Learning Symbolic Automata

Samuel Drews and Loris D'Antoni

University of Wisconsin–Madison

Abstract. Symbolic automata allow transitions to carry predicates over rich alphabet theories, such as linear arithmetic, and therefore extend classic automata to operate over infinite alphabets, such as the set of rational numbers. Existing automata algorithms rely on the alphabet being finite, and generalizing them to the symbolic setting is not a trivial task. In this paper, we study the foundational problem of learning symbolic automata. We first present Λ^*, a symbolic automata extension of Angluin's L^* algorithm for learning regular languages. Then, we define notions of learnability that are parametric in the alphabet theories of the symbolic automata and show how these notions nicely compose. Specifically, we show that if two alphabet theories are *learnable*, then the theory accepting the Cartesian product or disjoint union of their alphabets is also learnable. Using these properties, we show how existing algorithms for learning automata over large alphabets nicely fall in our framework. Finally, we implement our algorithm in an open-source library and evaluate it on existing automata learning benchmarks.

1 Introduction

Finite automata are a ubiquitous formalism that is simple enough to model many real-life systems and phenomena, and they enjoy a large variety of theoretical properties: automata are closed under Boolean operations, have decidable emptiness and equivalence checking procedures, and *can be learned* [3]. This last problem on automata learning is the focus of this paper; learning has been studied extensively for several variations of finite automata [7, 4] and has found many applications in program verification [2], program synthesis [14].

Unfortunately, finite automata have an inherent limitation: they can only operate over finite (and typically small) alphabets. Symbolic finite automata (s-FA) allow transitions to carry predicates over rich alphabet theories, such as linear arithmetic, and therefore extend classic automata to operate over infinite alphabets, such as the set of rational numbers. Existing automata algorithms rely on the alphabet being finite, and generalizing them to the symbolic setting is not a trivial task. However, algorithms have been proposed for s-FA equivalence, for minimization, and for performing Boolean operations. In this paper, we study the foundational problem of learning symbolic automata.

We start by extending Angluin's L^* algorithm for learning regular languages to symbolic automata. L^* iteratively updates a table of evidence, conjectures an automaton, and then if that conjecture is not correct, repeats with new evidence.

Our algorithm, Λ^*, operates in a largely similar manner, except that there is an additional stage after updating the table of evidence during which the evidence is *generalized* into predicates; these predicates form the transitions for the symbolic automaton.

We then define notions of learnability that are parametric in the alphabet theories of the symbolic automata and show how these notions nicely compose. For example, if two alphabet theories are *learnable*, then the theory accepting the Cartesian product of their alphabets is also learnable. We use these properties to show how existing algorithms for learning automata over large alphabets nicely fall in our framework: e.g., Maler and Mens present an ad hoc method for learning automata whose alphabet consists of elements in a subset of $\mathbb{Z} \times \mathbb{Z}$ [12], which we show is learnable because it is the Cartesian product of subsets of $\mathbb{Z}$—which itself is learnable.

Finally, we implement our algorithm in an open-source symbolic automata library and evaluate it on existing automata learning benchmarks from [12]. The implementation is truly modular and only requires the programmer to provide learnable Boolean algebras as input to the learner; the disjoint union and product algebras are implemented meta-algebras that can be instantiated arbitrarily. Our implementation, despite its generality and ability to support arbitrary alphabet theories, can learn all the benchmarks appearing [12] using a similar number of equivalence and membership queries.

In summary, our contributions are:

- An algorithm for learning Symbolic Finite Automata together with a proof of soundness (§ 3).
- A notion of learnability parametric in the underlying alphabet theory that composes over the operations of Cartesian product and disjoint union of algebras (§ 4).
- A modular implementation of our algorithm in an existing open-source library and an evaluation on existing benchmarks (§ 5).

2 Preliminaries

Symbolic automata have alphabets whose characters represent multiple values. These alphabets must respect certain properties; the necessary and sufficient properties are exactly those of a Boolean algebra.

