Learning state machines from data streams: A generic strategy and an improved heuristic

Robert Baumgartner r.baumgartner-1@tudelft.nl
Sicco Verwer s.e.verwer@tudelft.nl
Department of Software Technology
Delft University of Technology
Delft, The Netherlands

Editors: François Coste, Faissal Ouardi and Guillaume Rabusseau

Abstract

State machines models are models that simulate the behavior of discrete event systems, capable of representing systems such as software systems, network interactions, and control systems, and have been researched extensively. The nature of most learning algorithms however is the assumption that all data be available at the begining of the algorithm, and little research has been done in learning state machines from streaming data. In this paper, we want to close this gap further by presenting a generic method for learning state machines from data streams, as well as a merge heuristic that uses sketches to account for incomplete prefix trees. We implement our approach in an open-source state merging library and compare it with existing methods. We show the effectiveness of our approach with respect to run-time, memory consumption, and quality of results on a well known open dataset.

Keywords: State machine learning, Automata learning, Streaming data

1. Introduction

State machines are insightful models that naturally represent formal languages and discrete systems, and have been extensively used in various domains such as model checking (Baier and Katoen, 2008) and modeling discrete event systems such as truck driving (Verwer, 2010), computer networks (Pellegrino et al., 2017), and controllers and software systems (Walkinshaw et al., 2016). A major advantage of state machines is that, combined with expert knowledge, they are interpretable (Hammerschmidt et al., 2016). Learning state machines can be roughly subdivided into active learning and passive learning. Active learning learns from interactions with a system under test. Passive learning learns directly from observed execution traces without interfering with a system's execution. Most passive algorithms require all data to be available before running the inference step and hence cannot learn from data streams. Exceptions are the works of Balle et al. (2012, 2014) and Schmidt and Kramer (2014). These works employ a typical streaming approach where decisions made by the learning algorithm are postponed until sufficient data has been observed. In this work, we propose a novel streaming strategy, with its main advantage being that it corrects errors made during previous iterations. In this way, we can perform learning steps even when the statistical tests performed by the algorithm are inconclusive: subsequent iterations can correct these steps. As a consequence, we learn faster. Our streaming strategy is generic and can be used with multiple heuristics.

Like most passive learning algorithms, we adopt state-merging in the red-blue framework from Lang et al. (1998). This framework starts with a prefix-tree that directly encodes the entire input data set and then greedily merges states of this tree until no more consistent merges can be performed. Being a streaming algorithm, our method only keeps counts from all observed traces in memory and builds the prefix tree only for states with sufficient occurrence counts. To learn models from such a partially specified prefix tree, we propose a new Count-Min-Sketch based merge heuristic. The main idea is that we hash future prefixes and store them in sketches in each state. Our new heuristic performs consistency checks directly on these sketches. Importantly, Count-Min-Sketches allow for efficient updates which allows performing and undoing of merges in constant time.

We implemented our approach in the open source FlexFringe library (Verwer and Hammerschmidt, 2017), a flexible state-merging framework that allows to specify one's own heuristics and consistency checks. We added streaming capability and our own consistency check using Count-Min-Sketches. FlexFringe provides efficient structures for performing and undoing merges, as well as a multitude of heuristics such as Alergia (Carrasco and Oncina, 1994) that we use in our experiments. We demonstrate the effectiveness of our approach on the well known PAutomaC dataset. We evaluate on run-time, memory footprint and approximation quality. We experiment in settings where merge undoing is enabled, and the more common streaming setting where merges are postponed until sufficient data is available. In our experiments our method clearly outperforms the earlier approach of Balle et al. (2014) in terms of approximation quality. It also compares favorably to streaming using Alergia as the heuristic without merge undoing and performs better on most problems when undoing merges is enabled, albeit the gap has shrank compared to the not-undoing setting. The results clearly demonstrate that the ability to correct mistakes is crucial for obtaining good performance using traditional merge heuristics such as Alergia. The Count-Min-Sketches result in improved consistency checks when postponing merges, but even these benefit from merge undoing. Our method gets close to but does not reach the performance of a non-streaming version of Alergia that simply loads all the available data in the prefix tree.

2. Related Work

There are two main approaches to learning state machines, either by active learning (Angluin, 1987, 1988; Vaandrager, 2017), or via state merging (de la Higuera, 2005). While active learning requires the availability of an oracle being able to answer queries, state merging learns from example traces. Example traces are being represented via a complete tree, and then greedily minimized in accordance to Occam's razor. These all perform a consistency test of where a pair of state is mergeable and compute a heuristic value to determine which merge to perform first. For example, Alergia (Carrasco and Oncina, 1994) learns state machines representing probabilistic languages via sampled distributions, and k-Tails (Biermann and Feldman, 1972) compares the subtrees of state pairs. Other seminal works include the RPNI algorithm (Oncina and García, 1992) and the EDSM algorithm (Lang et al., 1998). Several types of models can be learned like this. Verwer et al. (2012) learn timed automata from timed strings, Walkinshaw et al. (2016) learn extended finite state machines to represent software and control systems, and Mariani et al. (2017) learn guarded finite state machines. Hybrid state machine models can be learned by taking different aspects of a system into account (Vodenčarević et al., 2011; Lin et al., 2018). Since the problem of finding an exact solution in passive learning has been shown to be NP-hard (Gold, 1978), several search strategies have been developed (Oliveira and Silva, 1998; Abela et al., 2004; Lang, 1999). Different ways to speed up the main algorithm have also been proposed through divide and conquer and parallel processing (Luo et al., 2017; Akram et al., 2010; Shin et al., 2021).

Few works deal with making state-merging algorithms more scalable when run on streaming data. Schmidt and Kramer (2014) utilize frequent pattern data stream techniques to tackle the problem. Dupont (1996) build on the RPNI algorithm and learn state machines from positive and negative examples in an incremental fashion. Conflicts that can arise from new unseen negative examples are resolved via splitting states until conflicts are resolved. Balle et al. (2012) present a theoretical work of streaming state machines using modified space saving sketches (Metwally et al., 2005), and extend it with a parameter search strategy in (Balle et al., 2014). Schouten (2018) implements a streamed merging method in the Apache framework, also using the Count-Min-Sketch data structure (Cormode and Muthukrishnan, 2005). In this paper, we present a new streaming learning method that, like the algorithm of Dupont (1996), can correct mistakes from earlier iterations, and like the algorithm of Balle et al. (2012) uses sketches to approximate the information contained in the observed traces.

3. Background

A PDFA is a tuple defined by $\mathcal{A} = \{\Sigma, Q, q_0, \tau, \lambda, \eta, \xi\}$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is a unique starting state, $\tau : Q \times \Sigma \cup \{\zeta\} \to Q$ is the transition function with ζ the empty string, $\lambda : Q \times \Sigma \to [0, 1]$ is the symbol probability function, and $\eta : Q \to [0, 1]$ is the final probability function, such that $\eta(q) + \sum_{a \in \Sigma} \lambda(q, a) = 1$ for all $q \in Q$. $\xi \notin \Sigma$ denotes the final symbol, indicating the end of a sequence.

