10th International Colloquium on Grammatical Inference

ICGI 2010

16th September 2010, Valencia (Spain)

ZULU Workshop

Proceedings edited by José M. Sempere

Program of the ZULU Workshop

15:00 – 17:10 **ZULU Workshop**
Chair: Colin de la Higuera

Introduction

S. Eisenstat, D. Angluin
Learning Random DFAs with Membership Queries: the GoodSplit Algorithm

F. Luque, G. Infante-Lopez
A Lazy L Algorithm for Learning Regular Languages from Queries*

M. N. Irfan, R. Groz, C. Oriat
Optimising Angluin Algorithm L by Minimising the Number of Membership Queries to Process Counterexemples*

P. García, M. Vázquez de Parga, D. Lopez
Some ideas on active learning using membership queries

J. M. Vilar
Next question, please. Two query learning algorithms participate in the Zulu challenge

17:10 – 17:30 Coffee Break

17:30 – 19:00 **B. Balle**
Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries

F. Howar, B. Steffen, M. Merten
Finding Counterexemples Fast: Lessons learned in the Zulu challenge

Discussion

The ZULU competition

Zulu is an active learning competition. Participants are to build algorithms that can learn deterministic finite automata (DFA) by making the smallest number of membership queries to the server/oracle.

Motivations

When learning language models, techniques usually make use of huge corpora that are unavailable in many less resourced languages (such as the Zulu language). One possible way around this problem is to interrogate an expert with a number of chosen queries, in an interactive mode, until a satisfying language model is reached. In this case, an important indicator of success is the amount of energy the expert has spent in order for learning to be successful. A nice learning paradigm covering this situation is that of Query Learning, introduced by Dana Angluin.

In the field of Grammatical Inference, Query Learning was thoroughly investigated to learn deterministic finite automata (DFA). As negative results, it was proved that DFA could not be learned from just a polynomial number of membership queries nor from just a polynomial number of strong equivalence queries. On the other hand, algorithm L* designed by Angluin, was proved to learn DFA from a polynomial number of both membership and equivalence queries. These results yield several successfull applications in Robotics, Games and Agents Technologies, Information Retrieval, Hardware and Software Verification.

However, what has not been hardly studied is how to optimise the learning task by trying to minimize the number of queries while making queries for which the Oracle's work and answers are simple. These are strong motivations for stemming research in the direction of developing new interactive learning strategies and algorithms, that is the aim of this competition.

Scientific committee of ZULU

- Dana Angluin, Yale University, USA
- Leo Becerra Bonache, Universidad de Tarragona, Spain
- François Coste, IRISA, Rennes, France
- Alex Clark, Royal Holloway University of London, United Kingdom
- Ricard Gavaldà, Universidad Politecnica de Barcelona, Spain
- Colin de la Higuera, Université de Nantes, France
- Jean-Christophe Janodet, Université de Saint-Etienne, France
- Aurélien Lemay, Université de Lille 3, France
- Laurent Miclet, ENSAT Lannion and IRISA, France
- Tim Oates, University of Maryland, USA
- Anssi Yli-Jyra, Helsinki, Finland
- Menno van Zaanen, Tilburg University, The Netherla

Credits

- David Combe
- Colin de la Higuera
- Jean-Christophe Janodet
- Myrtille Ponge

DFA generation by Gowachin - François Coste, INRIA.
Circular icons by Ben Gillbanks

Contents

S. Eisenstat, D. Angluin
Learning Random DFAs with Membership Queries: the GoodSplit Algorithm……1

F. Luque, G. Infante-Lopez
A Lazy L Algorithm for Learning Regular Languages from Queries*……5

M. N. Irfan, R. Groz, C. Oriat
Optimising Angluin Algorithm L by Minimising the Number of Membership Queries to Process Counterexemples*……9

P. García, M. Vázquez de Parga, D. Lopez
Some ideas on active learning using membership queries……13

J. M. Vilar
Next question, please. Two query learning algorithms participate in the Zulu challenge……15

B. Balle
Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries……19

F. Howar, B. Steffen, M. Merten
Finding Counterexemples Fast: Lessons learned in the Zulu challenge……22

Learning Random DFAs with Membership Queries: the GoodSplit Algorithm

Sarah Eisenstat[1⋆] and Dana Angluin[2⋆⋆]

[1] CSAIL, MIT, Cambridge MA USA
seisenst@mit.edu
[2] Computer Science Department, Yale University, New Haven CT USA
dana.angluin@yale.edu

Abstract. We consider the problem of learning a random deterministic finite state acceptor using membership queries and review known lower and upper bounds for the problem. We describe our entry in the Zulu competition, GoodSplit, and present empirical results on its performance.

1 Introduction

The problem of learning deterministic finite acceptors has been studied in many contexts using a variety of different data sources and criteria of successful learning. The Zulu competition [1] presented the task of learning to predict the behavior of a randomly generated deterministic finite state acceptor given a limited number of membership queries. After making (at most) that many queries, the algorithm is evaluted by determining the fraction of labels of a randomly generated sequence of test strings it predicts correctly. The given number of queries may not be sufficient for exact identification of the target machine, making it important to use partial information well.

Let Σ denote a finite alphabet of symbols and Σ^* the set of all finite strings over Σ. Concatenation of u and v is denoted by $u \cdot v$ or uv. The length of a string w is denoted $|w|$. The empty string is denoted by λ. The notations Σ^ℓ, $\Sigma^{\leq \ell}$ and $\Sigma^{< \ell}$ denote the set of strings of symbols from Σ of length exactly ℓ, at most ℓ and less than ℓ, respectively.

Domaratzki, Kisman and Shallit [2] give results on how many distinct languages are accepted by automata with n states over an alphabet of k symbols. Their results imply

$$(k-1)n \log_2 n + \Theta(n)$$

bits are necessary and sufficient to specify a language from this set, which gives a lower bound on the number of membership queries for exact identification.

The assumption that the target DFA is randomly generated may be helpful in finding efficient learning algorithms. A result of Korshunov [3] (quoted in Trakhtenbrot and Barzdin [4], p. 276) implies that a test set consisting of all strings of length at most about $\log_k \log_2 n$ is sufficient to distinguish all inequivalent pairs of states in almost all DFAs of n states with input alphabet of size k. For $d = \epsilon + \log_k \log_2$, querying all suffixes in $\Sigma^{\leq d}$ for each state and state successor entails about $k^{1+\epsilon} n \log_2 n$ queries.

⋆ Research supported by the Akamai MIT Presidential Graduate Fellowship.
⋆⋆ Research supported by the National Science Foundation under Grant CCF-0916389.

2 The GoodSplit Algorithm

GoodSplit maintains the following data. The set of queries asked so far is $A = A_0 \cup A_1$, where A_0 is the queries answered 0 and A_1 is the queries answered 1. This information is cached and consulted before a new query is made; its maintenance is implicit below. The variable ℓ is the maximum length of suffixes currently being considered (initially 0.) D is a set of strings known to lead to pairwise distinct states of the target machine (initially $\{\lambda\}$.) P contains the strings in D and all of their one-symbol extensions, i.e., $P = D \cup (D \cdot \Sigma)$. Two strings u_1 and u_2 are **consistent** if for every $v \in \Sigma^{\leq \ell}$, if both $u_1 v$ and $u_2 v$ have been queried, then they are both in A_0 or both in A_1, that is, both were answered the same way.

Until the query limit is reached, GoodSplit repeats the following sequence of steps.

