Automata Learning with on-the-Fly Direct Hypothesis Construction*

Maik Merten[1], Falk Howar[1], Bernhard Steffen[1], and Tiziana Margaria[2]

[1] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{maik.merten|falk.howar|steffen}@cs.tu-dortmund.de
[2] University Potsdam, Chair for Service and Software Engineering, Potsdam, D-14482, Germany
margaria@cs.uni-potsdam.de

Abstract We present an active automata learning algorithm for Mealy state machines that directly constructs a state machine hypothesis according to observations, while other algorithms generate a state machine as output from information gathered in an observation table. Our DHC algorithm starts with a one-state hypothesis that it successively extends using a direct construction approach. This approach enables direct observation of the automata construction process: the learning algorithm continues to complete its hypothesis, providing intuition to a field of formal methods otherwise dominated by algorithms that largely operate on internal data structures without visible feedback.
The DHC algorithm is competitive in cases where memory is the critical issue, e.g., in embedded networked systems. It is also well-suited as educational tool to teach the underlying well-established theoretical methods in a totally unbiased fashion, without cluttering the view onto the actual idea of the learning process with aspects only relevant to internal bookkeeping.

1 Introduction

Most system documentation has central shortcomings that reduce its usefulness, sometimes to the point of being useless. Apart from missing desirable properties like being faithful to the documented system, complete, or comprehensive, it shows distinctive lack of behavioral formal models like, e.g., automata or other formats of finite state machines.

Many real-life systems can be modeled as input/output state machines, e.g., most networked applications react according to received messages and an internal state, producing state transitions and output messages in response to input messages. The same is true for hardware circuits and for many communication protocols. State machines thus are of interest for, e.g., documentation

* This work was partially supported by the European Union FET Project CONNECT: Emergent Connectors for Eternal Software Intensive Networked Systems (http://connect-forever.eu/).

purposes of networked systems and reactive systems in general, potentially giving insightful information on system behavior. Behavioral models provided as state machines can also be used for automated or semiautomated verification (e.g., via means of model-checking) and for simulation purposes, as the strict formalism enables automatic execution. Thus ways to (semi-)automatically derive state machine descriptions of actually deployed systems are a useful addition to the documentation-toolbox.

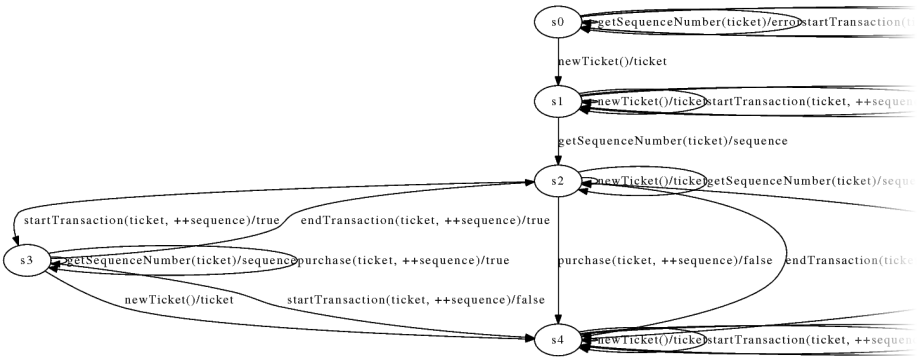

Figure 1. Model of an exemplary web service, learned from the WSDL via automata learning

Figure 1 shows a very small example of a model created via active automata learning of a web service that simulates a vending service. Messages defined in the WSDL description can be observed, modifying the state of the service. The model reveals a tight interlock between communication primitives: e.g., the "`purchase`" primitive can only be successfully executed once a transaction was opened using the "`startTransaction`" primitive. While a WSDL does not contain such information, this is potentially valuable knowledge that any documentation for this service should contain. Automata learning can reveal such properties with little or no prior knowledge on the inner workings of exploration targets.