In the following we will stick to the following convention: We denote the Kleene star operation over Σ by Σ^*, and denote $x\Sigma^*$ the set of all possible strings with prefix x. When we decompose a string into multiple parts or when referring to general strings over Σ^* we use the letter σ. Single elements of the alphabet are denoted by the character a, s.t. $a \in \Sigma$. For example, the string $\sigma_1 a \sigma_2$ is the string obtained through the concatenation of string σ_1, the symbol a and the string σ_2. Given string $\sigma = a^1 a^2 \dots a^n$, we call the sequence $a^{i+1} a^{i+2} \dots a^{i+m}$, $i \in [0, n - m]$ a substring of length m of σ. Given transition function τ, a string traverses the PDFA recursively in order $\tau(q_i, a^{i+1}) = q_{i+1}$ for all $0 \leq i \leq n - 1$. For convenience we define a traversal through an entire string or substring σ via shorthand notation $\tau(\sigma)$. We consider a parent of a state q a state q' s.t. $\exists a \in \Sigma : \tau(q', a) = q$. In the prefix tree each node has exactly one parent.

The probability $P(\sigma)$ of a string $\sigma = a^1, \dots, a^n$ can be computed via $P(\sigma) = \lambda(q_0, a^1) \cdot \lambda(q_1, a^2) \cdot \dots \cdot \lambda(q_{n-1}, a^n) \cdot \eta(q_n)$. In this work we model sequential information via sampled distributions over strings emanating from a state q_i via $D_{q_i}(\sigma), \sigma = a^{i+1} a^{i+2} ...$, where $D_q(\sigma) \to [0, 1]$ models a sampled distribution. We call a (sub-)string σ_2 an outgoing string from state q if $\exists \sigma' = \sigma_1 \sigma_2 : \tau(\sigma_1) = q$. We further define the size s_q of a given node q by the number of input strings $\sigma \in I$ from input I during the learning that traverse q, i.e.

$n_q = |\{\sigma \in I | \sigma = a_1 a_2 ... a_{|\sigma|} \wedge \exists i \in [1, |\sigma|] : \tau(q_{i-1}, a^i) = q\}|$. The size of the root node q_0 is $n_{q_0} = |\{\sigma \in I\}|$.

PDFAs can be learned via state-merging. In a first step, a state-merging algorithm constructs a prefix tree representing the input data completely, meaning that every state in this tree has only one unique access sequence. Algorithms differ in how they minimize this tree. Usually heuristics are employed to greedily merge equivalent state pairs (q, q'). In each iteration the heuristic will assign a score ϕ to every mergeable state pair (*candidate pair*), and the merge with the highest score will be performed. After merging two states into one, a subroutine is started such that τ is deterministic, i.e. that there is only one possible transition $\tau(q', a')$ for all $q' \in Q$, $a' \in \Sigma$. Multiple search strategies exist to limit the search of state pairs. In this work we stick close to the red-blue-framework (Lang et al., 1998). The first red state is the root node of the prefix tree, and blue states are all states emanating from red states. Only pairs of one red and one blue state can be merged, and the state resulting from a merge is always red. In case no merge can be performed, the blue state q with the largest size s_q will be turned red. Note that in this framework nodes that are not red always have exactly one parent, and the parent of a blue node is always red.

4. Methodology

4.1. Merge heuristic

The core idea of our merge heuristic is to store the counts of outgoing strings of each state. Because this quantity can become very large in data streams we use the Count-Min-Sketch (CMS) (Cormode and Muthukrishnan, 2005) data structure, and we equip each state with one such CMS. We store each state's outgoing strings in its sketch. Inserting elements and retrieving them can be done in time $\mathcal{O}(1)$, and only depend on the size of the sketches. Two CMS can be considered as matrices, and states can easily be merged and unmerged via matrix addition and subtraction. To compare two sketches we retrieve the counts of all seen strings σ and model the distribution over those as frequencies. We can then compare these two distributions via statistical tests. A problem that arises is that the number of possible strings grows $\mathcal{O}(|\Sigma|^{F_s})$, where we denote as F_s the maximum length of the strings that we consider (a hyperparameter). We propose two solutions to tackle this problem: The first one by constructing multiple sketches per state, one for each possible size of string. In this setting, the first sketch would only store strings of size 1, the second sketch strings of size 2, and so on. The second solution is concerning the hash-function that is utilized in the CMS. We describe the heuristic in more detail in the following subsection.

4.1.1. Count-Min-Sketches and the heuristic

We consider two states behaving similar if their multiset of outgoing strings is similar. We consider the regular set of outgoing strings of a state q the set $\{\sigma \in \Sigma^* | \exists x \in \Sigma^* \wedge \sigma' = \sigma : \tau(x) = q\}$. The multiset of outgoing strings of state q is simply the respective multiset, that also stores the number of times each of the elements has been attempted to be inserted into the set. We denote the count of an arbitrary element y via c_y.

Given state q, we assume a sampled distribution $D_q : \sigma \rightarrow [0, 1]$, again with $\sigma \in \Sigma^*$[1]. For us, $D_q(\sigma_i)$ is simply the frequency of string σ_i, that is $D_q(\sigma_i) = \frac{c_{\sigma_i}}{n_q}$. Because the set Σ^* can be potentially very large, we approximate D_q for each state via a variant of the Count-Min-Sketch (CMS) data structure (Cormode and Muthukrishnan, 2005), which is why we call our heuristic CSS (CMS-based Space Saving).

Formally, a CMS is a probabilistic data structure to summarize data streams. In practice a CMS is a matrix represented by $d = \left\lceil \ln \frac{1}{\gamma} \right\rceil$ rows and $w = \left\lceil \frac{e}{\beta} \right\rceil$ columns, where e is the basis to the natural logarithm ln. Given counts c_{y_i} of elements y_i of a given set of possible events $\mathcal{Y}$, the CMS is able to store the counts and retrieve them using $\mathcal{O}\left(\frac{1}{\beta}\right)$ space and $\mathcal{O}\left(\frac{1}{\gamma}\right)$ time. To do so each row of the CMS is associated with a hash-function h mapping elements y of set $\mathcal{Y}$, $y \in \mathcal{Y}$, to $h : y \rightarrow \mathbb{N} \cap [0, w - 1]$. That means in total there exist d hash functions, and we want them to be i.i.d. chosen from a pairwise-independent hash-family $\mathcal{H}$ (Cormode and Muthukrishnan, 2005). We denote h_j the hash function associated with row j, and for convenience we define an inverse mapping $h_j^{-1}(y)$. At initialization, each entry of the CMS is set to zero. To store an incoming element y the CMS hashes y for each row j and increments the row at $h_j(y)$ by 1. Retrieving an approximated count of a given element y works via again hashing the y once per row. The approximated count is the minimum $\min_{j \in [1,d]} h_j^{-1}(y)$. The CMS is then able to retrieve an approximated count $\hat{c}_{y_i}$ of y_i via the $retrieve(y_i)$ operation. The error bound is $\hat{c}_{y_i} \leq c_{y_i} + \beta \sum_i c_{y_i}$, which holds with probability at least $1 - \gamma$.