1. The algorithm updates D and P as follows. For each $s \in D$ and $a \in \Sigma$, if sa is not in P, then sa is added to P and queried. For each $s \in (P - D)$ that is not consistent with any $s' \in D$, s is added to D. These actions are repeated until neither P nor D changes.
2. At this point, each string in $(P - D)$ is consistent with one or more strings in D, and the algorithm attempts to refine the possible identifications for each string down to one. For a string s, let D_s denote the set of strings $s' \in D$ that are consistent with s. For each string $s \in (P - D)$, as long as $|D_s| > 1$, the algorithm greedily chooses a suffix $t \in \Sigma^{\leq \ell}$ and queries st.
   The greedy choice is made as follows. For $b \in \{0, 1\}$ let
   $$v_b(s, t) = |\{s' \in D_s : s't \in A_b\}|,$$
   that is, the number of s' in D_s such that $s't$ has been queried and answered b. Then $t \in \Sigma^{\leq \ell}$ is chosen to maximize
   $$v(s, t) = \min\{v_0(s, t), v_1(s, t)\},$$
   (with ties broken randomly.) (Because pairs of elements of D_s are inconsistent, there will be at least one t with $v(s, t) \geq 1$. The greedy choice maximizes the minimum number of possible identifications that could be eliminated by the query.)
3. The algorithm decides whether or not to increase the current maximum suffix length ℓ using the following heuristic. If the fraction of all pairs $(s, t) \in (P - D) \times \Sigma^{\leq \ell}$ such that $st \in A$ is greater than 90%, then ℓ is increased by 1.
4. The algorithm then makes additional membership queries according to the following heuristic. For $\lceil |D|/2 \rceil$ random choices of $(s, t) \in (P - D) \times \Sigma^{\leq \ell}$ such that $st \notin A$, the algorithm queries both st and $s't$ such that $s' \in D$ is consistent with s. The algorithm then returns to step 1.

When the query limit is reached, the algorithm executes step (1) once more and then constructs a DFA hypothesis as follows. The states are the strings in D and the initial state is λ. The state s is accepting if $s \in A_1$, rejecting if $s \in A_0$, and its label is chosen randomly if $s \notin A$. The transition function $\delta(s, a)$ maps to sa if this string is in D. Otherwise, $\delta(s, a)$ is chosen to be $s' \in D$ such that sa and s' are consistent. (If more than one s' is consistent with sa, then one is chosen at random.)

3 Results and Discussion

We present results of empirical tests of GoodSplit with randomly generated (by rejection sampling) minimized DFAs with n states and k alphabet symbols. Each test set is 1800 randomly generated strings. For given n and k we generate ten DFAs, and for each DFA we consider the median of ten trials of GoodSplit on that DFA. To illustrate the learning curve of GoodSplit, we show the fraction of correct labels of the test set as a function of the number of queries asked by GoodSplit for $n = 200$ states and $k = 5$ in Figure 1.

We also compare the number of queries needed by GoodSplit to achieve 100% correctness on the test set with the lower bound of $(k-1)n\log_2 n$ for the number of queries needed for exact identification. In Figures 2(a), 2(b), 2(c) we show the median of the medians for $k = 2, 5, 15$ for n from 20 to 200 by 10. In Figures 2(d), 2(e), 2(f) we show the median of the medians for $n = 20, 70, 120$ for k from 2 to 20. The "bumps" in the latter three curves correspond to transitions between different integral distinguishabilities and are related to the 90% parameter in step (3) of the algorithm. Much remains to be done to understand and improve the performance of GoodSplit.

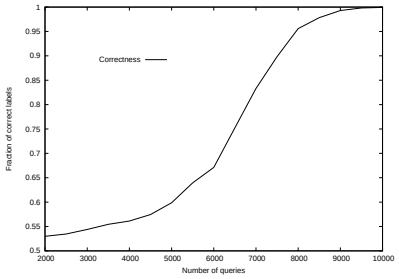
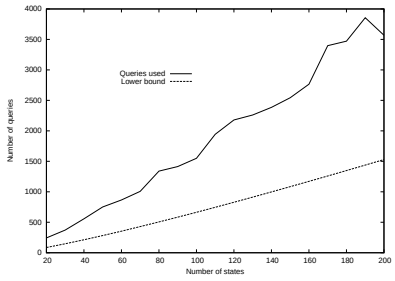


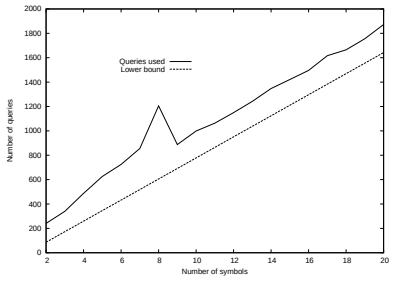
Fig. 1. GoodSplit: Learning curve for $n = 200, k = 5$

References

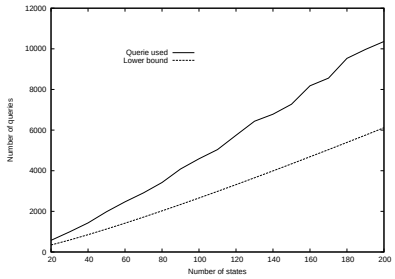
1. David Combe, Colin de la Higuera, Jean-Christophe Janodet, and Myrtille Ponge. Zulu – active learning from queries competition, 2010. `http://labh-curien.univ-st-etienne.fr/zulu/index.php`.
2. Michael Domaratzki, Derek Kisman, and Jeffrey Shallit. On the number of distinct languages accepted by finite automata with n states. *Journal of Automata, Languages and Combinatorics*, 7(4), 2002.
3. A.D. Korshunov. The degree of distinguishability of automata. *Diskret. Analiz.*, 10(36):39–59, 1967.
4. B. A. Trakhtenbrot and Ya. M. Barzdin. *Finite Automata: Behavior and Synthesis*. North Holland, Amsterdam, 1973.



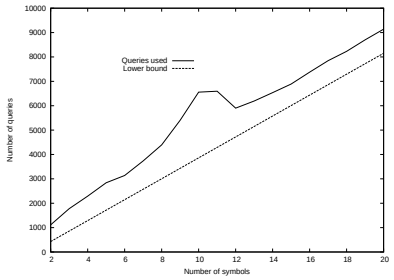
(a) $k = 2$



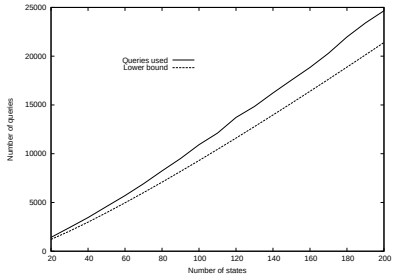
(d) $n = 20$



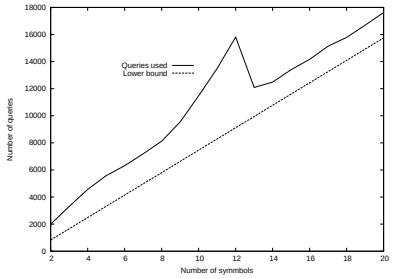
(b) $k = 5$



(e) $n = 70$



(c) $k = 15$



(f) $n = 120$

Fig. 2. GoodSplit: Queries to achieve 100% correctness

A Lazy L^* Algorithm for Learning Regular Languages from Queries

Franco M. Luque

Grupo de Procesamiento de Lenguaje Natural
Universidad Nacional de Córdoba & CONICET
Córdoba, Argentina
francolq@famaf.unc.edu.ar

Abstract. We present an algorithm for the problem of learning regular languages through membership queries and possibly equivalence queries This algorithm is a modification of Dana Angluin's L^* algorithm (1987). Our algorithm saves queries at each step, allowing the execution of more steps than L^* with the same amount of queries, and therefore building more confident hypothesis. The algorithm, along with some other improvements and adaptations, took part in the automata learning competition Zulu under the name LTA*.