In this paper, we present our approach of learning automata by direct hypothesis construction (DHC). The DHC algorithm works by successively expanding a spanning tree of already explored states in a breadth first manner, and introducing cycles whenever a newly found states is considered equivalent to an already explored state until a complete hypothesis automaton has been formed. Characteristic is its notion of equivalence, which is based on the next-step pattern called *signatures*. As signatures simply based on the input alphabet are of course not enough to characterize the states of a finite state system, they are successively extended to comprise artificial symbols consisting of whole sequences of input symbols in a way mimicking the rows of the observation table underlying the

well-known L^* algorithm. The result is an algorithm which continuously maintains a hypothesis and which has a significantly smaller memory footprint, an important property for our successful attempts to learn systems with more than a million states.

After providing the preliminaries in the next section, we present the DHC algorithm in Section 3, we discuss its profile in Section 4, and present our conclusions and perspectives in Section 5.

2 Scenario: Input/Output systems

Automata learning algorithms were originally designed to learn finite state acceptors (i.e. regular languages) [2]. The approach can be extended to learn Mealy machines in a straightforward manner: only the notion of language has to be replaced by the broader notion of a semantic functional [17]. In Mealy machines input words classify w.r.t. the output symbols produced, rather than w.r.t. acceptance.

In practice Mealy machines have been shown to be useful to describe large classes of reactive systems [15,13,14,1]. Given an input symbol, a machine will produce an output symbol depending on that input and its currently active internal state, and may also let the machine switch to a another active state. The formal definition for Mealy machines is as follows:

Definition 1 (Mealy Machine).
A Mealy machine is defined as a tuple $\langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states *of size* n*, i.e.:* $n = |Q|$,
- $q_0 \in Q$ *is the* initial state,
- Σ *is a finite* input alphabet *of size* k*, i.e.:* $k = |\Sigma|$,
- Ω *is a finite* output alphabet,
- $\delta : Q \times \Sigma \to Q$ *is the* transition function, *and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function.

Intuitively, a Mealy machine evolves through states $q \in Q$*, and whenever one applies an input symbol (or action)* $a \in \Sigma$*, the machine moves to a new state according to* $\delta(q,a)$ *and produces an output according to* $\lambda(q,a)$.
We write $q \xrightarrow{i/o} q'$ *to denote that on input symbol* i *the Mealy machine moves from state* q *to state* q' *producing output symbol* o*. The transition function* $\delta : Q \times \Sigma \to Q$ *and the output function* $\lambda : Q \times \Sigma \to \Omega$ *can be extended to process words in the obvious way.*

Learning algorithms for input/output systems use two types of queries to gather information on the target system:

- Membership Queries (MQs) retrieve behavioral information of the target system. Consisting of sequences of stimuli, MQs actively trigger the production of behavioral outputs which are collected and analyzed by the learning

algorithm. MQs are used to construct a hypothesis, which after being made consistent is subject of a verification by a second class of queries, the equivalence queries.
- Equivalence Queries (EQs) are used to determine if the learned hypothesis is a faithful representation of the target system. If the equivalence oracle handling the EQ finds diverging behavior between the learned hypothesis (output function λ^{hypo}, initial state q_0^{hypo}) and the target system (output function λ^{target}, initial state q_0^{target}), a counterexample $ex \in \Sigma^*$ with

$$\lambda^{hypo}(q_0^{hypo}, ex) \neq \lambda^{target}(q_0^{target}, ex)$$

is produced, which is used to refine the hypothesis after restarting the learning process.

With those two query types learning algorithms such as $L^*_{i/o}$ [12] create minimal automata models, this meaning that the learned result never contains more states than the minimized representation of the target system, and they also guarantee termination with an accurate model.

The minimality property results from identifying states by their potential for future output behavior, which according to the well-know Myhill/Nerode Theorem [11] directly relates these states to the finitely many states of the minimized representation of the target system [17]. More concretely, the states of the learned automata correspond to the equivalence classes of the following equivalence relation between states $q_a, q_b \in Q$:

$$q_a \equiv q_b \iff \forall w \in \Sigma^+,\ \lambda(q_a, w) = \lambda(q_b, w)$$

In practice it is impossible to algorithmically consider all such futures. On the other hand, it is possible to distinguish all states of a finite state systems with finitely many futures.

Equivalence queries provide the termination property: With every counterexample the learning algorithm identifies at least one new state by using the diverging output behavior. Given that the number of states to be discovered is bound by the number of states in the target system, termination is guaranteed and the equivalence queries guarantee termination with the correct result.