As already mentioned we count strings $\sigma \in \Sigma^*$, and we denote the count of string σ_i by c_{σ_i} and its approximated count by $\hat{c}_{\sigma_i}$. In addition to conventional CMS our data structure one extra attribute and two extra operations. The extra attribute is a counter for the final counts ξ, indicating that a sequence did end in the previous state. Our sketches further support a $+$ and a $-$ operation, which we define as the sum and subtraction of two matrices of equal size, and the same for the final counts. These two operations will be used to perform and undo merges. In order test whether two states behave similar we perform the statistical test from the works of Carrasco and Oncina (1994), which they use in their own Alergia-algorithm. This statistical test, herein after referred to as Alergia-test, is used to check whether two sampled distributions are similar. State-pairs that pass this test get assigned a score ϕ. To this end we use the Cosine-similarity. Given two vectors v_1 and v_2 the Cosine-similarity measures the angle in between the two vectors via

$$cosine\_similarity(v_1, v_2) = \frac{v_1 \cdot v_2}{||v_1||_2 ||v_2||_2}, \tag{1}$$

where $v_1 \cdot v_2$ is the dot-product of the two vectors, and $||v_i||_2$ denotes the L2-norm of vector v_i. We show the entire subroutine including the Alergia-test in Algorithm 1. In this subroutine α is a hyperparameter to be set before the start of the learning algorithm. It represents a bound on the probability of a wrong rejection, which is upper bounded with 2α (a wrong rejection means that the two distributions are similar, but the test deems them dissimilar).

1. Recount that the Kleene-star operation applied to Σ, i.e. Σ^*, denotes the set of finite strings. Hence the set of all $\{\sigma \in \Sigma^*\}$ is a finite set and we can obtain discrete probabilities for the occurrence of each string.

Algorithm 1: *Consistency − routine*

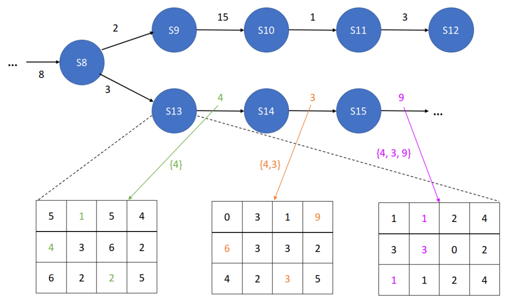
```
Input:
$CMS_{q_1}$, $CMS_{q_2}$: Count-Min-Sketches of state $q_1$ and $q_2$ respectively
$n_{q_1}$, $n_{q_2}$: Size of state $q_1$ and $q_2$ respectively
$S$: The set of all strings $x$ observed so far. $\alpha$: Hyperparameter to be set.
Output: Boolean value indicating consistency, score $\phi$ if applicable
$v_1$, $v_2 \leftarrow$ empty lists
foreach $\sigma \in S$ do
    $\hat{c}_{\sigma_{q_1}} \leftarrow CMS_{q_1}.retrieve(\sigma)$
    $\hat{c}_{\sigma_{q_2}} \leftarrow CMS_{q_2}.retrieve(\sigma)$
    if $\left|\frac{\hat{c}_{\sigma_{q_1}}}{n_{q_1}} - \frac{\hat{c}_{\sigma_{q_2}}}{n_{q_2}}\right| > \sqrt{\frac{1}{2} log\left(\frac{2}{\alpha}\right)} \left(\frac{1}{\sqrt{n_{q_1}}} + \frac{1}{\sqrt{n_{q_2}}}\right)$ then
        return false
    $v_1.append(\hat{c}_{\sigma_{q_1}}/n_{q_1})$, $v_2.append(\hat{c}_{\sigma_{q_2}}/n_{q_2})$
end
return true, cosine_similarity($v_1$, $v_2$)
```

4.1.2. The runtime problem

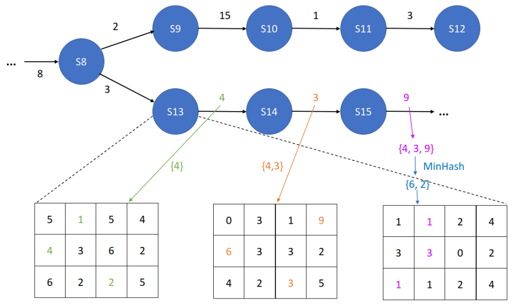
In practice we have to bound the size of outgoing strings of states, because they can be arbitrarily long. We denote the maximum length of an outgoing (sub-)string for any state $q \in Q$ by F_s. The run-time problem then is that the size of set S from Algorithm 1 does grow with $\mathcal{O}(|\Sigma|^{F_s})$. We propose two solutions to overcome this problem. The first one is a simple decoupling. Instead of storing all strings in one sketch, we construct F_s sketches per state, namely $C_1, C_2, \ldots C_{F_s}$, and each of them stores one size of string in ascending order: (sub-)strings of length 1 $\rightarrow C_1$, (sub-)strings of length 2 $\rightarrow C_2$, ..., (sub-)strings of length $F_s \rightarrow C_{F_s}$. Consistency checks are then performed in ascending order on those CMS, and higher order CMS are only checked if all lower order CMS passed the test already. The score ϕ is simply the average of all scores $\phi_1 \ldots \phi_{F_s}$ returned by those checks. Figure 1(*a*) illustrates an example, where state $S13$ stores sketches with $F_s = 3$.

While this helps us reduce runtime in practice at the cost of some memory, the worst case run-time remains at $\mathcal{O}(|\Sigma|^{F_s})$. This can practically be a problem on data where a large F_s is required to distinguish states properly. In this case we propose a second solution. We adopt the ideas introduced by Locality Sensitive Hashing (LSH) (Gionis et al., 1999; Datar et al., 2004). Intuitively, LSH hashes elements of some domain W of dimensionality r_1 into another domain U of dimensionality $r_2 < r_1$ while preserving a distance measure of choice. The hashing-algorithm is based on the chosen distance measure. Because we are dealing with set of discrete symbols $a \in \Sigma$ we choose to preserve the Jaccard similarity of two strings σ_1 and σ_2, denoted by $Jaccard(\sigma_1, \sigma_2)$, and hash the strings using the MinHash-algorithm (Broder, 1997; Carl Kingsford, 2016). Taking a hash-function h_i from hash-family $\mathcal{H} = \{h : \mathcal{W} \rightarrow \mathcal{U}\}$ we obtain $P(h_i(\sigma_1) = h_i(\sigma_2)) = Jaccard(\sigma_1, \sigma_2)$, where $Jaccard(\sigma_1, \sigma_2) = \frac{|\{\sigma_1\} \cap \{\sigma_2\}|}{|\Sigma|}$. In this notation we use the shorthand-notation $\{\sigma_i\}$ to denote the set $\{a' \in \Sigma | \sigma_i = a^1 a^2 \ldots a^n, \exists j \in [1, n] : a^j = a'\}$. In other words, the more similar two strings, the more likely they are being hashed into the same bin. We introduce a new hyperparameter

l_m, and define l_m mutually-independent hash-functions $h_i, i \in [1, l_m]$. Because our goal is to reduce run-time we only need to hash strings with length larger than l_m, hence the run-time is then $\mathcal{O}(|\Sigma| + |\Sigma|^{l_m})$, where the $|\Sigma|$ term stems from the maximum number of attempts it takes to hash a string of length F_S. Figure 1(b) shows the approach with a hashing down to size 2. We herein after call this approach *CSS-MinHash*.