1 Introduction

In the interactive learning problem, a teacher or oracle is available to answer certain types of queries that the learner can do. Here, we address the problem of learning regular languages with membership queries and equivalence queries. The membership queries allow the learner to ask if a particular element is a member of the target language. The equivalence queries allow the learner to ask if a certain Deterministic Finite Automaton (DFA) correctly describes the target language. If it doesn't, the oracle answers with a counterexample, this is, an element that is in the difference between the DFA's language and the target language.

In [1], Dana Angluin presents the L^* algorithm that learns from membership and equivalence queries, and proves that the regular languages can be learned with this algorithm. At each step, L^* mantains a table that, when it satisfies certain properties, encodes an hypothetical DFA acceptor for the target language.

In this work, we present a learning algorithm based on the L^* algorithm. Our algorithm saves queries by avoiding filling some unnecessary entries of the L^* table. This allows us to execute more steps and build bigger tables than the original algorithm with the same amount of queries. These bigger tables will be incomplete but will contain more relevant information, therefore providing more confident DFA hypothesis.

An implementation of our algorithm took part in the Zulu interactive learning competition [2]. In Zulu, the goal is to learn a DFA only with a limited number of membership queries. After doing the queries, the learner must label

(a)

	λ
λ	1
a	0
b	0
aa	1
ab	0
ba	0
bb	1

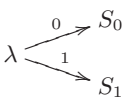
(b)

	λ	a
λ	1	0
a	0	1
b	0	0
aa	1	0
ab	0	0
ba	0	0
bb	1	0

(c)

	λ	a
λ	1	?
a	0	1
b	0	0
aa	1	?
ab	0	0
ba	0	0
bb	1	?

(d)



(e)

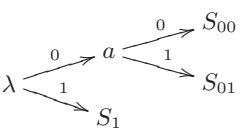


Fig. 1. (a) An inconsistent table. (b) L^* resolution of the inconsistency. (c) Lazy L^* resolution. (d) State tree before the lazy resolution and (e) after it.

a given test set, and its score is given by the number of correctly labeled elements. As a baseline, a modified version of the L^* algorithm is offered, where the equivalence queries are simulated through a sequence of membership queries. The implementation of our algorithm was based on this baseline, doing the same simulation of the equivalence queries, and introducing some other optimizations specific to the Zulu setting.

2 The Lazy L^* Algorithm

The main modification that we introduce to the L^* algorithm is the way that inconsistencies are solved. Consider, for instance, the target language $L = \{s \in \{a, b\}^* | \exists n, m : |s|_a = 2n, |s|_b = 2m\}$, that are the strings with an ever number of a's and an even number of b's, and consider the L^* table shown in Fig. 1 (a). This table is inconsistent, because the prefixes $s_1 = a$ and $s_2 = b$ have the same row (0), so they go to the same state, but s_1a and s_2a have different rows (1 and 0 respectively), so they go to different states. This means that the prefixes s_1 and s_2 must go to different states, so their rows must be differentiated by adding a new suffix as column.

The way that the L^* algorithm solves this inconsistency is by adding the new column a and filling all the column doing the necessary membership queries (or using previously known information). In the example, we must do four queries. The resulting table has different rows for s_1 and s_2, as can be seen in Fig. 1 (b).

What we observe here is that the goal of the algorithm is to solve the inconsistency by splitting the state represented by the row 0 into two new states. The rows corresponding to other states, in the example only the row 1, have nothing to do and there is no need to fill the new column for them. So, our algorithm would solve the inconsistency as shown in Fig. 1 (c), doing only two queries, two less queries than the original algorithm.

This new way of treating the inconsistencies leads to a very different behavior of the algorithm in at least two senses. In first place, the states represented in the table can be organized as the leaves of a binary tree that encodes the history of splittings. The internal nodes of the tree are labeled with suffixes (column headers), starting from the root λ, and the transitions are labeled with

0 and 1 indicating the value for that suffix in the row. The state tree for our example before and after the resolution of the inconsistency is shown in Fig. 1 (d) and (e) respectively. In second place, there will be no more unclosures in the algorithm. The row of any new prefix added by a counterexample will be resolved by traversing the state tree, so it will surely fall in a previously known state.

3 Conservative Counterexample Treatment

Another important improvement is the treatment of the counterexamples returned by the equivalence queries (or its emulation). This improvement of the algorithm is independent of the lazyfication presented in the previous section and can be used with the original L^* algorithm.

When L^* finds a counterexample, it adds as rows in the table the counterexample and all its prefixes, and depending on the length of the counterexample sometimes this implies lots of queries. The addition of the new rows forces the appearance of inconsistencies and/or unclosures in the table that must be solved in subsequent steps.

However, we observe that the goal must be to force the appearance of an inconsistency or an unclosure doing the least possible number of queries. A way to do this is to add the prefixes in increasing size, starting from the shortest prefix of the counterexample that is not already in the table, stopping the first time that an inconsistency or unclosure appear. In the case of the lazy L^* algorithm, it will surely stop with an inconsistency because there are no unclosures.

An interesting observation is that this way of treating counterexamples implies that usually the counterexamples themselves are not added to the table. Therefore, it is not guaranteed that the next hypothetical DFA will correctly classify the counterexample. So, before doing a new equivalence query, we can check if the DFA now correctly classifies all the previously seen counterexample, maybe finding a still valid counterexample without doing any query.

4 Other Improvements

When equivalence queries are not available, as in the Zulu competition, they can be emulated by sampling random strings and doing membership queries until a counterexample is found. In the L^* implementation offered as baseline for Zulu, the positive examples, this is, those strings that are not counterexamples, and their correct classification, are not saved. But we observe that a positive example may become a counterexample for a subsequent hypothetical DFA, because it will be different from the current hypothesis. So, along with the reusage of counterexamples presented in the previous section, we can also save all the positive examples and re-check them before doing the equivalence query emulation.

There is another improvement specific to the Zulu competition, where the oracle only offers a limited number of membership queries. The baseline version of L^* for Zulu, when the limit is reached, just terminates and keeps the last hypothetical DFA that was obtained from the last time that the table was closed

and consistent. But we observe that it may happen, and will usually do, that we already found a counterexample for this DFA and the limit was reached before reaching a new consistent and closed version of the table. So, these steps and their associated queries should not be lost and should influence in some way the final DFA. We took what we believe is the simplest approach to do this. When the limit of queries is reached, we take the last hypothetical DFA and see how it classifies every string queried after its construction. Then, we modify the DFA adding for each misclassified string a new ad-hoc path of states that corrects its classification.

5 Discussion

In order to test the algorithms before the Zulu competition, we set up a benchmark with problems of different alphabet size and number of states. We saw that both the lazyfication described in Sect. 2 and the counterexample treatment described in Sect. 3 improved significantly the performance over the baseline L^*.

However, our final algorithm, called LTA*, did not do well in the Zulu competition. We believe that the main weakness of our algorithm is in the simulation of equivalence queries, that we left untouched from the provided baseline L^*. In this point is where heuristics can be introduced to generate strings that are more likely to be counterexamples. The few simple heuristics we had the time to try did not clearly improve the algorithms, at least according to our benchmark.

Besides the usage of heuristics for counterexample generation, we have several other ideas to improve our algorithm. These include even more lazyness, even more conservative counterexample treatments, query guessing with correction through counterexamples, etc.

Our modifications does not change in spirit the original L^* algorithm. We believe that the same theoretical properties of L^* proved in [1] can be proved for our algorithm, including correctness and termination.

Acknowledgments

LTA may stand for Lazy Table Analysis but it is actually a tribute to Diego Armando Maradona and his famous quote “La Tenés Adentro”.