3 The DHC algorithm

Like all other active automata learning algorithms, the DHC algorithm also aims at constructing the minimal deterministic automaton of some given finite state target systems by identifying states according the to Myhill/Nerode theorem as described in the previous section. Characteristic to the DHC algorithm, however, is its consequent direct breadth-first oriented construction of intermediate hypotheses. This construction in particular avoids the use of the so-called observation tables, that may easily become a memory bottleneck.

As usual, our breadth-first search maintains a queue of states to be explored, which gets initialized with the initial state of the automaton. If a state is unique, i.e. not equivalent to any already explored state, its successors are enqueued for later exploration.

If we were able to decide the Myhill/Nerode equivalence described above this construction would immediately deliver the desired automaton. Unfortunately, in practice this equivalence can in general only be approximated on the basis of testing. The DHC algorithm therefore follows a very simple principle based on the following notion of signature, which works under the assumption that the set of input symbols is ordered:

Definition 2 (Signature). *The ordered set of output symbols produced at a state $q \in Q$ in response to applying all inputs of an ordered input alphabet is called* output signature*:* $sig(q) : (\lambda(q,a) \mid a \in \Sigma)$

Quite similarly to L^* [2] the DHC algorithm iterates now two steps:

Step 1 Construction of a (hypothesis) automaton based on the equivalence of states defined by equality of the corresponding signatures. This step always terminates with a closed and consistent hypothesis in the sense of L^*. In the following, we call states with identical signature *siblings*.

Step 2 Extension of the input alphabet in response to the counterexamples provided by the Equivalence Oracle. There are various variants of the DHC algorithms depending on the strategy of counterexample treatment. The most important ones are

- to add *all* suffixes of the counterexample to the input alphabet, which is strongly related to the counterexample treatment proposed by Maler and Pnueli [8], and
- to add only *one significant suffix* to the input alphabet according to the approaches of Rivest and Shapire [16].

These steps "aggregate" the futures provided by the counterexamples in order to be able to treat them just like individual inputs in the next iteration, with the result that the signatures directly match the state characterizing vectors of the L^* observation table.

In essence, the DHC algorithm resembles L^* very closely, with the difference that

- it continuously provides a hypothesis also during the first phase, rather than constructing it once at the end of this phase, and
- that this hypothesis, the main data structure of the DHC, is significantly smaller than observation table: it grows with $O(nk)$, whereas the observation tables grows with $O(nk^2 + n^2k)$. Unfortunately, this benefit comes with the necessity of recomputation during each iteration.

Whereas the first difference is nice for illustration or pedagogical reasons, the latter helped us in achieving our size record for active automata learning: a system with over a million states, but with an extremely fast membership oracle. The observation table for this example would have required several terabytes.

Thus there was no way to fit it into main memory, which excluded this example from the scope of our algorithms.

In its first step, the DHC algorithm constructs closed and consistent hypothesis automata by sibling-based completion.

3.1 Step 1: sibling-based completion

Definition 3. *A state is called* complete *if for every input symbol an output symbol and a successor state are determined. A hypothesis is called* complete *if every state of it is complete.*

Whereas it is easy to see that a complete hypothesis is closed in the sense of L^*, consistency is not so obvious. In fact, we will see that it is the result of the hypothesis construction which forces all siblings to have exactly the same successors, namely the one determined by the oldest sibling. This is in fact similar to the approach by Rivest and Shapire [16] or Maler and Pnueli [8].

The DHC algorithm is a classical worklist algorithm. It is initialized with the input-alphabet and it starts with a one-state hypothesis containing only the initial state. This hypothesis is obviously not complete, as it has no transitions yet. The initial state is therefore enqueued into the list of incomplete states, which need to be completed via additional information extracted from membership-queries.

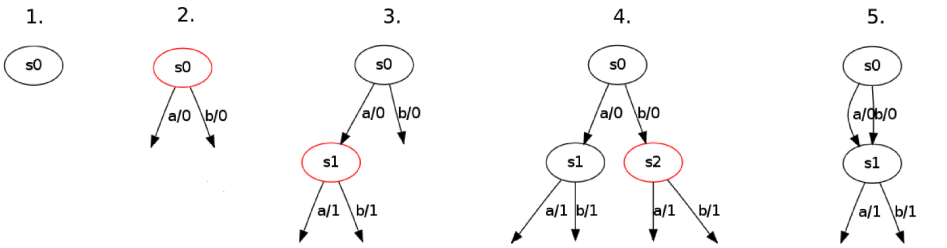

Figure 2. First steps of DHC hypothesis construction with the alphabet {a,b}.