(a) An example of how the futures are stored within the sketches. The numbers are purely fictional.



(b) Illustration of CSS-MinHash. Here we hash down to a size of 2.

Figure 1: The two solutions to the problem of uniform distributions in the sketches.

4.2. Streaming

Our streaming algorithm consists of two main ideas. The first one is that we discard information that does not occur often enough. This is common practice in streaming algorithms. To do so we use the red-blue framework and include a threshold t_S, where states can only be created with red or blue states as parent nodes, and a state's size n_q has to exceed t_S to be able to become a blue node. The second idea is to undo and redo merges. In our approach we divide the incoming data stream into batches of size B. After reading B sequences from the stream, we first perform a greedy minimization step. We keep track of all the operations (herein after called *refinements*) we performed on the prefix tree. A refinement is either the identification of a new red state or the merge of a state-pair. Once we cannot perform refinements anymore, we save the resulting automaton and reverse all the refinements we did, starting anew. In the next iteration we will first try to perform all the refinements from the previous iteration again, and then perform greedy minimization.

Because we can logically separate the two main steps we perform, namely the batch-wise streaming of the prefix tree and the subsequent minimization routine, we have split those up into two algorithms. We describe the two steps individually in the next two subsections.

4.2.1. Streaming the prefix tree

Streaming the prefix tree starts with the root node marked as a red state. We read in B elements of the data stream, and create states emanating from the root node. Those states are neither red nor blue, unless they have been accessed by an incoming sequence at least t_S times, in which case we mark them blue. States can only be created emanating from red or blue states, thus in each iteration the fringe of the prefix tree can grow at a maximum of

Algorithm 2: Streaming the tree

```
Input: Stream of sequences $I$, batch-size $B$, threshold $t_S$, upper bound on states $n$
Output: Hypothesis automaton $H$
$H \leftarrow$ root node
$c \leftarrow 0$
$R_{old} \leftarrow$ empty queue
foreach $\sigma_i \in X$ do
    $\sigma_i \leftarrow \sigma_i \xi$
    $q \leftarrow$ root node
    $n_q \leftarrow n_q + 1$
    update $q$ with relevant data
    foreach $j \in len(\sigma_i)$ do
        $a^j \leftarrow \sigma_i[j]$
        if transition from $q$ with $a^j$ exists then
            $q \leftarrow \tau(q, a^j)$
        else if $q$ is red or $q$ is blue then
            create new node $q'$
            set $\tau(q, a_j) = q'$
            $q \leftarrow q'$
        else if $q.parent$ is red and $n_q \geq t_S$ then
            mark $q$ blue
        else
            break inner loop
        $n_q \leftarrow n_q + 1$
        update $q$ with relevant data
    end
    $c \leftarrow c + 1$
    if $c == B$ then
        $R_{old} \leftarrow perform\_minimization\_routine(H, R_{old})$
        $c \leftarrow 0$
    if size of last $H = n$ or $I$ empty then
        $perform\_minimization\_routine(H, R_{old})$
        return $H$
end
```

two more states from each existing state. We do this to only save relevant information of the common behavior of the system. Every time a node q gets traversed by a string $\sigma \in I$ we update it with data. This operation depends on the merge heuristic that we chose at the start of the program. In case of our CSS-heuristic we update the node with its outgoing strings.

Once a batch is completed a minimization routine, described in section 4.2.2, will start. The minimization routine returns a hypothesis automaton and marks all the states that have been red or blue as a result of the minimization routine. We then start reading in another batch as described before, and the minimization routine starts again. The algorithm is depicted in Algorithm 2.

4.2.2. Minimization routine

The core idea of our minimization routine is that, depending on the implementation, undoing and performing refinements again can be done cheaply. Performing merges and undoing them can be done cheaply in constant run-time in Flexfringe (Verwer and Hammerschmidt, 2017). The advantage of undoing and redoing refinements is that the learner gets the chance to correct mistakes earlier. This way we can return a model earlier without compromising its future correctness. The first time we start the minimization routine it performs the normal minimization routine described in section 3. What is new is that we store all the refinements that are performed in this step in a queue R_{new}. We save the found automaton as hypothesis H and undo all refinements that we did. At the end of the subroutine we we save R_{new} as another queue R_{old} and return to the prefix-tree streaming subroutine.

From the second time onward, when we enter the minimization subroutine we will start with R_{old} and retry every refinement we performed in the previous run. Assuming that the underlying distribution of strings from the data stream did not change most of these refinements will still hold, hence this approach saves us run-time. The way we deal with the case that a refinement is not possible anymore is based on the following observation. We identified two main causes why a refinement cannot take place anymore:

1. Consistency-failure: The underlying structure of the node has changed according to the merge heuristic. It could be for instance that the distribution of strings that passed it changed significantly.

2. Structural-failure: The underlying structure of the tree has changed. This can be caused by previous consistency failures, in which case the sub-tree might change.

While consistency-failures cannot be fixed in this batch, we follow the intuition that refinements with structural failures can often still hold after obtaining an appropriate tree structure. As an example, say that an earlier performed merge refinement of two states q_1 and q_2 cannot be performed, because state q_2 is neither red nor blue anymore. We then continue the minimization procedure and later reconsider what to do with q_2. If at that time q_2 is blue, we perform the merge if it is valid. The advantage of this strategy is that it avoids having to recompute consistency checks where unnecessary. Compared with structural checks, which can be done in a simple flag check, consistency check are much more expensive. Our strategy works then as follows.

We first test on structural failures on each refinement $r \in R_{old}$. In case a structural failure did not happen, we test on a consistency failure. We perform the refinement if none of those two failures did occur. If a consistency failure occurred we discard r and test for the next refinement from R_{old}. If however only a structural failure occurred, we push r onto another queue R_{failed}. Once we've exhausted R_{old} we perform the greedy minimization again. This time however, each time a new refinement is identified we iterate once over R_{failed} and test again for both failure modes. If no failure can be detected we perform the refinement. Just as in the first iteration we store all the refinements performed in R_{new} and save it as R_{old} at the end of the subroutine. The whole procedure is described in Algorithm 3.

Algorithm 3: Minimization routine

```
Input: Current hypothesis $H$, queue $R_{old}$
Output: queue $R_{new}$
$R \leftarrow$ empty stack
$R_{failed}$, $R_{new} \leftarrow$ empty queue
while $R_{old}$ not empty do
    $r \leftarrow R_{old}.pop()$
    if $r$ possible via structure and via consistency then
        perform $r$ on $H$
        $R.push(r)$
        $R_{new}.push(r)$
    else if $r$ possible via structure then
        $R_{failed}.push(r)$
end
while $H$ contains at least one blue or white state do
    Select best possible refinement $r$
    perform $r$ on $H$
    $R.push(r)$
    $R_{new}.push(r)$
    foreach $r \in R_{failed}$ do
        if $r$ possible via structure and possible via consistency then
            $R_{failed}.delete(r)$
            perform $r$ on $H$
            $R.push(r)$
            $R_{new}.push(r)$
    end
end
output $H$
while $R$ not empty do
    $r \leftarrow R.pop()$
    undo $r$ on $H$
end
return $R_{new}$
```

5. Experiments and results

We implemented our heuristic and our streaming approach in Flexfringe (Verwer and Hammerschmidt, 2017; Verwer and Hammerschmidt). We compared with the Alergia-algorithm (Carrasco and Oncina, 1994) as implemented in the framework, and with the heuristic of Balle et al. (2012), herein after called *SpaceSave*. In order to compare the results we used the PAutomaC-dataset (Verwer et al., 2014), consisting of 48 scenarios. Each scenario comes with a set of example traces extracted from existing automata, and the task is to infer the original automaton. Each scenario also comes with a test set, consisting of string-probability pairs, assigning a probability of each string to the real automaton. In the subsequent experiments we first learn a state machine for each scenario from the dataset.

