Insights to Angluin's Learning

Therese Berg, Bengt Jonsson, Martin Leucker and Mayank Saksena

Department of Computer Systems, Uppsala University, Sweden
{thereseb, bengt, leucker, mayanks}@docs.uu.se

Abstract. Among other domains, learning finite-state machines is important for obtaining a model of a system under development, so that powerful formal methods such as model checking can be applied.
A prominent algorithm for learning such devices was developed by Angluin. We have implemented this algorithm in a straightforward way to gain further insights to practical applicability. Furthermore, we have analyzed its performance on randomly generated as well as real-world examples. Our experiments focus on the impact of the alphabet size and the number of states on the needed number of membership queries. Additionally, we have implemented and analyzed an optimized version for learning prefix-closed regular languages. Memory consumption is one major obstacle when we attempted to learn large examples.
We see that prefix-closed languages are relatively hard to learn compared to arbitrary regular languages. The optimization, however, shows positive results.

1 Introduction

The last decades have witnessed significant advances in *model-based techniques* for specification, implementation, verification, and validation of reactive and usually distributed systems, e.g., in telecommunication, embedded control, and related application areas. The techniques include model checking [LPY97,GHP97], code generation [HLN+90] and model-based test generation [FJJV97,SEG00]. They all assume that a formal model of the system under study is available. Such formal models are assumed to be developed during the specification phase of system development, or *a posteriori* from an existing implementation.

One large obstacle to the adoption of model-based techniques is that in practice, quite often *no* formal specification is available or it is *outdated* due to the iteration process in the development of a system. Even if a formal specification is present that captures the latest version of the system intended to develop, it is not clear whether it corresponds to its actual realization.

One approach to overcome these limitations is to develop techniques for generating formal models with less manual effort and more automated support. In the extreme case, a formal model could be generated a posteriori, from the developed system. If no model of the system under development was present, this model can be used to analyze and validate the implementation. If a formal model was available a priori, the generated model can be compared with this one to show conformance of the implementation with respect to its specification.

For software systems with given source code, various static and dynamic analysis techniques have been developed, which can also be used to generate abstract models of a developed system [CDH+00,Hol00]. However, peripheral hardware systems, combined hard- and software systems, or third-party software systems do not allow means of static analysis. In practice, there is often no other way to analyze these systems than by looking at their traces, i.e., their sequences of input and output actions. Also, a program that analyzes the source code statically is heavily dependent on the particular implementation language used. A tool that analyzes externally observed traces is easier to adapt to a new program written in a new language.

In a seminal paper, Angluin [Ang87] described a method for learning finite-state automata, if it is possible to ask whether a string is a member of the language of the automata. This result implies that, in principle, finite-state automata can be learned for finite-state systems that have the following two characteristics:

- one can send sequences of actions to the system and
- the system signals whether it could execute the sequence.

This approach has been used in projects for test sequence generation by Steffen et al. [HHNS02], and by Peled et al. for developing techniques for conformance testing of finite automata [GPY02]. The number of reported efforts to use Angluin's algorithm (or some related algorithm) for generating finite automata models of reactive systems is still rather small and it is still not possible to make conclusions about the applicability of the techniques, how well it scales, or to pinpoint the crucial bottlenecks.

The objective of the research reported in this project is to investigate the efficiency of Angluin's algorithm for learning finite automata, and among them models of reactive systems, to investigate potential bottlenecks in applying it, and to investigate the effect of a rather straight-forward optimization for prefix-closed DFA. For this purpose, we have developed a naive implementation of Angluin's algorithm together with an optimization, which can optionally be invoked. We have applied this implementation to a series of synthetically generated systems, and to a set of rather simple models of reactive systems intended for verification by the Concurrency Workbench. From the results, we draw conclusions regarding the applicability and scalability of Angluin's algorithm, as well as the effect of our implemented optimization.

[HNS03] studies domain-specific optimizations to Angluin's learning algorithm including optimizations for prefix-closed languages. They have considered examples from telecommunication software but not studied the performance on synthetic examples. They have in this article used a slightly different model and therefore we could not easily compare our results. In [Ang01], Angluin revisits his algorithm and discusses several variants and their complexity. Practical results, however, are not mentioned.

In the next section, we recall basic definition of automata theory. In Section 3, we describe Angluin's learning algorithm as well as our optimization for prefix-closed languages. Our experiments are described and discussed in Section 4.