To complete the enqueued incomplete states the algorithm generates a new set of membership queries, which are strings of symbols from the input-alphabet. This is done by determining an access sequence that leads from the hypothesis' initial state to the incomplete state under examination, and extending this sequence with every symbol of the input-alphabet, resulting in k queries. The access sequence can e.g. be determined using a standard Dijkstra search [3] on the current hypothesis or via any other search strategy.

Once a state is completed, there are two possibilities:

1. If it happens to have the same output signature as an already present complete state both stat es are considered equivalent and "merged" as illustrated in the step from 4 to 5 of Figure 2.
2. Otherwise, new successor states are created for each input symbol and enqueued as incomplete states in the worklist. This step clearly reveals the breadth-first exploration scheme, ensuring that any edge redirected to a previous state with identical signature is then pointing at the same or lower search level.

The sibling-based completion terminates once the queue of incomplete states is empty.

Figure 2 shows the first steps of the DHC algorithm. The hypothesis at first includes only the incomplete initial state, which is completed using membership queries. This results in new incomplete states, that are completed using additional queries. In this example both successors of the initial state show the same output behavior after completion, which causes them to be merged.

A pseudocode representation of the core algorithm is given in Figure 3. The method `getAccessSequence` returns a sequence of input symbols which reaches the respective state from the initial state of the hypothesis. The `doMembershipQuery` method expects a sequence of input symbols and returns the system response by means of a membership query. An eventual sibling to the current state will be retrieved by the `findStateWithSameSignature` method. If a sibling is found, the current state will be removed from the hypothesis by the `remove` method, after ensuring any transitions are redirected to the detected sibling by invoking `rerouteAllTransitions`. If no sibling was found, `createSuccesorsForEveryTransition` will create new states for every transition of the retained state, which subsequently are enqueued for exploration.

3.2 Step 2: Refining the input alphabet

The second step deals with the case when the equivalence oracle returns a counterexample. A simple way to treat a counterexample is to add all the suffixes of the counterexample to the input-alphabet and re-start the learning with the extended alphabet. This resembles the approach presented in [8] for guaranteeing that the newly started breadth-first exploration will take into account the knowledge about the diverging behavior inherent in the counterexample.

A more efficient approach to counterexample treatment analyzes the counterexample in order to determine a suffix d which separates two previously assumed to be equivalent states[16]. This can be achieved using a binary search on the counterexample and some additional MQs to pinpoint the diverging behavior [17]. The DHC algorithm then proceeds with step 1 after adding the suffix d as a new (artificial) symbol to the input alphabet. This guarantees that each additional input symbol leads to at least one more state. Table 1 illustrates the impact of this improved counterexample treatment according to a number of different methods explained in the next Section. As we see comparing the first

```
function DHC(Alphabet alphabet) {
  Hypothesis hypo = new Hypothesis();
  Queue statesToComplete = new Queue();
  statesToComplete.enqueue(hypo.getStartState());

  while(statesToComplete.isNotEmpty()) {
    State currentState = statesToComplete.dequeue();
    Sequence accessSeq = currentState.getAccessSequence();

    for(Symbol sym in alphabet) {
      Query query = accessSeq.append(sym);

      // Communicate with the target system, fetch the output symbol
      Symbol output = doMembershipQuery(query);

      // set the transition output for the sym input-symbol
      // to the retrieved output symbol.
      currentState.setTransitionOutput(sym, output);
    }

    State sibling = hypo.findStateWithSameSignature(currentState);
    if(exists(sibling)) {
      // reroute all transitions to currentState to sibling
      hypo.rerouteAllTransitions(currentState, sibling);
      hypo.remove(currentState);
    } else {
      currentState.createSuccesorsForEveryTransition();
      for(State successor of currentState) {
        statesToComplete.enqueue(successor);
      }
    }
  }

  return hypo;
}
```

Figure 3. Pseudocode of the DHC core algorithm.