We can test the quality of our learned state machines via comparison of the divergences in between the assigned probability-distributions by the learned machine and the provided true probability of the strings to occur. Similar to the PAutomaC-competition (Verwer et al., 2014), we use the perplexity score to compare the two distributions. The smaller the perplexity, the closer the two distributions are. All results that we report are produced by a notebook with the following relevant specifications:

1. Operating system: Ubuntu 20.4
2. CPU: Intel i7@2.60Ghz
3. RAM: 16GB

5.1. The heuristic

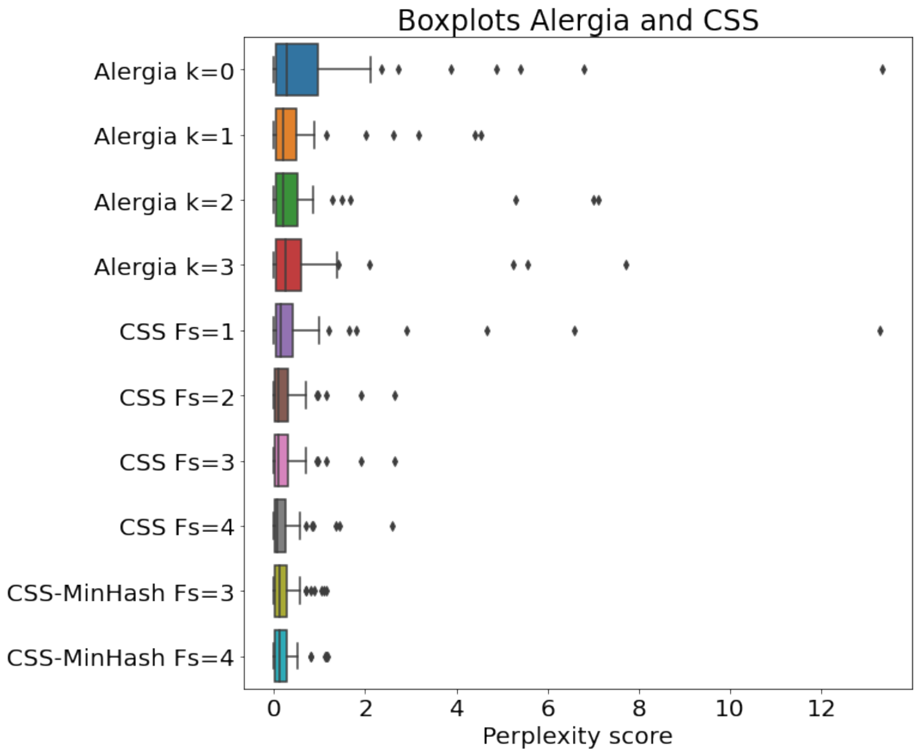
To compare the heuristics we first run them all in normal *batch mode*. In order to better compare Alergia with the other two heuristics we augment Alergia with the well-known k-*tails*-algorithm (Biermann and Feldman, 1972). In this case the merge procedure remains the same as in the Alergia algorithm, but the consistency-check includes checking the sub-trees of each node pair up to a depth of k. The reason for this is that the Alergia heuristic naturally does not have the lookahead-feature that CSS provides. In our experiments we found that the results stop improving after $k = 3$, so we only report results up to $k = 3$. We then run the same experiments with CSS, varying the F_S parameter from a length of 1 up to a length of 4. Note that Alergia with $k = 3$ and CSS with $F_S = 4$ both look 4 steps ahead of each state, hence the comparison is fair. We also test CSS-MinHash, where we hash down the strings to a size of 2, while keeping F_S at 3 and 4 respectively. Last but not least we did the same experiments with the SpaceSave heuristic. We report the perplexities of all heuristics but the SpaceSave heuristic in Fig. 3(a), and we report the results of the SpaceSave heuristic in Fig. 2(b). We chose two plots because the perplexities for the SpaceSave are much higher, hence putting them into the same plot as the other heuristics would make the results of the other heuristics illegible.

Additionally, for all the steps we measured the times it took to run through. The times are in Table 1.

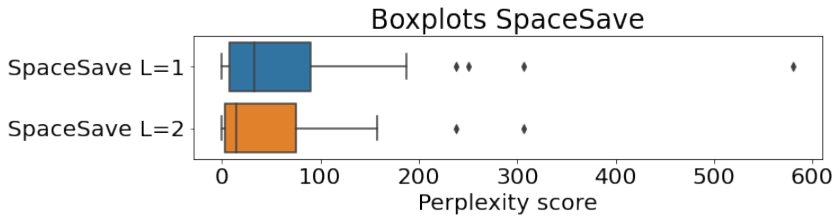
Heuristic	$F == 0$	$F == 1$	$F == 2$	$F == 3$	$F == 4$
Alergia	1m45s	1m55s	2m8s	2m24s	-
CSS	-	2m1s	2m15s	2m14s	2m50s
CSS-MinHash	-	-	-	3m14s	3m40s
SpaceSave	-	4m37s	6m51s	-	-

Table 1: Runtime comparisons of the heuristics and different future length parameters F, F being a placeholder for either k, F_s, or L. Empty fields do not exist or are not interesting to us.

We can see that the runtimes of CSS are small, and that CSS-MinHash took longer. The reason here is that for the dataset even small steps ahead are enough to distinguish them



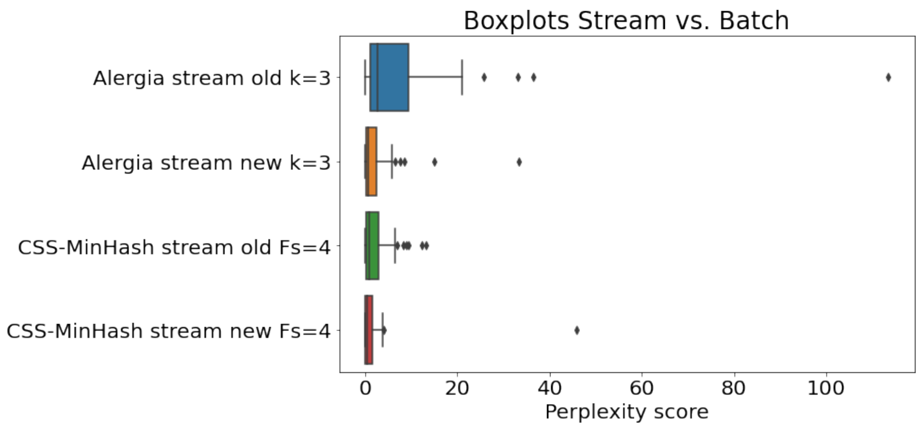
(a) Comparison of all heuristics tested but the SpaceSave heuristic. All tested heuristics come with varying lookahead parameters (k for Alergia, F_S for CSS).