2 Preliminaries

For the following, we fix an *alphabet* Σ, i.e. a finite set of *letters*, usually denoted by $a, b, \ldots, a_1, a_2, \ldots$ A *language* is a subset of Σ^*, the set of finite (possibly empty) sequences of letters, also called *strings* or *words*.

A *deterministic finite-state automaton* (*DFA*) over Σ is a structure $\mathcal{A} = (Q, \delta, q_0, F)$ where Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\delta : Q \times \Sigma \to Q$ is the *transition function*. We denote the number of states Q, the size of the alphabet Σ, and the size of the transition function δ by respectively $|Q|$, $|\Sigma|$, and $|\delta|$. The latter is defined to be the number of elements of the domain of δ, i.e. $|Q \times \Sigma|$.

A *run* of $\mathcal{A}$ on a finite word $w = a_1 \ldots a_n \in \Sigma^*$ is a sequence $q_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_n$, where q_0 is the initial state of $\mathcal{A}$ and $q_{i+1} = \delta(q_i, a_{i+1})$ for $i \in \{0, \ldots, n-1\}$. It is called *accepting*, if $q_n \in F$. The *language* accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \text{ there is an accepting run of } \mathcal{A} \text{ on } w\}$. We call a language $\mathcal{L}$ *regular* if there is a DFA accepting $\mathcal{L}$.

Let us recall the notion of Nerode's right congruence. Given a language $\mathcal{L}$, we say that two words $u, v \in \Sigma^*$ are *equivalent*, written as $u \equiv_{\mathcal{L}} v$, if for all $w \in \Sigma^*$ we have $uw \in \mathcal{L}$ iff $vw \in \mathcal{L}$. It is easy to see that $\equiv_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ is a right congruence, i.e., it is an equivalence relation that additionally satisfies $u \equiv_{\mathcal{L}} v$ implies $uw \equiv_{\mathcal{L}} vw$ for all $w \in \Sigma^*$. We denote the equivalence class of a word w by $[w]$.

It is folklore that a language $\mathcal{L}$ is regular iff $\equiv_{\mathcal{L}}$ has finite *index*, i.e., the number of equivalence classes of Σ^* with respect to $\equiv_{\mathcal{L}}$ is finite. Let us recall the idea of the proof for the direction *right-to-left*: Given a language $\mathcal{L}$ with finite index, we construct an automaton $\mathcal{A}_{\mathcal{L}}$ such that $\mathcal{L}(\mathcal{A}_{\mathcal{L}}) = \mathcal{L}$. The states of $\mathcal{A}_{\mathcal{L}}$ are the equivalence classes of Σ^* with respect to $\equiv_{\mathcal{L}}$, the initial state is the equivalence class containing the empty string, denoted by ε, final states are the ones containing strings in $\mathcal{L}$, and the transition function maps $([w], a)$ to $[wa]$.

It can be shown that this construction yields a minimal DFA accepting $\mathcal{L}$, i.e., the number of states is minimal among all DFA accepting $\mathcal{L}$. Furthermore, it can be shown that every minimal DFA is isomorphic to the one we constructed.

3 Learning finite state machines

3.1 Angluin's learning algorithm

We here try to give a succinct description of the main ideas behind Angluin's learning algorithm. We assume that a system in which we are interested can be modeled by a DFA $\mathcal{A}$. The problem can now be looked upon as identifying the regular language which is accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$.

In a learning algorithm a so called *Learner*, who initially knows nothing about $\mathcal{A}$, is trying to learn $\mathcal{L}(\mathcal{A})$ by asking queries to a *Teacher* and an *Oracle*. There are two kinds of queries.

- A *membership query* consists in asking the *Teacher* whether a string $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{A})$.

– An *equivalence query* consists in asking the *Oracle* whether a hypothesized DFA $\mathcal{M}$ is correct, i.e., whether $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$. The *Oracle* will answer *yes* if $\mathcal{M}$ is correct, or else supply a counterexample u, either in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{M})$ or in $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{A})$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA $\mathcal{M}$ using the obtained answers. When the *Learner* feels that she has built a "stable" hypothesis $\mathcal{M}$, she makes an equivalence query to find out whether $\mathcal{M}$ is correct. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to revise $\mathcal{M}$ and perform subsequent membership queries until arriving at a new hypothesized DFA, etc.