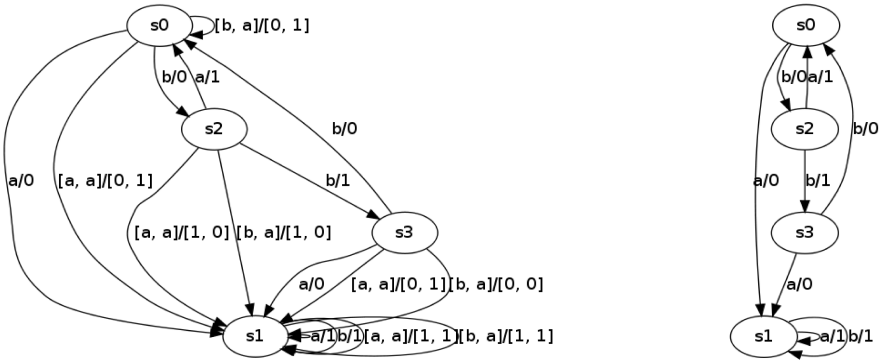

Figure 4. The effect of splitters

with the last row, the gains are significant and vary from a factor 10 to a factor 30.

After extending the alphabet in one of these ways the breadth-first exploration could proceed as described in the previous section. The artificial input symbols, which we call *splitters*, are indeed sequences of input symbols only introduced to split states previously considered equivalent. Splitters are somewhat different from the normal alphabet symbols. They should, e.g., not be represented in the final result presented to a user, and they need not be considered when filling the worklist after having detected a new state as will be discussed in Section 4. Figure 4 illustrates the effect of splitters.Splitters are indicated by the square brackets in the left graph (for instance, the splitter "[a,a]" joined by output such as "[0,1]") and removed in the right graph, the actual learning result.

4 Notes on efficient implementation strategies

While the DHC algorithm itself is simple and can be implemented with little effort, there are some hurdles to overcome to make the implementation actually perform well. The following suggestions employ standard algorithm-engineering approaches. To evaluate the impact of the proposed implementation strategies, we conducted experiments with an implementation [10] that allows enabling and disabling specific optimizations.

Exploring only successor states of original alphabet symbols: As can be observed in Figure 4, states reached by splitter transitions are also reached by successively following the transitions of the splitters' individual symbols. This is guaranteed to be the case, as splitters are unrolled into sequences of individual symbols on membership query construction, which generates identical traces to successively following transitions of the individual symbols. Thus, as splitter transitions do not reach states not reached otherwise, it is not necessary to to enqueue any

Split	Ref	Sib	49 states	66 states	88 states
			17.90 s	49.59 s	353.41 s
		✓	10.70 s	30.67 s	210.04 s
	✓		8.95 s	23.21 s	162.42 s
	✓	✓	2.03 s	5.08 s	20.92 s
✓			13.68 s	29.86 s	181.69 s
✓		✓	8.42 s	18.78 s	111.10 s
✓	✓		6.77 s	13.96 s	79.89 s
✓	✓	✓	1.70 s	3.44 s	11.39 s

Table 1. Comparison of the impact of the optimizations.

successor state of splitter transitions. For the provided example automaton, for instance, this decreases the number of generated membership queries from 78 to 46, the same number the algorithm of [7] would require when following the same path of construction.

Adding only one splitter per counterexample (Split): To complete a state, one membership query per alphabet symbol will be generated, meaning that the number of MQs generated by the DHC learning algorithm depends on the effective size of the alphabet, that includes the original input-alphabet Σ and the splitters added during the enhancement of the signatures. As described in Sect. 3.2, it is possible to analyze counterexamples so that only one splitter needs to be added to the alphabet, which drastically decreases the size of the required signatures, and consequently the number of membership queries.

The impact of this optimization is presented in Table 1, which shows that employing this optimization, denoted as *Split* in the table, decreases the overall runtime for all examined examples.

Determining access sequences quickly (Ref): For every incomplete state we need to determine an access sequence for the construction of membership queries. A straightforward approach would be to, e.g., employ a Dijkstra search, which is a safe bet and will work in all cases. Figure 5, however, shows that the Dijkstra search (not surprisingly) scales extremely poorly with increasing state count.

