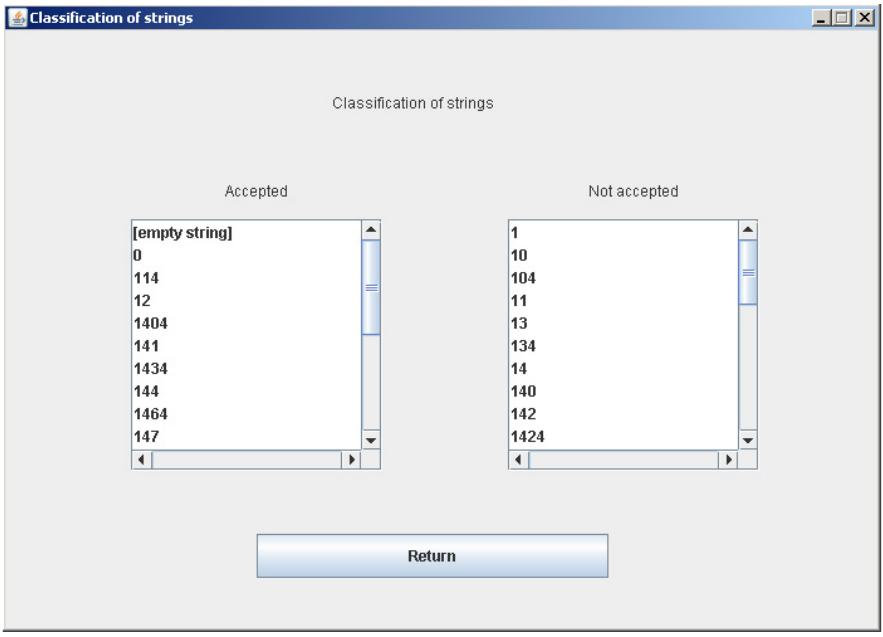
0	1	2	3	4	5	6	7	8	9	140	141
Yes	*No*	*No*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*	*No*	*Yes*

142	143	144	145	146	147	148	149	24	44	54
No	*No*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*Yes*	*No*	*Yes*

74	84	1404	1424	1434	1454	1464	1484	1494	10	104
No	*Yes*	*Yes*	*No*	*Yes*	*No*	*Yes*	*No*	*Yes*	*No*	*No*

11	114	12	13	134	15	16	164	17	174	18	19	194
No	*Yes*	*Yes*	*No*	*No*	*Yes*	*No*	*No*	*No*	*Yes*	*Yes*	*No*	*No*

Next figure shows the current classification of strings when the user asks for this information



4 Implementation

The algorithm has been implemented in the Java language for one main reason: to be able to make use of the necessary JFLAP tool source code modules. This is also followed by machine compatibility reasons, but one of the purposes of this tool was to facilitate the user's comprehension of the hypothesis automata returned by the algorithm through their graphical representation.

The implementation is divided in two sections based on this: on one side lie the main algorithm procedures, the basic data structures and the user interaction interface; on the other side lies the graphical representation interface for the hypothesis automata, which connects the previous section with the JFLAP modules that allow management of the automaton data structure and those that portray the graphical representation on-screen.

The data structures are fairly simple: the strings over the alphabet that are queried or given as counterexamples are managed through a vector and the classification tree through a binary tree. All of them have been built within their respective Java classes, along with their own constructors and methods, except some such as the automaton data structure, which remains a JFLAP class.

The class responsible for handling the graphical representation is a more complex one, as it must be the one that handles both the requests of the running algorithm and the communication with the different JFLAP modules. Much of this communication, such as the events of reshaping the representation of the displayed automaton, is controlled by the JFLAP modules once they have been properly linked. Other operations, for instance the assembly and disassembly of the automaton, require to be handled explicitly and to use direct calls to the various methods of the automaton data structure.

Moreover, the application can handle the addition of many optional features beyond the execution of the algorithm and the graphical display of automata. One of the earliest implemented ones was the ability to memorize the answers to previously answered membership queries, which reduced the work on the user's side considerably. Another important one was to allow the users to access the classification of said answered queries through a new interface, which allows them to view those values when necessary.

Other features are the option to reset the application at any time and the ability to save the on-screen built automaton to a file recognized by the JFLAP tool. This last feature is particularly useful in the case the user would like to further use an automaton structure learned through our application. It makes it possible to save an automaton, load it through JFLAP and then edit it with the various automata-related tools.

By default, the application is set to answer equivalence queries by itself whenever possible. Based on the information learned and stored up to the point of an equivalence query, it automatically checks the built hypothesis automaton to see whether any of the memorized words can act as a counterexample. This way, if any of the words do serve as a counterexample, the query is never shown to the user, who would need to answer equivalence queries only when the application acknowledges it requires input from the user to continue.

It is still possible for the user to deactivate this function, however. The learning process becomes less efficient this way, but it allows the user to follow an execution closer to the actual learning algorithm [5]: the learning iterations become more clearly separated from each other. It also allows users to experiment with different counterexample inputs of their own, rather than the ones chosen by the program. It is possible to turn this function on and off at any time during the execution through the options menu.

One of the latest implemented features was also a system to undo user actions during the learning process. This feature is particularly useful, as it saves users the need to restart the learning process from scratch in case they accidentally answer a query incorrectly or input an undesired counterexample. This was rather necessary

for users who want to work with big and complex automata, whose learning process involves a large sum of queries and fairly long words.

Still, due to the internal structure of the algorithm and the implementation around it of the application, the undo function only becomes available to the user after the process' first counterexample is given. It is not possible to undo actions previous to that point once it is activated, either. The source of this is the way the algorithm initializes its data structure with the information it receives up to that point, following a different process from the rest of the learning process' iterations.

The resulting interface allows the user to receive queries from the algorithm, to input answers and counterexamples, to observe the graphical representation of an automaton and to change the on-screen position of its different states for its better visibility and comprehension. It also allows the user to follow the learning process step-by-step in a didactic manner.

5 Conclusions

JFLAP integrates visual and interactive tools allowing users to gain hands-on experience with theoretical concepts. In the case of the learning algorithm, this package makes the interaction of the user with the learner easier. JFLAP has, among others, the ability of comparing finite automata; transforming nondeterministic finite automata and regular expressions into minimal deterministic finite automata; deciding whether a particular string is recognized by a finite automaton. This means that even the package would allow us to present the learning algorithm interacting directly with JFLAP itself.

JFLAP started as a series of tools with students working under the direction of Susan Rodger (`http://www.jflap.org/`). We plan to send her the implementation of the learning algorithm to add a new functionality to JFLAP.

References

[1] D. Angluin, *Learning regular sets from queries and counterexamples*, Information and Computation, Vol. 75, 1987, 87–106.

[2] D. Angluin, *Queries and concept learning*, Machine Learning, Vol. 2, 1988, 319–342.

[3] J.L. Balcázar, J. Dáz, R. Gavaldá and O. Watanabe, *Algorithms for Learning Finite Automata from Queries: A Unified View*. Chapter in Advances in Algorithms, Languages, and Complexity, D.-Z. Du and K.-I. Ko (eds.), Kluwer Academic Publishers, 1997, 73–91.

[4] S.H. Rodger and T.W. Finley, *JFLAP. An interactive Formal Languages and Automata Package*, Jones and Bartlett Publishers, 2006. http://www.jflap.org/.

[5] M.J. Kearns and U.V. Vazirani, *An introduction to Computational Learning Theory*, MIT Press, 1997 (Second printing).