The information gained by the *Learner* can at any point during the learning process be represented as a partial mapping $\mathcal{O}$ from Σ^* to $\{accepted, rejected\}$. The domain $Dom(\mathcal{O})$ of $\mathcal{O}$ is the set of strings for which membership queries have been performed, or which the *Oracle* has given as counterexamples in equivalence queries.

Roughly speaking, a learning algorithm should prescribe how to transform a partial mapping $\mathcal{O}$ into an automaton. This can be done by fixing a subset S of $Dom(\mathcal{O})$, defining an equivalence relation $\simeq$ on S, and building the automaton as the set of equivalence classes of strings in S. Intuitively, two strings u and u' should be equivalent if there is reason to believe that $u \equiv_{\mathcal{L}(\mathcal{A})} u'$. Since the *Learner* can only obtain partial information about $\mathcal{A}$ from $\mathcal{O}$, one idea is to approximate $\equiv_{\mathcal{L}(\mathcal{A})}$ by an equivalence $\simeq$, which uses only information in $\mathcal{O}$. To be able to build an automaton on the basis of S and $\simeq$, the following two criteria should preferably be satisfied.

– *completeness:* If $u \in S$, and $a \in \Sigma$ then $ua \simeq u'$ for some $u' \in S$.
– *consistency:* If $u \simeq u'$ for $u, u' \in S$ and $a \in \Sigma$, then $ua \simeq u'a$ (i.e., $\simeq$ is a right congruence).

Completeness ensures that we can define transitions from each equivalence class for each letter in Σ; consistency ensures that such transitions have a unique target equivalence class. We note that in order to check completeness and consistency, it is necessary to define $\simeq$ on all strings u and ua such that $u \in S$ and $a \in \Sigma$. Whenever the current values of S and $\simeq$ satisfy the completeness and consistency criteria, the *Learner* can form the corresponding hypothesis $\mathcal{M}$ and make an equivalence query about $\mathcal{M}$.

Let us now describe Angluin's algorithm more specifically. Angluin's algorithm maintains a prefix-closed set S and a suffix-closed set E of strings, both of which are monotonically increased during the algorithm. Initially S and E contain the empty string ε. We define $\simeq$ as follows: $u \simeq v$ if for all $w \in E$ we have $uw \in \mathcal{L}(\mathcal{A})$ iff $vw \in \mathcal{L}(\mathcal{A})$.

From a complete and consistent $\mathcal{O}$, a hypothesis $\mathcal{M}$ is formed as the automaton, whose states are equivalence classes of strings in S. If $\mathcal{O}$ is not complete, then S is increased with strings that represent missing equivalence classes. If $\mathcal{O}$

is not consistent, E is increased with a suffix which replaces the inconsistent equivalence class with two new classes.

The description of Angluin's algorithm in [Ang87] represents $\mathcal{O}$ by an *observation table* $\mathcal{OT}$. The observation table is a table with rows corresponding to strings in S and columns corresponding to strings in E. The algorithm gradually fills the entry (u, v) for row u and column v by *accepted* or *rejected*, after receiving a reply for a membership query for uv.

Of course, some membership queries can be saved by entering also the counterexamples returned in negative equivalence queries as accepted or rejected. We note that the observation table is redundant in that the result of a membership query for u occurs in all entries (v, w) such that $u = vw$. Thus, we do not need to make one membership query for each such entry, but we can simultaneously fill all such entries.

Angluin's algorithm is designed to construct minimal DFA for the guessed language.

3.2 Prefix-closed models

In many applications, we want to learn an automaton $\mathcal{A}$, which is a model of a reactive system. Often, reactive systems can be modelled as transition systems. These can be understood as (non-deterministic) finite-state automata (with partial transition relations) in which every state is an accepting state. Thus, the language defined by such an automaton will be prefix-closed. In this section, we discuss how to exploit this fact for optimizing the learning process.

A language $\mathcal{L}$ is *prefix-closed* if for every w in $\mathcal{L}$, all prefixes of w are in $\mathcal{L}$. A DFA is *prefix-closed* if its language is prefix-closed. It follows that a minimal prefix-closed DFA has only one non-final state, the so-called *sink*, with transitions only to itself. Note that Angluin's algoritm learns minimal DFA.