This work was supported in part by grant PICT 2006-00969, ANPCyT, Argentina.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
2. Combe, D., de la Higuera, C., Janodet, J.C.: Zulu: an interactive learning competition (2010)

Optimising Angluin Algorithm L^* by Minimising the Number of Membership Queries to Process Counterexamples

Muhammad Naeem Irfan, Roland Groz and Catherine Oriat
LIG, Computer Science Lab,
Grenoble Universities,
38402 Saint Martin d'Hères, France
{Irfan, Groz, Oriat}@imag.fr

Abstract. Angluin algorithm L^* is a well known approach for learning unknown models as minimal deterministic finite automata (DFA) in polynomial time. It uses concept of oracle which presumably knows the target model and comes up with a counterexample, if the conjectured model is not correct. This algorithm can be used to infer the models of software artefacts and a cheap oracle for such components uses random strings (built from inputs) to verify the inferred models. In such cases and others the provided counterexamples are rarely minimal. The length of the counterexample is an important parameter to the complexity of the algorithm. The proposed technique tends to reduce the impact of non minimal counterexamples. The gain of the proposed algorithm is confirmed by considering a set of experiments on DFA learning.

Keywords: counterexample; deterministic finite automata; deterministic finite automata inference.

1 Introduction

The algorithm L^* by Angluin [1] was designed to learn the unknown models as DFAs. This algorithm operates with the assumption of existence of a minimally adequate oracle (teacher) which can always reply with a counterexample, if the learnt model is not correct. When we apply this algorithm to learn the models of software black box components, the existence of such an oracle is a strong assumption, which is not met in the general case. To circumvent the lack of oracle, we try to find counterexamples with *random walks*, as was suggested by Angluin. A counterexample is a sequence of inputs, whose output for the target model is different from the output for the learnt conjecture. We randomly construct a string of inputs which is provided in parallel to the unknown model and to the conjecture until we find a discrepancy between both outputs. The counterexamples produced with this naïve method are not minimal; this phenomenon affects the complexity of the algorithm negatively. The variants of Angluin's initial algorithm for treating counterexamples have been proposed, in particular by Rivest and Schapire [2], Maler and Pnueli [3] and Shahbaz [4].

In this paper, we present *Suffix1by1* algorithm [5] which has gain over the above mentioned counterexample processing methods, when counterexamples provided by

the oracle are not minimal. The motive behind *Suffix1by1* strategy is to process counterexample efficiently by adding only the sequences to the observation table from counterexample, which help refining the learnt model. Contrary to Rivest and Schapire algorithm, this method respects the important properties of prefix and suffix closure for the observation table.

2 Background

Angluin [1] proposed an algorithm which learns minimal deterministic finite automata DFA of target models in polynomial time with the complexity $O(|I|\ mn^2)$, where n is the number states of minimum DFA, m is the length of the longest counterexample and I is the set of inputs. The data structure used by the algorithm to record the observations is a matrix whose rows are indexed by $S \cup S.I$ and columns by E, where S is a prefix-closed set of inputs, and E a suffix-closed set of inputs. This matrix is called observation table, it records OT(s, e) the observations on whether $s.e$ is a member of the language to be recognized by the DFA where $s \in S \cup S.I$ and $e \in E$. Once the observation table is complete and satisfies the properties (closure and consistency) required by the algorithm then a model is constructed from the observations. If the conjectured model is not correct, the *oracle* replies with a counterexample. If we have a counterexample CE, the original method of processing the counterexamples adds all the prefixes of CE to S and then extends the table. After adding the prefixes of CE with this method the observation table can be not closed or inconsistent. Rivest and Shapire [2] proposed a new version by introducing some amendments in L^* by requiring the relaxation on prefix and suffix closure properties. Relaxation on such properties may classify the treated counterexample again as counterexample. Maler and Pnueli [3] in their amendment, proposed to add the suffixes of counterexample to the columns instead of rows of observation table to avoid inconsistency. Their method adds all the suffixes of the CE to columns E of the observation table. The counterexample processing method by Shahbaz [4] divides the CE as $u.v$ where u is the longest prefix in $S \cup S.I$ and adds all the suffixes of v to E. This method tends to improve the complexity of the algorithm, which is reduced to $O(|I|\ qn^2)$, where q is the length of suffix of the counterexample used to add new columns. We have $q < m$, because the length of prefix removed from the counterexample CE is always greater than 1, i.e. $| u | \geq 1$, as $S \cup S.I$ or rows indices of the observation table always include I, so CE will always have a prefix of length at least 1 matching rows indices of the observation table.

Two rows $r_1, r_2 \in (S \cup S.I)$ in the observation table are said to be *equivalent*, iff $\forall e \in E$ we have $T(r_1,e)=T(r_2,e)$, which is denoted by $r_1 \cong_E r_2$.

An observation table is *closed* iff $\forall r_1 \in S.I \backslash S\ \exists r_2 \in S$ such that $(r_1 \cong_E r_2)$. It is *consistent* iff $\forall r_1, r_2 \in S$ *if* $(r_1 \cong_E r_2) \Rightarrow \forall i \in I\ (r_1 . i \cong_E r_2 . i)$.

We denote by $suffix^j$(CE), the suffix of CE of length j and $prefix^j$(CE), the prefix of CE of length j. For example, if we have a counterexample sequence CE = $c.b.a.a.a.c$, then $suffix^4$(CE) = $a.a.a.c$ and $prefix^4$(CE) = $c.b.a.a$.

3 Improved Technique

While searching the counterexamples, the oracle implementations can traverse the states in the unknown model which are already present in the conjecture before reaching a state which is not learned yet. Such traversals may always not be minimal. The intent for the presented technique is to reduce the effect of counterexamples generated from less efficient traversals, on learning. This method processes the suffixes from the counterexample CE by increasing length up to the suffix which forces refinement (makes the observation table not closed). On finding such a suffix it stops adding the suffixes to the columns of the observation table and makes the table closed and consistent. It is important to note that this method adds rows to S only when the observation table is not closed, so S always has only non-equivalent rows, which results in the reduction of consistency check. After processing till a suffix of the counterexample which forces refinement and finding a new conjecture which has at least one more state than the previous conjecture, we check whether CE is still a counterexample for the newly built conjecture. If this is the case, we repeat the above method to process CE and we continue until the counterexample can no longer help in refining the conjectured model.

Input: Pre-refined table *(S, E , T)*, CE
Output: Refined observation table *(S, E , T)*
begin
  while CE is a counterexample **loop**
    for $j = 1$ **to** $|CE|$ **loop**
      if $suffix^j$ (CE) $\notin E$ **then**
        add $suffix^j$ (CE) to E
        construct the output queries for the new column
        fill (S, E , T) by running output queries
        if (S, E , T) is not closed **break** for loop
      end if
    end for
    make *(S , E , T)* closed
    construct the conjecture M
  end while
  return refined observation table *(S , E , T)*
end

This variant may look like a minor and contrived change to Shahbaz method for minimal counterexamples. However, in the case of non-minimal counterexamples, it can have a dramatic impact, as can be observed from the experiments. The rationale behind proposed improvement comes from the observation that random walks can cycle through states of the target model before reaching new states which are not present in the learned model. Therefore, only the tail parts of such counterexamples actually correspond to discriminating sequences, so considering suffixes of such counterexamples makes it possible to get rid from unproductive parts of traversals.

If p is the length of the longest suffix added by *Suffix1by1* counterexample processing method, then the complexity of algorithm is given as: $O(|I|pn^2)$, and $p \leq q < m$, where q is the length of suffix of CE added to the columns of the observation table by Shahbaz method and m is the length of the CE.