Definition 1 (Effective Boolean Algebra). *An* effective Boolean algebra $\mathcal{A}$ *is a tuple* $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ *where*

- $\mathfrak{D}$ *is a set of* domain elements
- Ψ *is a set of* predicates *closed under the Boolean connectives, with* $\bot, \top \in \Psi$
- $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ *is a* denotation function *such that* $[\![\bot]\!] = \emptyset$*;* $[\![\top]\!] = \mathfrak{D}$*; and for all* $\varphi, \psi \in \Psi$*,* $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$*,* $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$*, and* $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$

Since Ψ is closed with respect to $\vee$, $\wedge$, and $\neg$, we can *generate* Ψ from an initial set Ψ_0 as follows: for all $i > 0$,

$$\begin{aligned}\Psi_i = \Psi_{i-1} &\cup \{\neg\varphi \mid \varphi \in \Psi_{i-1}\} \\ &\cup \{\varphi_1 \vee \varphi_2 \mid \varphi_1, \varphi_2 \in \Psi_{i-1}\} \\ &\cup \{\varphi_1 \wedge \varphi_2 \mid \varphi_1, \varphi_2 \in \Psi_{i-1}\}\end{aligned}$$

And finally, $\Psi = \bigcup_i \Psi_i$ for all $i \geq 0$. Consider the following examples of Boolean algebras:

Example 1 (Equality Algebra). Let c be a fixed variable. The *equality algebra* for an arbitrary set $\mathfrak{D}$ has predicates formed from Boolean combinations of formulas of the form $c = a$ where $a \in \mathfrak{D}$. Formally, Ψ is generated from $\Psi_0 = \{\varphi_a \mid a \in \mathfrak{D}\} \cup \{\bot, \top\}$ where for all $a \in \mathfrak{D}$, $[\![\varphi_a]\!] = \{a\}$. Example predicates in this algebra include the predicates $c = 5 \vee c = 10$ and $\neg(c = 0)$.

Example 2 (Interval Algebra). The finite union of left-closed right-open intervals over non-negative integers (i.e. $\mathbb{N}$) also forms a Boolean algebra: let $\Psi_0 = \{\varphi_{ij} \mid i, j \in \mathbb{N} \wedge i < j\} \cup \{\bot, \top\}$ where $[\![\varphi_{ij}]\!] = [i, j)$. Example predicates in this algebra include those of the form $[0, 5) \cup [10, 15)$ or $[50, \infty)$.

Now we are ready to present the formalism of symbolic finite automata.

Definition 2 (Symbolic Finite Automata). *A* symbolic finite automaton *(s-FA) M is a tuple $(\mathcal{A}, Q, q_{init}, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the* alphabet*; Q is a finite set of states; $q_{init} \in Q$ is the* initial state*; $F \subseteq Q$ is the set of* final states*; and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is the* transition relation *consisting of a finite set of* moves *or* transitions*.*

Characters are elements of $\mathfrak{D}_{\mathcal{A}}$, and *words* are finite sequences of characters, or elements of $\mathfrak{D}^*_{\mathcal{A}}$. The empty word of length 0 is denoted by ϵ.

A move $\rho = (q_1, \varphi, q_2) \in \Delta$, also denoted $q_1 \xrightarrow{\varphi} q_2$, is a transition from the *source* state q_1 to the *target* state q_2, where φ is the *guard* or *predicate* of the move. A move is *feasible* if its guard is satisfiable. For a character $a \in \mathfrak{D}_{\mathcal{A}}$, an *$a$-move* of M, denoted $q_1 \xrightarrow{a} q_2$ is a move $q_1 \xrightarrow{\varphi} q_2$ such that $a \in [\![\varphi]\!]$.

An s-FA M is *deterministic* if, for all transitions $(q, \varphi_1, q_1), (q, \varphi_2, q_2) \in \Delta$, $q_1 \neq q_2 \rightarrow [\![\varphi_1 \wedge \varphi_2]\!] = \emptyset$; i.e., for each state q and character a there is at most one a-move out of qq. An s-FA M is *complete* if, for all $q \in Q$, $\bigvee_{(q,\varphi_i,q_i)\in\Delta} \varphi_i = \top$; i.e., for each state q and character a there exists an a-move out of q. Throughout the paper we will assume all s-FAs are deterministic and complete, since determinization and completion are always possible. An example s-FA is $\mathbf{M_{11}}$ in Figure 1. This s-FA has 4 states and it operates over the interval algebra from Example 2.

Given an s-FA $M = (\mathcal{A}, Q, q_{\text{init}}, F, \Delta)$ and a state $q \in Q$, we say a word $w = a_1 a_2 \ldots a_k$ is *accepted at state q* if for $1 \leq i \leq k$ there exist moves $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q$ and $q_k \in F$. The *language accepted at q* is denoted by $\mathcal{L}_q(M)$, and we denote the *language accepted by M* as $\mathcal{L}(M) = \mathcal{L}_{q_{\text{init}}}(M)$. For example, the

s-FA $\mathbf{M_{11}}$ in Figure 1 accepts, among others, words consisting only of numbers accepted by the predicate $[0, 51) \cup [101, \infty)$ and rejects, among others, the word consisting of the two symbols $51, 25$.