Studying strings that are possibly accepted by prefix-closed DFA, we make the following simple but important observations:

1. Prefixes of accepted strings are accepted.
2. Suffixes of rejected strings are rejected.

We can use these characteristics of prefix-closed DFA to reduce the needed number of membership queries as follows. Before querying a string, we first test it for (2), that is whether it is an extension of a string already observed to be rejected. If so, we can add the result immediately to the observation table. Otherwise, we ask the teacher. Thus, we never need to query extensions of observed rejected strings.

Angluin's *Learner* starts with queries for short strings, and thereafter queries successively longer and longer strings. In general, the test for (1) will not be able to consult previous observations, so it is rarely applicable. There is, though, an exception when it could be useful, namely when performing queries for prefixes of received counterexamples. If the counterexample c is accepted, we know that all its prefixes are accepted, too. In the best case, applying (1) would save *me*

membership queries, where m is the maximum length of any received counterexample and e is the number of equivalence queries made. Knowing the best case bound and due to lack of evaluation time, we did not implement (1).

4 Experimental Results

The implementation We have implemented Angluin's learning algorithm, closely following the high-level description in [Ang87]. Furthermore, we have implemented our proposed optimization for prefix-closed models. It differs from the ordinary learner only in that it can infer the answer of some membership queries, due to properties of prefix-closed languages. Accordingly, the number of membership queries can be expected to be smaller, while the number of equivalence queries is unchanged. Furthermore, it requires the same amount of memory for the observation table.

We simulated the teacher (and oracle) on the same computer as the learner, for reasons of simplicity. In practice, a teacher will typically be realized as a process communicating with a slow external device.

Our learners are written in Java using the library AMoRE developed at RWTH Aachen for maintaining automata.

The experiments Our experiments aim at finding out how our implementation of Angluin's learner and our optimized learner perform and scale in practice. We have examined real-world examples and randomly generated examples. The real-world examples are several protocols shipped with the Edinburgh Concurrency Workbench. They were originally formulated in Milner's CCS. We transformed their transition system representation into prefix-closed DFA.

For reasons of comparison, we studied two kinds of random examples, prefix-closed random examples and arbitrary random DFA.

As pointed out before, the expected bottle-neck in practice for a learner is the number of membership and equivalence queries, since a communication with a typically slow external device is required and quite many queries are needed. Thus, we concentrated our experiments on the number of membership and equivalence queries. To get an overall picture, we also measured the execution time and memory consumption for large examples. Hereby "execution time" means the total execution time minus the time for equivalence queries. In other words, we measure the time spent by the learner plus the one spent by the teacher. Since there are several ways to realize The oracle, we disregard this time.

In our experiments, we vary the alphabet size and the number of states of the automata. Our measurements do not adhere to strong statistical requirements. Thus, they cannot be used to *prove* the practical performance of the algorithms in a statistical sense. However, they are good enough to show a tendency and to point out directions for future optimizations and analysis.

4.1 Angluin's algorithm

A theoretical upper bound for the number of membership queries is the worst-case size of the observation table. Angluin [Ang87] calculates this bound to $O(m|\Sigma||Q|^2)$, where m is the maximum length of any received counterexample. If the *Oracle* always provides a smallest counterexample, then $m = |Q|$, and thus the number of membership queries are in the worst case $O(|\Sigma||Q|^3)$.

To investigate how the algorithm behaves in practice, we studied it on arbitrary random examples as well as prefix-closed random examples. Let us start with the arbitrary ones.

Random examples

The samples We studied random examples varying the number of states and letters. We generated and learned DFA between 10 and 100 states, in steps of 10.

Each set of measurements was carried out with different alphabet size. For systems with up to 60 states, we studied from 5 up to 50 letters in steps of 5, and, for systems with more states, from 10 up to 50 letters in steps of 10.

We sampled 10 DFA for each state and letter combination, except for those with the number of states 70 or higher, for which we sampled only 5.

Experience Fixing the number of states but varying the number of letters, we observe a linear behavior, as expected. See Figure 1, in which the number of states are fixed to 10 and 60.

The collected data shows that, in terms of membership queries, Angluin's learning algorithm is approximately linear in states on random DFA, despite the algorithm's worst-case complexity. As an example, we show the number of membership queries relative to the number of states, with the number of letters fixed to 10 and 40, in Figure 2.