4 Experiments

To have a significant set of experiments and assess the impact of various factors on the complexity, we use sets of randomly generated machines. In Figure 1 we can observe the results for machines generated with inputs set $|I| = \{3,4,5...10\}$, outputs $|O| = 7$ and number of states $n = 10$. We can note that *Suffix1by1* clearly outperforms the initial method of processing counterexamples.

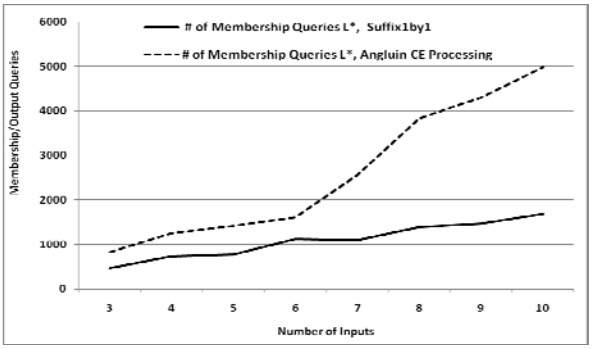


Figure 1 $|I| = \{3,4,5...10\}$, $|O| = 7$, $n = 10$

5 Conclusion

We investigated the inference of unknown models as DFAs with the new method of processing counterexamples. New technique to process the counterexamples has a gain over the existing techniques and it reduces significantly the effect of non minimal counterexamples, which is confirmed by the considered examples. Possible extensions of proposed work involve experimenting with more comprehensive set of examples and adaptation of this technique for learning unknown models as non-deterministic and parameterised machines.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. Information and computation, 2:87-106, 1987.

[2] Rivest, R.L., Shapire, R.E.: Inference of finite automata using homing sequences. In: Machine Learning: From Theory to Applications, pp. 51-73 (1993).

[3] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2):316–326, 1995.

[4] M. Shahbaz: Reverse Engineering Enhanced State Models of Black Box Components to Support Integration Testing. Ph.D. Thesis, Grenoble Institute of Technology (2008).

[5] M. N. Irfan, C. Oriat, R. Groz: Angluin Style Finite State Machine Inference with Non-optimal Counterexamples, Workshop on Model Inference In Testing 2010, ISSTA, Trento, Italy, 11-19 (2010)

Some ideas on active learning using membership queries*

Pedro García, Manuel Vázquez de Parga and Damián López
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{pgarcia,mvazquez,dlopez}@dsic.upv.es

1 Inference of teams of automata

Three out of four of the algorithms used by our group are based on the inference of a team of automata [1]. Each automata in the team is obtained using a generalized Blue-Fringe inference scheme [2]. Briefly, the generalization consist of: first, a random selection of a *blue* state; second, the random traversal of the *red* states set; if a *red*-mergible state is found, the *blue* state is deterministically merged, otherwise, the *blue* state is promoted to the *red* set; in both cases the *blue* set is updated. This process ends when the *blue* set is empty.

Note that the process is not deterministic and that each run of this scheme may return different automata. Thus, after n iterations of this process we may obtain a set of n automata. This team can be used in several ways to classify test samples. The one we used consist of the processing of the test samples by the automata in the team. If the automata accepts (rejects) the sample, it gives a positive (negative) vote inversely proportional to the square of its size.

2 Maximum discordance criterion

The first algorithm designed by our group uses the above described team inference process.

Briefly speaking, the initial training set is obtained using a small set of the queries available. A team of automata is inferred using this training set. The team is used to classify a set of random generated strings of a certain length. Those strings with higher discordance are selected to query

*Work partially supported by Spanish Ministerio de Educación y Ciencia under project TIN2007-60769

the oracle and added to the training set. This process is iterated while there are queries available.

The discordance value of a string is function of the distance among the absolute value of the vote obtained by the string and the absolute value of the maximum vote that any string can obtain.

In each iteration, in order to select each new set of p queries, a bigger pool of np strings of length k are randomly chosen. If the number of discordant strings in that pool is lower than p, then, the value of k is incremented in one unit, and a new pool of np strings of length is randomly generated. This process is repeated while the number of discordant strings is lower than p. The next iteration considers the final value of k of the previous iteration decremented in one.

3 Canonical generation of L^* experiments

The L^* algorithm by Angluin [3] is the core of the other algorithms our group proposes.

We recall that the algorithm by Angluin uses equivalence queries. This queries guide the experiments to include in the evidence table. The lack of such queries is substituted by the canonical generation of the experiments to include.

Two different traversals (depth-first and breath-first) can be followed to fill in the table. The behaviour of both approaches is quite different, and therefore, both were considered as different algorithms.

Once the limit of queries has been reached, two options were considered: the construction of an automaton from the information on the table; and, the inference of a team of automata using the above described method and the strings queried to the oracle.

The team of automata is inferred using the prefix-tree acceptor of the strings queried to the oracle, where those states detected during the construction of the table are marked as *red* states.

References

[1] P. García, M. Vázquez de Parga, D. López, and J. Ruiz. Learning automata teams. *LNAI*, 2010. 10th International Colloquium, ICGI-10.

[2] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. *LNAI*, 1433:1–12, 1998. 4th International Colloquium, ICGI-98.

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

Next question, please. Two query learning algorithms participate in the Zulu challenge

Juan Miguel Vilar

Departamento de Lenguajes y Sistemas Informáticos
Universitat Jaume I
12071 Castellón (SPAIN)
jvilar@lsi.uji.es

Abstract. We describe the two algorithms that participated in the recent Zulu competition. This is a competition about automata learning from membership queries. Our algorithms bootstrap the Blue-Fringe algorithm that learns finite state automata from given data. Unfortunately, they did not perform well in the competition.

1 Introduction

Under the query learning model, a learner can make queries to an oracle whose answers are used to infer a hypothesis about the object to be learned. We comment here two algorithms that participated in the Zulu competition [2], which aimed at testing learning algorithms based only on membership queries. Our algorithms bootstrap the well known Blue-Fringe algorithm [3].

We describe the competition first, then the algorithms and their performance in the competition. Finally, we outline some conclusions.

2 The Zulu Competition

The task of this competition was to learn regular languages using membership queries, i.e. the learners had access to an oracle that given a string answered true if the string belonged to the target language and false otherwise. The learners had a limited number of queries, after which they had to classify a set of test strings. The oracle was implemented via a web server that received queries and answered them using a simple protocol.

Eighteen different tasks were originally defined: two for each of the nine possibilities combining small, medium and large automata sizes with small, medium and large alphabets. For each task, an automaton was randomly generated. The number of queries was defined by running a modified version of L^*, the algorithm from [1]. The number of queries that the algorithm needed to reach an error rate below 40% was allocated to the task.

In six additional tasks of the (medium alphabet, medium size) category, only 25%, 50% and 75% of the queries made by L^*were allocated to the automata.

The performance of $L^*$was considered the baseline for comparing the algorithms.

3 The Algorithms

Our two algorithms follow this schema:

1. Initial queries are performed for the empty string and each of the strings consisting in just one symbol of the alphabet.
2. The already made queries are used as a corpus for inferring an automaton using an algorithm for automata learning from given data.
3. Statistics of the usage of the arcs of the automaton inferred in step 2 are used to decide next query.

Steps 2 and 3 are repeated until no more queries are available. The two algorithms differ in how they produce the next query.