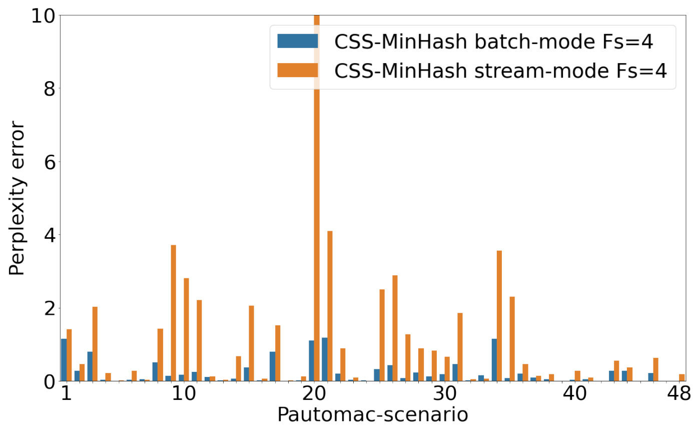
(b) Errors on SpaceSave heuristic. Pay attention to the difference in magnitude.

Figure 2: Boxplots of all heuristics. Due to the difference in magitude in between the SpaceSave heuristic and the other heuristics we separated them into two subplots.

correctly, whereas the strength of CSS-MinHash comes out when dealing with larger strings. We also have to point out that the times for the SpaceSave heuristic are not optimal, since we do not use the proposed data structure for the sketches (Metwally et al., 2005). This does however not influence the errors presented. We found that its performance is due to the employed consistency check. We discuss this in detail in Appendix A and do not test it further, since the nature of their sketches does not enable undoing.



(*a*) Comparison of the stream settings and the new stream setting.



(*b*) CSS-MinHash in batch-mode vs. the new stream-mode. Note that on problem 20 the stream mode had a perplexity score of around 40, but we trimmed the y-axis at a length of 10 for better comparisons.

Figure 3: Perplexity score results of the streaming experiments.

5.2. Streaming

After measuring the base-performance of our Count-Min-Sketch heuristic, we are also interested in the effect of our new streaming method. Having already experimented with different parameters, we decided to move on with Alergia $k = 3$ and CSS-MinHash $F_s = 4$, because these delivered the best results respectively. Note again that both look four steps into the future per state. To compare our streaming approach we compare it with the traditional streaming approach that does not undo and redo refinements, which we call *streaming old*. We depict the perplexities of the streaming approaches in Fig. 3(*a*). Because we also want to see how the streaming compares with the batch-mode version, we picked the best streaming version, i.e. the CSS-MinHash with $F_S = 4$, and compared it with its respective batch-mode version in Fig. 3(*b*).

Strategy	Prob. 1	Prob. 5	Prob. 11	Prob. 18	Prob. 23	Prob. 34	Prob. 45
Batch	250MB	56MB	1.1GB	3.1GB	1.0GB	2.4GB	257MB
Stream old	15MB	13MB	34MB	123MB	63MB	53MB	13MB
Stream new	14MB	13MB	29MB	123MB	63MB	44MB	13MB

Table 2: Memory-footprints compared on a few selected problems. Heuristic employed: CSS-MinHash $F_s == 4$.

The run-times are as follows: Alergia in the old stream setting took 37s, while CSS took 67s, and on the new stream setting they took 51s (Alergia) and 78s (CSS). We report the maximum achieved heap size as measured using the Valgrind tool (Nethercote and Seward, 2007) in Table 2. As we can see the streaming did drastically reduce the memory footprint and run-time compared with the batch version. We can also see that the old streaming approach is faster but gives worse results, and that the errors for Alergia are much higher on the old streaming approach, since on the new streaming approach it gets the chance to correct its mistakes. The main advantage of CSS over the k-tails augmented Alergia is then that CSS has more information on the fringes of the prefix tree. We see that in the new streaming CSS still performs better than Alergia on most problems, although the difference has become much smaller than on the old streaming setting.

6. Discussion and conclusion

We propose a new strategy for streaming state machine learning using Count-Min-Sketches and the ability to undo previous mistakes. We demonstrate the efficacy of our approach on the PAutomaC dataset. We found that on this dataset the lookahead improvements start saturating at about 2 to 3 steps ahead into the future of each state respectively. We then showed that CSS and the variant CSS-MinHash perform well. We compared with the SpaceSave heuristic and point its subpar performance to the consistency heuristic, as discussed in Appendix A, and we point out the limitation that it does not support the undo-operation. Lastly we compared CSS with Alergia and showed similar performance when we enhance Alergia with the k-tails algorithm. An advantage of CSS here is that Alergia does require the prefix tree to be complete. After that we showed the effectiveness of our new streaming strategy in comparison with the traditional approach. It is clear that the new streaming strategy greatly improved results. It also showed the effectiveness of our CSS heuristic, as can especially be seen in the comparison of Alergia and CSS in the old streaming scenario. This discrepancy does however become smaller in the new streaming strategy. We expect our heuristic to make better choices mainly on the fringes of an incomplete prefix tree. Both streaming strategies greatly improved run-time and memory footprint compared with the batched version. Limitations of our work are the limited size of the dataset, and limitations to our hashing using MinHash, which mitigates but does not yet solve the potential run-time bottleneck completely.

Acknowledgments

This work is supported by NWO TTW VIDI project 17541 - Learning state machines from infrequent software traces (LIMIT).

References

John Abela, François Coste, and Sandro Spina. Mutually compatible and incompatible merges for the search of the smallest consistent dfa. volume 3264, pages 28–39, 10 2004. ISBN 978-3-540-23410-4. doi: 10.1007/978-3-540-30195-0_4.

Hasan Ibne Akram, Alban Batard, Colin De La Higuera, and Claudia Eckert. Psma: A parallel algorithm for learning regular languages. In *NIPS workshop on learning on cores, clusters and clouds*. Citeseer, 2010.

Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, nov 1987. ISSN 0890-5401. doi: 10.1016/0890-5401(87)90052-6. URL https://doi.org/10.1016/0890-5401(87)90052-6.

Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X. URL http://www.amazon.com/Principles-Model-Checking-Christel-Baier/dp/026202649X%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D026202649X.

Borja Balle, Jorge Castro, and Ricard Gavaldà. Bootstrapping and learning pdfa in data streams. In Jeffrey Heinz, Colin Higuera, and Tim Oates, editors, *Proceedings of the Eleventh International Conference on Grammatical Inference*, volume 21 of *Proceedings of Machine Learning Research*, pages 34–48, University of Maryland, College Park, MD, USA, 05–08 Sep 2012. PMLR. URL http://proceedings.mlr.press/v21/balle12a.html.

Borja Balle, Jorge Castro, and Ricard Gavaldà. Adaptively learning probabilistic deterministic automata from data streams. *Mach Learn*, 96:99–127, 2014. doi: 10.1007/s10994-013-5408-x.

A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972. doi: 10.1109/TC.1972.5009015.