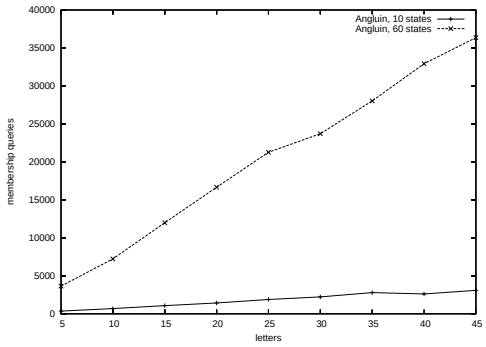


Fig. 1. Random automata, number of states fixed to 10 and 60.

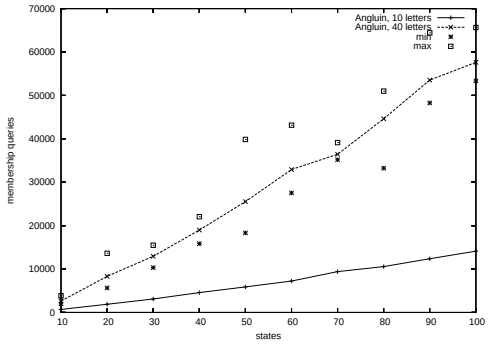


Fig. 2. Random automata, number of letters fixed to 10 and 40.

To get an impression of the performance of the algorithm, learning a random example of 100 states and 25 letters, took 1 hour, 40,000 membership and 15 equivalence queries, and 110 MB of space. This long execution time was one reason for learning fewer automata of larger sizes. The other reason is the huge memory consumption of the observation table.

Additionally, we studied the number of membership queries with respect to the number of transitions, $|\delta| = |Q||\Sigma|$. This is possible since we discovered by our measurements that the nominal variables states and letters behave interchangeably. Figure 3 shows the number of membership queries with respect to the number of transitions. We can describe the observations roughly by the relation $|membership\ queries| = k|\delta|$, where $k \approx 14$.

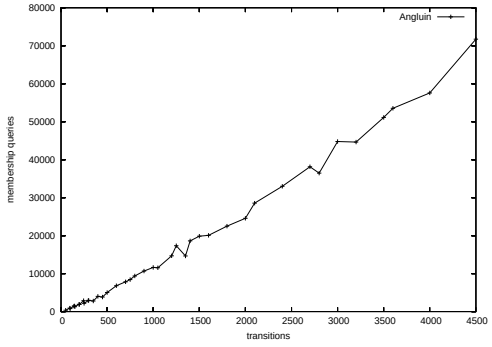


Fig. 3. Membership queries and transitions for random automaton.

Random prefix-closed Examples

The samples In general, prefix-closed DFA require more time and space to learn with Angluin's algorithm, so we studied fewer samples. We learned automata with 10 to 50 states in steps of 10 and varied the number of letters from 10 to 50 in steps of 10. We learned approximately 10 automata of each kind up to 30 states and fewer of larger ones.

Experiences As mentioned before, arbitrary random examples are in general easier to learn than prefix-closed random examples. An example for this is shown in Figure 4. Learning a particular random generated automaton, with 40 letters and 40 states, requires approximately 19,000 membership queries and a prefix-closed automaton of the same size requires 40,000 membership queries, that is about twice as many.

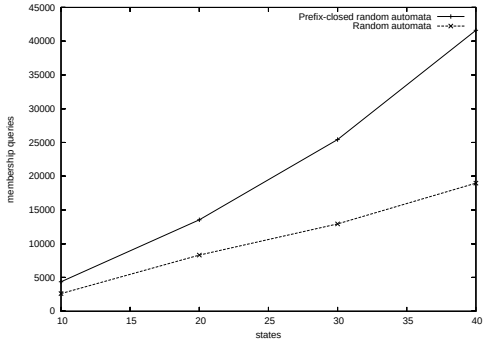


Fig. 4. Random prefix-closed automata and random automata, both with 40 letters.

Let us study the cause for this difference. Angluin's learning algorithm tries to learn an automaton by finding representatives of different Nerode's right congruence classes, as described in Section 3.1. To show that two strings u and v are not members of the same congruence class, it has to find one string w so that uw is accepted but vw not, or the other way around. In minimal prefix-closed DFA, as maintained by Angluin, every state except one is accepting, and it is more likely that states accept almost the same language. This makes it more difficult to find a distinguishing string w.

Furthermore, we see that the curves for learning prefix-closed languages grow steeper than for arbitrary random examples. On prefix-closed examples we come closer to the worst case complexity of Angluin's algorithm. Thus, prefix-closed examples are harder to learn than arbitrary ones. This result is slightly disappointing, since reactive systems can usually be modeled by prefix-closed automata. This experience is in contrast to the one gained in the area of model checking, where worst-case complexities usually do not show up in real-world examples.