3.1 Automaton inference

Let $\mathcal{C}$ be a set $\{(x_i, b_i)\}_{i=1}^{n}$ where each x_i is a string belonging to Σ^* and each b_i is a Boolean value that is true if x_i belongs to the target language and false otherwise. We can use $\mathcal{C}$ to build an automaton consistent with it by using an algorithm for automata learning from given data. These algorithms find an automaton consistent with a set of samples and, if the set is large enough, that automaton corresponds to the language. We have used the Blue-Fringe algorithm, described in [3]. This is one of a family of algorithms that build an automaton by merging the states of a tree-like representation of the data. The merging order is then critical. In the case of the Blue-Fringe algorithm, this order is based on the information about each state, which in turn is related to how many common tails the strings departing from each state have.

The automaton was completed with a sink state which was final or not according to a majority vote from $\mathcal{C}$.

3.2 Selection of the new query

We have decided that our corpus will be prefix-complete, i.e. if $x \in \mathcal{C}$[1] then $x' \in \mathcal{C}$ for every x' that is a prefix of x. This implies that the new query will be formed by appending one letter of the alphabet to one of the strings in the corpus, i.e. the set of *candidates* will be $\{xa \mid x \in \mathcal{C}, a \in \Sigma, xa \notin \mathcal{C}\}$.

We need a criterion for selecting one candidate. As commented above an important information for Blue-Fringe is the number of tails that depart from a state. When looking for a new query, we will try to increase this number. The idea is first to obtain usage statistics of each arc of the automaton and then use those statistics to score the candidates. To this end, we will concentrate in what we call "last arcs". These are the arcs used in parsing the last symbol of a string. So the first set of statistics we collect is which of the arcs of the automaton have been used a minimal number of times as last arc. We have followed this algorithm:

[1] Abusing notation we will use $x \in \mathcal{C}$ as a shorthand for $x \in \{y \mid \exists b : (y, b) \in \mathcal{C}\}$.

1. For each corpus string, parse it and increment in one the count of the arc corresponding to the last symbol of the string.
2. Assign a score of one to those arcs that have been used the minimum number of times and zero to the other arcs.

Now, when selecting a string from the candidates, we use two criteria giving rise to the two different algorithms that we presented to Zulu:

- QLLA (query learning from last arcs): prefer those candidates for which their last arc has a score of one.
- QLSLA (query learning form sum of last arcs): prefer those candidates that maximize the number of arcs with score one, not counting repetitions, using in parsing them.

In either case, there will usually be more than one selected candidate. We complement those criteria with two additional conditions:

- Compute usage statistics for the arcs of the automaton while parsing the strings. Prefer the strings with lower sum of usages.
- If still there are ties, prefer the first string in shortlex order (shortest string first; for strings of equal length, use lexicographical order).

3.3 Implementation issues

We implemented the algorithm in Haskell. Unfortunately, the speed of the Blue-Fringe algorithm was too slow so we resorted to two compromises. The first one was that instead of inferring an automaton after each query, we only inferred an automaton after failed queries. That is, if the result of a query agreed with the prediction of the automaton, the next query was computed using that same automaton. We suspect that the effect of this decision is small.

A more profound modification was due to the long computation times for the larger tasks. As the end of the computation approached, it was clear that it was not possible to end the execution on time. The solution was to generate the queries until the limit by enumerating the strings over Σ in shortlex order and skipping those already in the corpus. This was needed for both algorithms in tasks 17 and 18 (large automata, large alphabet) and for QLSLA in task 16 (medium automata, large alphabet).

4 Results of the competition

The algorithms were scored by comparing the percentage of correctly classified test strings with the corresponding percentage for the baseline algorithm. The results over the original tasks are presented in Table 1. In each cell, there are three numbers: the smaller ones are the ratios for the two tasks in the category and the third is their average. Values smaller than one mean that the success rate of the proposed algorithm is worse than the baseline. In that sense, only QLLA in the middle tasks performed adequately.

Table 1. Results of the algorithms in the original tasks

Alphabet size	Algorithm	Automata size		
		Small	Medium	Large
Small	QLLA	$0.905_{(1.000, 0.809)}$	$0.977_{(0.993, 0.961)}$	$0.924_{(1.057, 0.790)}$
	QLSLA	$0.835_{(1.000, 0.670)}$	$0.701_{(0.642, 0.760)}$	$0.711_{(0.706, 0.716)}$
Medium	QLLA	$0.892_{(0.891, 0.892)}$	$1.043_{(1.029, 1.057)}$	$0.924_{(0.934, 0.915)}$
	QLSLA	$0.904_{(0.930, 0.878)}$	$0.857_{(0.753, 0.961)}$	$0.698_{(0.620, 0.777)}$
Large	QLLA	$0.882_{(0.834, 0.930)}$	$0.836_{(0.837, 0.835)}$	$0.785_{(0.792, 0.778)}$
	QLSLA	$0.727_{(0.684, 0.769)}$	$0.884_{(0.881, 0.888)}$	$0.770_{(0.803, 0.737)}$

Table 2. Results of the algorithms in the restricted data tasks

Algorithm	Reduction percentage		
	25%	50%	75%
QLLA	$0.939_{(0.879, 0.999)}$	$0.762_{(0.703, 0.820)}$	$0.744_{(0.660, 0.828)}$
QLSLA	$0.768_{(0.683, 0.853)}$	$0.676_{(0.638, 0.713)}$	$0.661_{(0.611, 0.711)}$

It is more difficult to extract conclusions in the case of the restricted data tasks (Table 2) because obviously a worsening of the values is expected.

In any case, it is clear that QLLA outperforms QLSLA. In the global competition, QLLA ranked 18 and QLSLA ranked 20 of a total of 23 algorithms.

5 Conclusions

We have presented two algorithms for learning automata using membership queries. These algorithms bootstrap on an algorithm for learning from given data. The overall performance was not good. In restrospect, the criteria used for finding the next query should have used also information from the heads of the previous queries instead of relying on the tails only.

Acknowledgments. Work partially supported by the Spanish *Ministerio de Ciencia e Innovación* (*Consolider Ingenio 2010* CSD2007-00018) and *Fundació Caixa Castelló-Bancaixa* (P1·1B2006-31).

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(1), 87–106 (Oct 1987)
2. Combe, D., de la Higuera, C., Janodet, J.C., Ponge, M.: Zulu competition. http://labh-curien.univ-st-etienne.fr/zulu/index.php
3. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In: Proceedings of ICGI. pp. 1–12 (1998)

Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries

Borja Balle

Laboratori d'Algorísmia Relacional, Complexitat i Aprenentatge
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona
bballe@lsi.upc.edu

Abstract. Two algorithms for learning DFA with membership queries are described. Both of them are based on Kearns and Vazirani's version of Angluin's L^*. Our algorithms tied in the third place in the Zulu competition.

1 Introduction

This short note descibes two algorithms we used to participate in the Zulu competition. This was a competition for algorithms that learn DFA using only membership queries, a problem which has many practical applications (see [3]).

Algorithms entering the competition were given several tasks to solve. Those tasks consisted on learning randomly generated DFA with different number of states n and alphabet sizes $|\Sigma|$. There was a limit in the number of queries a learning algorithm could make for each task. These limits depended on the performance in each particular task of a baseline algorithm provided for the competition: an implementation of Angluin's L^* algorithm [1] using only membership queries. Since the competition's focus was on approximating the target automata, the quality of hypotheses was measured by means of a test set generated from a (mildly) target-dependent distribution. The predicted labels were used to evaluate the hypothesis accuracy.

All notation and definitions used are standard in the literature.

2 Algorithms

In this section we describe two algorithms we entered in the competition — named Balle1 and Balle3 there, L_1 and L_2 here — which tied in the third place. Both algorithms are based on the version of L^* introduced by Kearns and Vazirani in [4], which we will denote by $\mathsf{L}^*_{\mathsf{kv}}$ to avoid confusion. Our two algorithms differ only in the method used to simulate the equivalence queries in $\mathsf{L}^*_{\mathsf{kv}}$ by means of membership queries. The actual implementation was done with Matlab.