A.Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997. doi: 10.1109/SEQUEN.1997.666900.

Danny Sleator Carl Kingsford. 15-451/651: Algorithms, lecture #8: Streaming algorithms, September 2016.

Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications*, pages 139–152, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48985-6.

Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. ISSN 0196-6774. doi: https://doi.org/10.1016/j.jalgor.2003.12.001. URL https://www.sciencedirect.com/science/article/pii/S0196677403001913.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138857. doi: 10.1145/997817.997857. URL https://doi-org.tudelft.idm.oclc.org/10.1145/997817.997857.

Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2005.01.003. URL https://www.sciencedirect.com/science/article/pii/S0031320305000221. Grammatical Inference.

Pierre Dupont. Incremental regular inference. In Laurent Miclet and Colin de la Higuera, editors, *Grammatical Interference: Learning Syntax from Sentences*, pages 222–237, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70678-6.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606157.

E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(78)90562-4. URL https://www.sciencedirect.com/science/article/pii/S0019995878905624.

Christian Hammerschmidt, Sicco Verwer, Qin Lin, and Radu State. Interpreting finite automata for sequential data. 12 2016.

Kevin J Lang. Faster algorithms for finding minimal consistent dfas. *NEC Research Institute, Tech. Rep*, 1999.

Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1433, pages 1–12. Springer Verlag, 1998. ISBN 3540647767. doi: 10.1007/bfb0054059.

Qin Lin, Yihuan Zhang, Sicco Verwer, and Jun Wang. Moha: A multi-mode hybrid automaton model for learning car-following behaviors. *IEEE Transactions on Intelligent Transportation Systems*, 20(2):790–796, 2018.

Chen Luo, Fei He, and Carlo Ghezzi. Inferring software behavioral models with mapreduce. *Science of Computer Programming*, 145:13–36, 2017.

Leonardo Mariani, Mauro Pezzè, and Mauro Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, 43(8):715–738, 2017. doi: 10.1109/TSE.2016.2623623.

Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, page 398–412, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540242880. doi: 10.1007/978-3-540-30570-5_27. URL https://doi.org/10.1007/978-3-540-30570-5_27.

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL https://doi.org/10.1145/1273442.1250746.

A.L. Oliveira and J.P.M. Silva. Efficient search techniques for the inference of minimum size finite automata. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207)*, pages 81–89, 1998. doi: 10.1109/SPIRE.1998.712986.

J. Oncina and P. García. *Inferring regular languages in polynomial updated time*, pages 49–61. 1992. doi: 10.1142/9789812797902_0004. URL https://www.worldscientific.com/doi/abs/10.1142/9789812797902_0004.

Gaetano Pellegrino, Qin Lin, Christian Hammerschmidt, and Sicco Verwer. Learning behavioral fingerprints from Netflows using Timed Automata. In *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pages 308–316. Institute of Electrical and Electronics Engineers Inc., jul 2017. doi: 10.23919/INM.2017.7987293.

Jana Schmidt and Stefan Kramer. Online induction of probabilistic real-time automata. *Journal of Computer Science and Technology*, 29(3):345–360, 2014.

Hans Schouten. Learning State Machines from data streams and an application in network-based threat detection. Technical report, 2018. URL http://repository.tudelft.nl/.

Donghwan Shin, Domenico Bianculli, and Lionel Briand. Prins: Scalable model inference for component-based system logs. *arXiv preprint arXiv:2106.01987*, 2021.

Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

Sicco Verwer. *Efficient Identification of Timed Automata: Theory and practice*. PhD thesis, 2010.

Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017. doi: 10.1109/ICSME.2017.58.

Sicco Verwer and Christopher Hammerschmidt. Flexfringe. https://github.com/tudelft-cda-lab/FlexFringe.

Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, 86:295–333, 03 2012. doi: 10.1007/s10994-011-5265-4.

Sicco Verwer, Rémi Eyraud, and Colin De La Higuera. PAutomaC: A probabilistic automata and hidden Markov models learning competition. *Machine Learning*, 96(1-2):129–154, oct 2014. ISSN 15730565. doi: 10.1007/s10994-013-5409-9. URL http://ai.cs.umbc.edu/icgi2012/challenge/Pautomac/committee.php.

Asmir Vodenčarević, Hans Kleine Bürring, Oliver Niggemann, and Alexander Maier. Identifying behavior models for process plants. In *ETFA2011*, pages 1–8, 2011. doi: 10.1109/ETFA.2011.6059080.

Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016. ISSN 1573-7616. doi: 10.1007/s10664-015-9367-7. URL https://doi.org/10.1007/s10664-015-9367-7.

Appendix A. Discussion of SpaceSave

As can be seen in Figure 2, the SpaceSave heuristic did not work all too well for us. The hyperparameters we used can be taken from Table 3. While we might have been able to improve via setting hyperparameters, we found it harder than the other two methods to find appropriate hyperparameters. We point this to the similarity test developed by Balle et al. and described in (Balle et al., 2012). In short, the similarity test assumes a distinguishability parameter μ for a pair of two sketches that is being approximated by value and a confidence interval estimated from sampled sketches. The term for the upper bound is determined by Equation (2). It is clear the the term includes the estimate $\hat{\mu}_k$ plus a confidence interval. The confidence interval itself is the sum of 8ν (in our implementation we decreased it to 2ν because it is a large constant even with relatively small ϵ), with $\nu = 2\epsilon$, a hyperparameter, and a max-term, where both terms within the max-term come in the form of an nth-square-root of $\frac{1}{M} \ln \frac{K_k}{\delta}$, with the pooled mass $M = \frac{m_1 \cdot m_2}{(\sqrt{m_1}+\sqrt{m_2})^2}$ and $K_k \propto (m_1 + m_2)^2$. m_1 and m_2 are the sizes of the two respective sketches, i.e. *the number of strings processed* (Balle et al., 2012). Comparing M and K_k in the limit leads to the following: $\lim_{m_1 \to \infty} K_k = \infty$, but $\lim_{m_1 \to \infty} M = m_2$, and vice versa with $m_2 \to \infty$ due to the symmetry in m_1 and m_2. This means that for imbalanced states, i.e. one that has processed many strings, while the other has processed few of them in relative terms, the denominator becomes smaller and the nominator becomes larger within the max-term, i.e. the confidence bound becomes larger, even though both states might have gathered enough evidence to confidently conclude good approximation of their sampled distributions.

We show this problem with two tests that actually occurred in problem 5 of PAutomaC, an example where m_1 and m_2 are rather similar, the balanced example, and one imbalanced example, where the sizes of the states were rather unequal. In the balanced example we

Parameter	$L == 1$	$L == 2$
ϵ	0.005	0.005
δ	0.05	0.05
μ	0.6	0.4
K (number of prefixes to track)	20	20
r (number of bootstrapped sketches)	10	10

Table 3: Hyperparameters used for the SpaceSave heuristic. The parameter's meaning can be taken from (Balle et al., 2012).

had one state with a size of approximately $m_1 \approx 8000$, and $m_2 \approx 11000$. The pooled mass became $M = 4804$. As for the imbalanced example, $m_1 \approx 25000$ and $m_2 \approx 950$, hence $M = 1016$. K_k became very large in both cases, so the log term $ln\frac{K_k}{\delta}$ became ≈ 42 in both cases. In both cases, the max-term was determined by the left side of the max-term in Eq. 2, and the term $\frac{1}{1c_1M}$ became ≈ 0.001 for the balanced state pair, and ≈ 0.005 for the imbalanced state pair, resulting in a max-term of 0.2 for the balanced states, and 0.45 for the imbalanced states. Given that the distinguishability of two states based on the L_∞^p-distance as defined in (Balle et al., 2012) lies in the interval $[0, 1]$, this a large change. It should be noted that in both cases the approximated distinguishability $\hat{\mu}_k$ was very small, and we considered 950 a good enough sample size.