A particular random example with 100 states and 25 letters took 11 hours, 110,000 membership queries, 29 equivalence queries and 160 MB of memory.

The top curve in Figure 5 shows membership queries versus transitions for random prefix-closed examples. It is no longer linear. We can give a very rough description of the observations by the quadratic relation $|\mathit{membership\ queries}| = k|\delta|^2$, where $k \approx 0.016$.

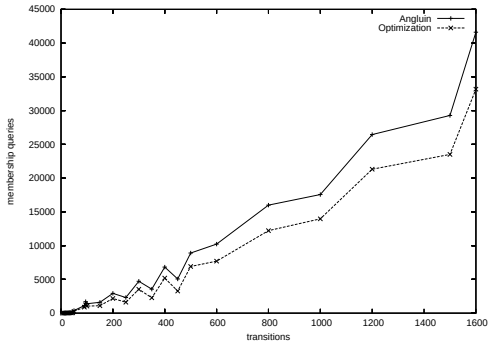


Fig. 5. Transitions for random prefix-closed automata. Angluin vs. with optimization.

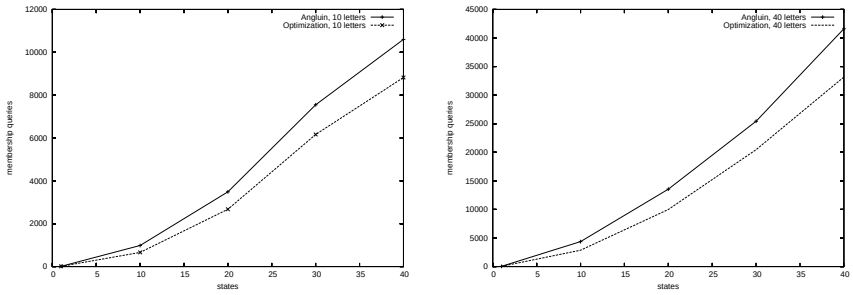
4.2 The optimization for prefix-closed systems

As pointed out in the previous section, the optimized version for prefix-closed languages takes into account that suffixes of rejected strings are rejected. Before issuing a membership query to the teacher, we check whether we can deduce it from previous membership queries. In our setting, the optimized learner gives about the same execution time as the ordinary learner. Since we simulate the *Teacher* on the same computer, a query takes roughly the same amount of time as a table lookup. Note that in the setting where a concrete hardware system is learned, the time for a table lookup might be negligible compared to the time a membership query needs.

Prefix-closed random examples

The samples On the optimized learner, we studied the same random prefix-closed examples as with the ordinary learner.

Experience We observe that the optimization yields noteworthy savings in terms of membership queries. To give an example, we have shown the number of membership queries with respect to the number of states in Figure 6(a) and Figure 6(b) for the number of letters fixed to 10 and 40, for the optimized version



(a) (b)

Fig. 6. Random prefix-closed examples learned with Angluin and optimization, number of letters fixed to 10 and 40.

in comparison with Angluin's version. We save in case of larger automata approximately 20% in both cases when using the optimization.

The particular example of size 100 states and 25 letters, from the previous subsection, took 12 hours, 96,000 membership queries, 29 equivalence queries and 160 MB of memory for our optimized learner.

Real-world examples

The samples We studied 6 transition systems of CCS processes. They are simple examples like buffers, vending machines, or several examples of schedulers and mutual exclusion protocols. Their number of states lie between 2 and 13 and the number of letters between 3 and 6.

We failed to learn some larger protocols, namely some instances of parameterized schedulers, the Jobshop (77 states, 7 letters) and an ATM protocol (1715 states, 27 letters). The reason is that we did not invest effort into a good algorithm for finding counterexamples; it took too long to find counterexamples for the protocols in question. (Note that the execution time which we measure is independent of the time spent for finding counterexamples.) The ATM, though, failed due to lack of memory.

Experience The number of membership and equivalence queries, as well as execution time, are shown in Table 1.

Comparing the number of membership queries of the optimized version with respect to Angluin's algorithm, we saved about 60%. Details can be found in Table 2.

To check whether real-world examples show a different behavior in learning by the optimized algorithm, we compared them with random prefix-closed examples of the same sizes. The results are shown in Table 3.