Although the differences between L^* and $\mathsf{L}^*_{\mathsf{kv}}$ are subtle enough to be considered implementation issues (cf. [2]), they matter a lot in practical applications. It is known that, given access to membership and equivalence queries, $\mathsf{L}^*_{\mathsf{kv}}$ can be implemented using a number of membership queries that, at least for acyclic DFA, is optimal up to constant factors in the worst case — note this is also the case for the version given by Rivest and Schapire [5]. Furthermore, $\mathsf{L}^*_{\mathsf{kv}}$ presents a feature that is unique among its relatives: the new hypothesis obtained by processing a counterexample s does not necessarily classify s correctly (cf. [4]). This is a valuable property for an algorithm in the Zulu setting, where finding new counterexamples may require spending a large number of membership queries.

Internally, $\mathsf{L}^*_{\mathsf{kv}}$ keeps a data structure called *discrimination tree* that contains information about the states of its current hypothesis, which correspond to equivalence classes of states in the target DFA. Each leaf on the tree corresponds to a different state in the current hypothesis and is identified by a string, the *access string* of that state. A discrimination tree together with a DFA (usually the target to be learned) define a partition of Σ^* in as many sets as leafs in the tree. The process

by which a string s is assigned to a leaf is called *sifting* and goes as follows: starting at the root node, recursively follow the right branch if the DFA accepts the concatenation of s with the string labeling the current internal node of the tree, and follow the left branch otherwise; repeat until a leaf is reached. When constructing a hypothesis from the tree, the transition labeled by σ leaving a state s is directed towards the leaf $\tau(s, \sigma)$ obtained by sifting the word $s\sigma$ down the tree. An example of a discrimination tree plus an hypothesis obtained from it is shown in Fig. 1.

As suggested in [2], our implementation of $\mathsf{L}^*_{\mathsf{kv}}$ uses a caching strategy to avoid repeated queries to the oracle. The procedure for asking membership queries is implemented as a proxy that keeps a dictionary with all the answers already obtained from the oracle. If the algorithm asks a query whose answer is cached in the dictionary, that answer is returned. Otherwise, a new query to the oracle is made and the answer is stored in the dictionary.

Equivalence queries in the original $\mathsf{L}^*_{\mathsf{kv}}$ are replaced in our implementation by a two-layer simulation using membership queries. The first layer is in charge of confronting the current hypothesis with all the answers cached in the dictionary. This takes advantage of the particular feature of $\mathsf{L}^*_{\mathsf{kv}}$ mentioned above. If a counterexample is found in the dictionary, it is returned. Otherwise, the algorithm enters the second layer, detailed in Fig. 2. It is in this second layer where the difference between our two algorithms lies, in particular in the function `Sample`. This function receives as input a length (drawn at random in our case) and outputs a word of that length sampled from a certain distribution — the uniform distribution over Σ^l in the case of L_1. Note that this is the same distribution used in the baseline provided for the Zulu competition.

In the case of L_2 the distribution uses some information obtained from the current hypothesis and discrimination tree, which is based on the following observation: the number of membership queries used for identifying the destination $\tau(s, \sigma)$ of a transition in the hypothesis is the number of steps needed to sift $s\sigma$; that is, the height of the corresponding target leaf in the tree. Based on this observation one can make the heuristic guess that transitions ending in shorter leafs — closer to the root — are less 'informed' than transitions going to taller leafs, and thus more likely to be wrong. Accordingly, in L_2 the distribution used by `Sample` is obtained from a random walk of length l over the hypothesis, with the probability of each transition depending on its destination's height. Under this distribution, strings traversing more transitions towards shorter leafs are more probable. Hopefully, these strings are more likely to be counterexamples to the current hypothesis. In our implementation, a weight is assigned to each transition using the expression

$$w(s, \sigma) = \left(\frac{1}{h_{\tau(s,\sigma)} - h_{\min} + 1} \right)^2 , \tag{1}$$

where $h_{\tau(s,\sigma)}$ is the height of the leaf corresponding to the state $\tau(s, \sigma)$ and $h_{\min}$ is the height of the shortest leaf in the discrimination tree. Transition probabilities are obtained by normalizing these weights for each state: $p(s, \sigma) = w(s, \sigma)/W_s$ where $W_s = \sum_\sigma w(s, \sigma)$. The example hypothesis in Fig. 1 has its transition probabilities computed according to this rule.

3 Discussion

The fact that L_1 performs much better than the baseline — note both of them use the same sampling strategy to search for counterexamples — indicates that implementation issues matter a lot in practical applications.

Even though L_2 seems more principled than L_1, both algorithms performed similarly in the Zulu competition. After the competition, further experiments were conducted on these algorithms, but, surprisingly enough, no statistically significant difference was observed between both algorithms. At the present moment we have no reasonable explanation for this phenomenon.

Acknowledgements. This work is partially supported by: the Generalitat de Catalunya 2009-SGR-1428 (LARCA), the EU PASCAL2 Network of Excellence (FP7-ICT-216886), and an FPU fellowship (AP2008-02064) from the Spanish Ministry of Education.

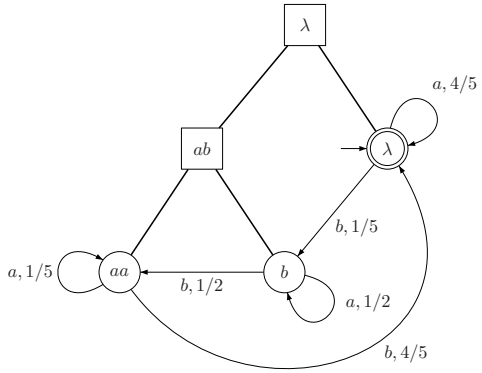


Fig. 1. Discrimination tree with superimposed hypothesis

Input: current hypothesis c and maximum sample size k
Output: a counterexample s or **equal**

for $i \leftarrow 1$ **to** k **do**
  // $U \sim \text{Unif}([0,1])$
  $l \leftarrow \sqrt{U \cdot (\texttt{Depth}(c)+6)^2 + 1}$;
  $s \leftarrow \texttt{Sample}(l)$;
  if $c(s) \neq \texttt{MQ}(s)$ **then return** s;
end
return equal;

Fig. 2. Simulation of an equivalence query with membership queries

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
2. Balcázar, J., Dıaz, J., Gavalda, R.: Algorithms for learning finite automata from queries: A unified view. Advances in Algorithms, Languages, and Complexity pp. 53–72 (1997)
3. Combe, D., Colin, H., Janodet, J.: Zulu: an Interactive Learning Competition (2009)
4. Kearns, M., Vazirani, U.: An introduction to computational learning theory. The MIT Press (1994)
5. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Machine Learning: From Theory to Applications pp. 51–73 (1993)

Finding Counterexamples Fast
Lessons learned in the ZULU challenge

Falk Howar, Bernhard Steffen, Maik Merten

University of Dortmund, Chair of Programming Systems,
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
{falk.howar, steffen, maik.merten}@cs.tu-dortmund.de
Tel. ++49-231-755-7759, Fax. ++49-231-755-5802

Introduction

The concept of query learning (due to Dana Angluin [1]) is used and developed in several different communities today. The TU Dortmund team originates in the area of program verification. Active learning in this community is used to produce exact and complete models of a system under test. Having in mind this application of learning, we entered the ZULU challenge with the goal to improve in (1) saving membership queries and (2) finding counterexamples, while still producing exact models. In both disciplines we proceeded in two steps: first finding good generic solutions and then customizing these to reflect the specific profile of the ZULU problems. Our main contribution is a highly configurable learning algorithm, which can mimic most of the known algorithms, and a new approach to steering the search for counterexamples using a monotone growing hypothesis annotated with coverage information.