This can be overcome by maintaining the reverse edges of the spanning tree when constructing the DHC hypothesis, which immediately allows one to collect the access sequence in linear time. Table 1 illustrates the impact of this optimization here indicated as *Ref*.

Finding siblings fast (Sib): Finding siblings by searching the complete hypothesis becomes increasingly slower as the hypothesis grows as is shown in Figure 5, where employing a graph-wide search for siblings scales unfavorably with increasing state counts.

A good optimization is to sort all states with an unique output signature into a decision tree, which guarantees a lookup time solely depending on the

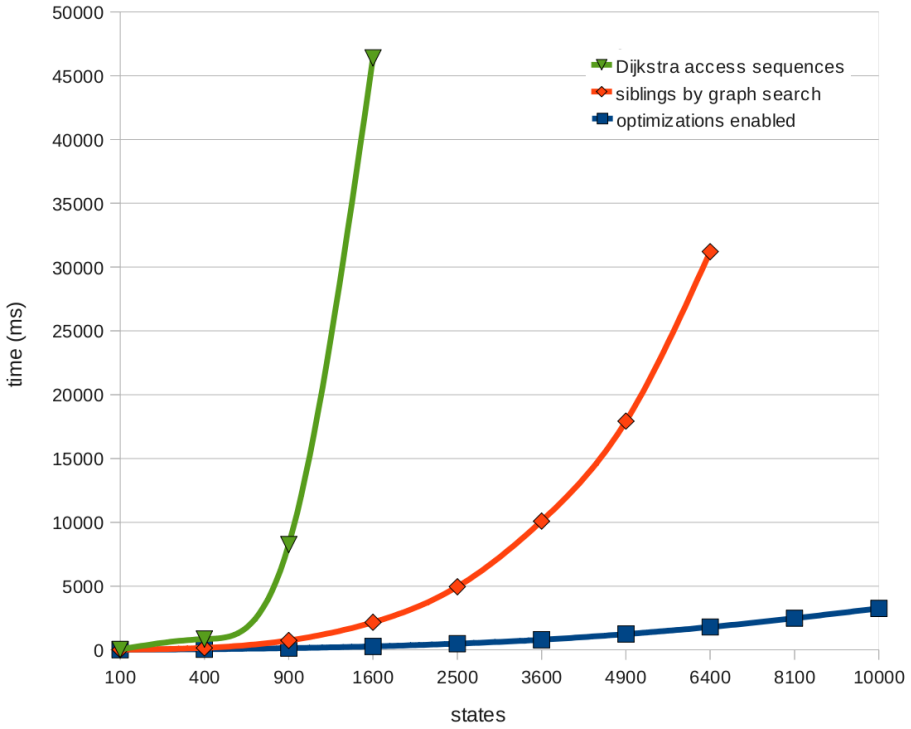

Figure 5. The impact of the optimization strategies on scalability.

input-alphabet of the hypothesis. The immense impact of this optimization is visible in Figure 5, where this optimization is included in *optimizations enabled*, and also in Table 1, where this optimization is referred to as *Sib*.

The overall impact of our optimizations is summarized in Figure 5, which in particular shows how nicely the optimized algorithm scales with growing state spaces.

5 Conclusion

In this paper we have presented an automata learning algorithm that follows the well-known breadth-first-oriented exploration pattern and is thus, due to its continuous provision of a visualizable hypothesis structure, fit for educational purposes. Moreover, these DHC hypotheses, the main data structures of the DHC algorithm, are significantly smaller than observation table: they grow with $O(nk)$, whereas observation tables grow with $O(nk^2 + n^2k)$ already in the case of [7], and the estimation for L^* has even an additional factor "maximal size of a