Protocol	states	letters	mq	eq	time (ms)	mq with opt	eq with opt	time with opt (ms)
Abp-lossy	3	3	22	2	65	9	2	1057
Buff3	9	3	202	5	2305	77	5	4907
Dekker-2	2	3	7	1	646	4	1	7
Peterson-2	2	3	7	1	352	4	1	288
Sched2	13	6	691	7	43031	115	7	48207
VMnew	11	4	513	7	26191	191	7	20091

Table 1. Learning real-world example.

Protocol	mq	mq with opt	saved mq (%)
Abp-lossy	22	9	59
Buff3	202	77	62
Dekker-2	7	4	43
Peterson-2	7	4	43
Sched2	691	115	83
VMnew	513	191	63

Table 2. Saved membership queries with optimization.

We see that the optimized learner is often better on the protocols relative to random examples (see Table 4). On average the real-world examples required 7% fewer membership queries without and 35% with the optimization. This might indicate that real-world examples exhibit a certain structure which makes the algorithm perform better.

The bottom curve in Figure 5 shows membership queries with the optimized learner versus transitions for random prefix-closed examples. A very rough summary of our observations is $|\textit{membership queries}| = k|\delta|^2$, with $k \approx 0.013$.

5 Conclusions and future work

Among the conclusions we draw from our experiences is the fact that random prefix-closed automata are harder to learn in comparison to completely randomly generated automata. For our random examples, the number of membership queries can roughly be described as linear in the number of transitions. Membership queries for our prefix-closed examples, in comparison, are approximately quadratic in transitions.

Moving deeper into the domain of prefix-closed automata we conclude that it is possible to reduce the number of membership queries by using an optimization specially shaped for these automata. The optimization reduces the number

states	letters	mq	eq	time (ms)	mq with Opt	eq with Opt (ms)	time with Opt (ms)
3	3	22	2	153	12	2	115
9	3	233	5	1443	173	5	1384
2	3	7	1	25	4	1	10
13	6	992	8	10885	737	8	10989
11	4	497	7	5230	341	7	5408

Table 3. Random prefix-closed automata.

protocol	m.q. quotient	with opt.
Abp-lossy	1	0.75
Buff3	0.87	0.45
Dekker-2	1	1
Peterson-2	1	1
Sched2	0.70	0.16
VMnew	1.03	0.56

Table 4. Random prefix-closed automata vs. real-world examples.

of membership queries considerably. For the randomly generated prefix-closed automata we measured a reduction of 20%.

Turning our attention to the real-world examples we see that the optimization works much better, saving 60% membership queries relative to unoptimized learning. We also compared the result of learning real-world examples with randomly generated prefix-closed examples of the same size, in order to investigate if they behaved in the same manner. The result reveals a better performance for the real-world examples in terms of membership queries, especially with the optimization. This seems to imply that our real-world examples have a more suited structure for learning. Hopefully this observation can be used to optimize the learning process further.

Memory consumption is a problem we experienced when learning large models. In order to learn these models, we need more memory efficient data structures. Further optimizations for prefix-closed DFA are possible. For instance, one can save space and time by using the fact that there is exactly one non-final state.

Consider the following idea which might improve learning. When some state representatives are "sufficiently equivalent" we can stop distinguishing them further. In observation table terminology, this would correspond to not querying any more column labels for row labels which are "sufficiently equivalent". Can such criteria improve the learning process and does it yield correct models for some classes of systems?

References

[Ang87] Dana Angluin. Learning regaular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang01] Dana Angluin. Queries revisited. In Naoki Abe, Roni Khardon, and Thomas Zeugmann, editors, *Algorithmic Learning Theory, 12th International Conference 2001*, volume 2225 of *Lecture Notes in Computer Science*. Springer, 2001.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Software Engineering*, June 2000.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.

[GHP97] Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, editors. *The Spin Verification System*, volume 32 of *DIMACS series*. American Mathematical Society, 1997. ISBN 0-8218-0680-7, 203p.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02,* 5^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M.B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403–414, April 1990.

[HNS03] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003. to appear.

[Hol00] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification: Proc.* 7^{th} *Int. SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147, Stanford, CA, 2000. Springer Verlag.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.

[SEG00] M. Schmitt, M. Ebner, and J. Grabowski. Test generation with autolink and testcomposer. In *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'2000*, June 2000.