A Configurable Inference Framework

The learning algorithm we used can best be described as an implementation of generalized *Observation Packs* [2]. It combines a discrimination tree [3] with a reduced observation table [4]. The realization as a general framework allows us to switch between different strategies for handling counterexamples easily as well as using a non-uniform observation table (i.e., a table with multiple sets of distinguishing suffixes). Additionally, these sets can be initialized arbitrarily (e.g., as $\{\varepsilon\}$ for deterministic finite automata (DFA) or as Σ for Mealy machines).

For ZULU, we configured two versions of our learning algorithm, both using a strategy for analyzing counterexamples that is based on [4]. The strategy extracts from each counterexample (1) a new distinguishing suffix and (2) the word from the *SA*-set that will produce an unclosure subsequently. The registered algorithms differed as follows.

Initial set of distinguishing suffixes: In one configuration, the initial set of distinguishing suffixes was initialized as $\{\varepsilon\}$ (as in the literature). In the other configuration, we used $\{\varepsilon\} \cup \Sigma$ in order to simulate the effect of changing from DFA to Mealy models.

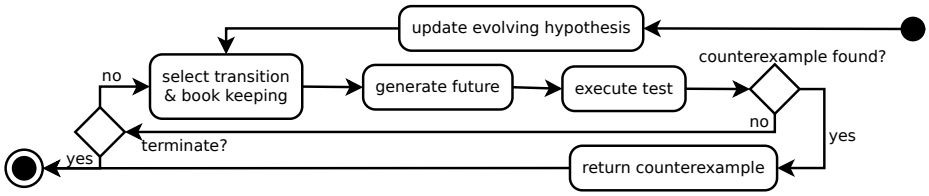


Figure 1. Continuous Equivalence Query

Observation Table: We used uniform and non-uniform observation tables. In a non-uniform table, the sets of distinguishing suffixes are managed independently for each component (cf. [2]). This leads to using less membership queries but more equivalence queries than a uniform table does.

Continuous Equivalence Queries

Active learning algorithms proceed in rounds. Each round is opened by an *equivalence query*. A counterexample returned from such a query will be analyzed and exploited during the rest of the round. This general formulation suggests that equivalence queries in different rounds are independent, i.e., run on unrelated conjectures. But, using a reduced observation table (or more precisely: analyzing counterexamples as in [4]) will guarantee a monotone growing hypothesis: the S-set of access sequences will only be extended by elements from the SA-set. The prefix-closed set S can be understood as a successively produced spanning tree of the target automaton. This observation has two consequences:

1. finding counterexamples coincides with proving transitions (elements from the set SA) leading to yet undiscovered states,
2. different conjectures can be related using the S-set.

Relating the different conjectures results in using only one *evolving* hypothesis automaton. In this automaton, only the transitions that correspond to elements from the SA-set can change (and only in a monotone fashion). This allows the formulation of continuous equivalence tests: each transition in the hypothesis can be annotated with a measure, e.g., counting the number of tests that have been executed for this transition; the counter will be reset when the transition changes. This global coverage criterion can be used to steer the search for counterexamples. Fig. 1 shows schematically one round of such a continuous equivalence query: after updating the hypothesis, single transitions are selected and then tested with some future (i.e., suffix word) until either a counterexample is found or a termination criterion is met. For the ZULU challenge, we concretized this general scheme as follows.

Select transition & Book keeping: For the *E.H.Blocking* algorithm, transitions from the SA-set were chosen randomly. Once used, a transition was excluded from subsequent tests. When no transitions were left to choose from,

Table 1. Algorithms: Configuration and Ranking

Algorithm	Dist. Set		Equivalence Query		Training	Rank
	Init.	Uniform	Continuous	Strategy	(Avg.)	
E.H.Blocking			yes	block transitions	89.38	1
E.H.Weighted	{ε}	no	yes	weight transitions	89.26	2
Random			no	random walks	88.93	6
run_random			no	random walks	80.17	14
run_blocking1	{ε}∪Σ	yes	yes	block transitions	79.89	15
run_weighted1			yes	weight transitions	79.65	16

all transitions were re-enabled. The *E.H. Weighted* algorithm uses weights on all transitions, which will be increased each time a transition is selected. The probability of choosing a transition is inversely proportional to the weight.

Generate future: The suffixes were generated randomly with increasing length. The length was initialized as some ratio of the number of states in the hypothesis automaton, and was increased after a certain number of unsuccessful tests. The exact adjustment of the suffix length was developed in a trial-and-error way to fit the properties of the problems in the ZULU challenge.

We did not use an explicit termination criterion. A query terminated as soon as the number of queries granted by ZULU was reached.

Results

For the actual competition, we registered six candidate algorithms: three using a non-uniform observation table with a DFA-style initial set of distinguishing suffixes and three using a uniform observation table with a (modified) Mealy-style initial set of distinguishing suffixes. Those two groups correspond to the decisions discussed above.

Both groups were equipped with the same three equivalence algorithms: (1) E.H.Blocking, (2) E.H.Weighted, and (3) a plain random walks algorithm as reference. The random walks algorithm tested randomly generated words. Table 1 shows the configuration of the algorithms, their average scores during the training phase and the rankings from the competition phase.

On the training problems, we also ran algorithms for the other two possible configurations of the learning phase. There was, as can partly be seen in Table 1 and Table 2, a significant gap between algorithms using uniform tables and algorithms using non-uniform tables (about 10 to 15 points). For the algorithms using the same kind of table, there were gaps of 2 to 5 points between the versions with different initial sets of distinguishing suffixes; DFA-style initial sets leading to better results.

Compared to the big differences in the scores that were caused by the configuration of the learning phase, the differences between the three equivalence

Table 2. Detailed Training Example: Problem 49763507

Algorithm	New Membership Queries			Rounds	States	Score
	Close Obs.	Analyze CEs	Search CEs			
E.H.Blocking	6,744	358	999	259	352	94.11
E.H.Weighted	6,717	349	1,035	262	351	94.61
Random	6,586	519	996	228	332	93.28
run_random	8,080	14	7	5	312	74.89
run_blocking1	8,074	11	16	6	319	73.06
run_weighted1	8,077	9	15	6	319	74.39

algorithms at first glance seem to be not significant. But, this is due to the ZULU rating mechanism as shown in Table 2 for the training problem 49763507. While the ratings do not differ significantly, the number of states does. The continuous equivalence algorithms produce conjectures with significantly more states using the same amount of queries (over all) as the random walks algorithm does.

Conclusion

We played the ZULU challenge following the same pattern for the learning phase and for the equivalence phase. We first built good general frameworks and then customized these to reflect the special requirements of the ZULU scenario: The problems were best learned as DFA and using a non-uniform observation table. Both decisions aim at keeping the table as small as possible, i.e., saving membership queries. Counterexamples were found considerably faster (especially for bigger systems) using an evolving hypothesis. This allowed spending more queries on the learning part. Also, due to their structure, the analysis of the counterexamples from the continuous equivalence queries required less membership queries than analyzing randomly generated counterexamples.

The configuration we used for the learning phase made us compete at eye level with the other players. Finally, the solution for finding counterexamples fast resulted in the small but deciding advance (89.39 vs. 88.93 in the training phase, but 1st vs. 6th in the competition phase).

References

1. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
2. José L. Balcázar, Josep Díaz, and Ricard Gavaldà. Algorithms for Learning Finite Automata from Queries: A Unified View. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.
3. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
4. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.