We provided a quick fix for the problems via simply replacing the similarity check with the Hoeffding-bound that is also used in Alergia and CSS, as we show in Figure 4, in which we compare the heuristic vs Alergia with $k == 2$. A better performance might be achieved with some tuning, which we did not do at this point. Because the nature of the sketches does not permit the undo operation, which is required for our merging strategy, we did not proceed with this heuristic from here on. It should be noted that the large runtimes are partly attributable to the data structure we used to model the sketches described in the original paper, since we do not use the data structure described by (Metwally et al., 2005).

$$\mu^* \leq \min_{1 \leq k \leq r^2} \left\{ \hat{\mu}_k + 8\nu + max \left\{ \sqrt{\frac{1}{2c_1M} \ln \frac{K_k}{\delta}}, \sqrt[4]{\frac{(16\nu)^2}{2c_2M} \ln \frac{K_k}{\delta}} \right\} \right\}. \quad (2)$$

A.1. Hyperparameters

In this chapter we give the hyperparameters. They are defined as in our tool Verwer and Hammerschmidt, which is open source and can be downloaded and used. Hence in the following we provide simple lists as can be used by .ini-files by the tool.

A.1.1. Alergia batch-mode

```
[default]
heuristic-name = alergia
data-name = alergia_data
; for use with small datasets use low counts
```

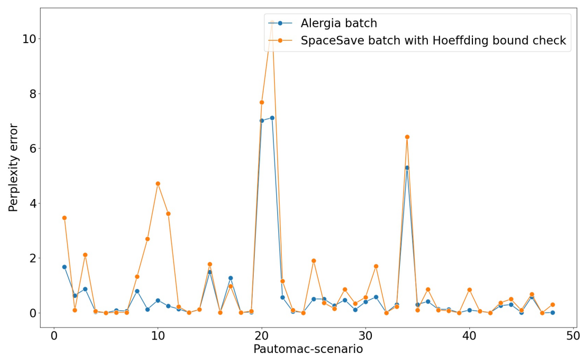


Figure 4: New results of the SpaceSave heuristic with the Hoeffding bound instead of the old check.

```
state_count = 10
symbol_count = -1
satdfabound = 2000
sinkson = 1
blueblue = 0
sinkcount = 25
confidence_bound = 0.5
largestblue = 1
finalred = 0
lowerbound = 0
finalprob = 1
mergelocal = -1
mcollector = 4
markovian = 0
printwhite = 0
printblue = 1
extend = 0

; ktail can be varied to reproduce the experiments
ktail = 2
```

A.1.2. Alergia stream-mode

```
[default]
heuristic-name = alergia
data-name = alergia_data
; for use with small datasets use low counts
mode = stream
```

```
state_count = 10
symbol_count = -1
satdfabound = 2000
sinkson = 1
blueblue = 0

; sinkcount 100 for old streaming strategy, 5 for new streaming strategy sinkcount = 100
confidence_bound = 0.1
largestblue = 1
finalred = 0
lowerbound = 0
finalprob = 1
mergelocal = -1
mcollector = 4
markovian = 0
printwhite = 1
printblue = 1
extend = 0

ktail = 2
mergesinks = 0
extendsinks = 0
; addtails = 0 saves memory
addtails = 0
parentsizethreshold = -1
; redbluethreshold does enable the streaming framework, i.e. appending only to red and
blue states
redbluethreshold = 1
```

A.1.3. CSS batch-mode

```
[default]
heuristic-name = css
data-name = css_data
; for use with small datasets use low counts
mode = batch
state_count = 10
satdfabound = 2000

blueblue = 0
finalred = 0
lowerbound = 0
finalprob = 1
mergelocal = -1
mcollector = 4
```

```
markovian = 0
printwhite = 0
printblue = 1
extend = 0

; sinks
sinkson = 1
sinkcount = 25
mergesinks = 0
extendsinks = 0

; for streaming and LSH
testmerge = 1
largestblue = 1

; general LSH parameters
confidence_bound = 0.5
; numoftables can be 1 when conditionalprobs is turned off. Too little collisions
numoftables = 3
; the width of the sketches
vectordimension = 20
; futuresteps == $F_S$
futuresteps = 3
; conditionalprob = 0 model conditional probabilities, unreported experiments
conditionalprob = 1
; minhash does only work if conditionalprob is turned on
minhash = 1
minhashsize = 2

ktail = 0

; saving space for streaming
addtails = 0
parentsizethreshold = -1
redbluethreshold = 0
```

A.2. CSS stream-mode

```
[default]
heuristic-name = css
data-name = css_data
; for use with small datasets use low counts
mode = stream
state_count = 10
```

```
satdfabound = 2000

blueblue = 0
finalred = 0
lowerbound = 0
finalprob = 1
mergelocal = -1
mcollector = 4
markovian = 0
printwhite = 1
printblue = 1
extend = 0

;sinks
sinkson = 1
; sinkcount 5 for new stream strategy, 100 for old
sinkcount = 100
mergesinks = 0
extendsinks = 0

; for streaming and CSS
testmerge = 1
largestblue = 1

; general CSS parameters
; 0.05 for new streaming strategy, 0.5 for old
confidence_bound = 0.5
; numoftables can be 1 when conditionalprobs is turned off. Too little collisions
numoftables = 3
vectordimension = 20
futuresteps = 4
conditionalprob = 1
; minhash does only work if conditionalprob is turned on
minhash = 1
minhashsize = 2

ktail = 0
addtails = 0
parentsizethreshold = -1
redbluethreshold = 1
```

A.3. SpaceSave

```
[default]
heuristic-name = space_saving
data-name = space_saving_data
mode = stream

state_count = 10
symbol_count = 0
satdfabound = 2000

sinkson = 1
sinkcount = 100
blueblue = 0
largestblue = 1
; finalprob
finalprob = 0
finalred = 0
lowerbound = 0
correction = 0
mergesinks = 0
extend = 0
printblue=1
printwhite=0

; keep. NOTE: testmerge must be 0 with this heuristic
testmerge = 0
ktail = 0
addtails = 0
parentsizethreshold = -1
redbluethreshold = 1

; space saving parameters for L=1, uncomment below
;epsilon= 0.005
;delta = 0.05
;mu = 0.6
;pref_L = 1
;pref_K = 20
;bootstrap_R = 10

; space saving parameters for L=2, uncomment below
epsilon= 0.005
delta = 0.05
mu = 0.4
pref_L = 2
```

```
pref_K = 20
bootstrap_R = 10

; only for hoeffding bound
confidence_bound = 0.1
```