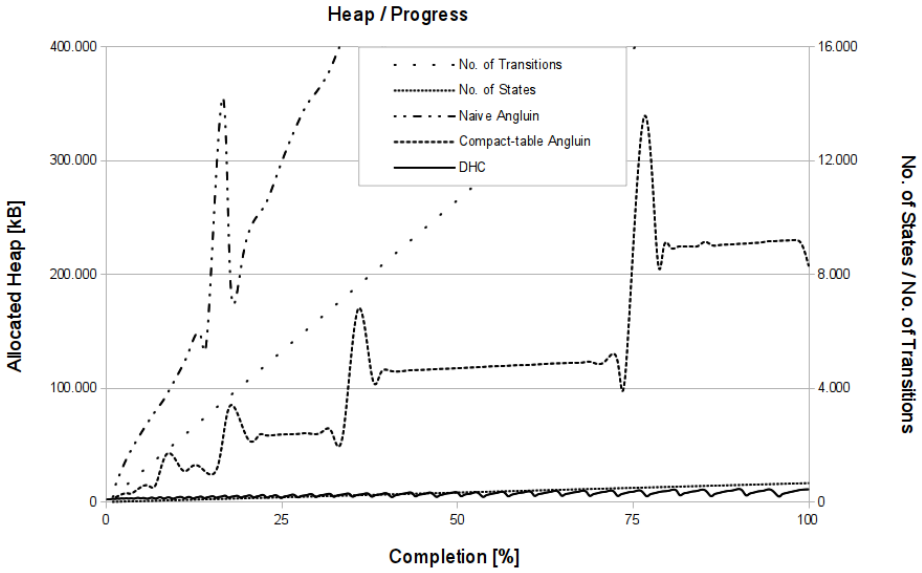

Figure 6. Memory consumption of DHC compared to two algorithms with observation tables.

counterexample". This difference is nicely illustrated in Fig. 6, which clearly indicates that the memory consumption of table-based algorithms correlates with the number of discovered transitions (coarsely dotted line), whereas DHC's memory consumption only grows linearly with the number of discovered states (finely dotted line). [3]

Although this memory efficiency comes at the price of recomputation during each iteration, the DHC algorithm allowed us to raise the record for active learning to systems with more than a million states in settings where membership queries are extremely fast. The DHC algorithm is implemented in the LearnLib [10,9], which is available as free download at `http://www.learnlib.de`.

A closer investigation reveals that the DHC algorithm can be considered as an elegant means for splitting the data maintained in the observation table in two parts, the ones to be kept in main memory, namely the DHC hypotheses, and data that can be kept on disk or SSD, namely a cache, storing the results of all membership queries. This extends the practical impact of DHC's reduction of the memory footprint to large systems where answering membership queries is (very) expensive:

[3] The large spikes seen for the table-based algorithms are caused by copying data from fixed-size data structures into freshly allocated data structures of bigger size, and the ripples in DHC's memory profile can be explained by the from-scratch construction principle whenever a counterexample is delivered, which leads to the deallocation of memory consumed by the disproved hypothesis.

– It allows one to deal with systems of a few hundred thousand states whose observation table would grow far beyond the available main memory.
– The additional cache lookups required for the DHC algorithm do not weight in comparison to the membership querying times.

This separation of concerns is quite similar to the one proposed in [5,6,18] for externalizing the bulk of the required memory for graph search. We are convinced that following this line of thought that considers layered memory hierarchies can be quite successful also for active automata learning, even though their work depends on that fact that the considered graphs are provided in a white box fashion. This excludes the direct use of GPUs as proposed by [18], but it seems to have nevertheless a great potential for parallelization. First results in this direction are reported in [4].

References

1. Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In *ICTSS*, pages 188–204, 2010.
2. Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
4. Falk Howar, Oliver Bauer, Maik Merten, Bernhard Steffen, and Tiziana Margaria. The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning. *Submitted to ISoLA2011*.
5. Shahid Jabbar. External directed search. *KI*, 21(1):37–38, 2007.
6. Shahid Jabbar. *External memory algorithms for state space exploration in model checking and action planning*. PhD thesis, 2008.
7. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
8. Oded Maler and Amir Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995.
9. Maik Merten, Falk Howar, Bernhard Steffen, Sofia Cassel, and Bengt Jonsson. Demonstrating learning of register automata. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 466–471. Springer, 2012.
10. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *Seventeenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, 2011.
11. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
12. Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, Germany, 2003.
13. Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, New York, NY, USA, 2008. ACM.

14. Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.
15. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.
16. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
17. Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valérie Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.
18. Damian Sulewski, Stefan Edelkamp, and Peter Kissmann. Exploiting the computational power of the graphics card: Optimal state space planning on the gpu. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*. AAAI, 2011.