3 Learning Algorithm

Here we present outralgorithm, Λ^*, which learns symbolic automata. The premise is that the automaton to be learned, called the *target*, is hidden in a black box, so knowledge of it comes from some *oracle* that admits two kinds of queries: *membership queries* that ask whether a word is in the language of the target, and *equivalence queries* that ask whether a conjectured automaton is equivalent to the target—if not, a counterexample is provided in the form of a word that is classified differently by the two automata. Λ^*, which builds upon L^* [3], maintains a data structure called the *obversation table* that comprises its knowledge about the target. The observation table is used to build the intermediary guesses of the target automaton and, eventually, the final automaton.

3.1 Observation Table

The observation table consists of rows of prefixes and columns of suffixes; thus each entry holds the information about whether the target automaton accepts the word formed by concatenating the prefix and suffix. The idea is that the prefixes provide knowledge about words that lead to states, and the suffixes help differentiate those states.

Definition 3 (Observation Table). *An* observation table T *for an s-FA* M *is a tuple* (Σ, S, R, E, f) *where*

- Σ *is a potentially infinite set called the* alphabet
- $S, R, E \subset \Sigma^*$ *are finite subsets of words;* S *is called the set of* prefixes*,* R *is called the* boundary*, and* E *is called the set of* suffixes
- $f : (S \cup R) \cdot E \to \{0, 1\}$ *is a* classification function *such that for a word* $w \cdot e \in (S \cup R) \cdot E$*,* $f(w \cdot e) = 1$ *if* $w \cdot e \in \mathcal{L}(M)$*, and* $f(w \cdot e) = 0$ *if* $w \cdot e \notin \mathcal{L}(M)$ [1]

and the following properties hold:

- S *and* R *are disjoint*
- $S \cup R$ *is prefix-closed and* $\epsilon \in S$
- *for all* $s \in S$*, there exists a character* $a \in \Sigma$ *such that* $s \cdot a \in R$
- $\epsilon \in E$*.*

[1] We use $\cdot$ to denote both the concatenation of strings and its lifting to sets of strings, as is standard.

Table $\mathbf{T_1}$ in Figure 1 is an example observation table, where the elements in S are shown above the horizontal divider and the elements in R below.

The observation table induces the construction of an automaton. For intuition, each $s \in S$ corresponds to a state q such that s is a string of moves from q_{init} to q. The boundary R gives information about the transitions between states. The states are differentiated by the strings in E and the classification function f, as if there exist $s_1, s_2 \in S$ and $e \in E$ such that $f(s_1 \cdot e) \neq f(s_2 \cdot e)$, then s_1 and s_2 *behave* differently and must lead to different states. The table can be visualized as having rows consisting of the elements in $S \cup R$ and as having columns consisting of the elements in E. We use the notation $row(w)$ for $w \in S \cup R$ to denote the vector indexed by $e \in E$ of $f(w \cdot e)$.

Λ^* manipulates the observation table and eventually conjectures an s-FA. An overview of this process is that the table must first satisfy certain properties—we call such a table *cohesive*—that are established through membership queries to the oracle. The cohesive observation table is used to construct an intermediary automaton that is ultimately used to produce a conjectured s-FA.

An observation table is *closed* if for each $r \in R$ there exists $s \in S$ such that $row(s) = row(r)$; in other words, each element in the boundary corresponds to a state. An observation table is *reduced* if for all $s_1, s_2 \in S$, $row(s_1) \neq row(s_2)$, meaning each state is uniquely characterized by f and E. An observation table is *consistent* if for all $w_1, w_2 \in S \cup R$, if $a \in \Sigma^*$ and $w_1 \cdot a, w_2 \cdot a \in S \cup R$ and $row(w_1) = row(w_2)$, then $row(w_1 \cdot a) = row(w_2 \cdot a)$. A table being consistent means that if the words w_1 and w_2 are equivalent according to f and E, then $w_1 \cdot a$ and $w_2 \cdot a$ ought to be equivalent as well, and thus there is no evidence to the contrary.[2] An observation table is *evidence-closed* if for all $e \in E$ and for all $s \in S$, $s \cdot e \in S \cup R$. We say an observation table is *cohesive* if it is closed, reduced, consistent, and evidence-closed.

Consider, for example, the observation tables in Figure 1. $\mathbf{T_2}$ is not closed, since $row(51) = -$ and there is no $s \in S$ with $row(s) = -$. Table $\mathbf{T_5}$ is not consistent because $row(51) = - = row(51, 0)$, but $row(51 \cdot 0) = - \neq + = row(51, 0 \cdot 0)$. Table $\mathbf{T_{11}}$ is closed, reduced, consistent, and evidence-closed.

If an observation table is cohesive, then it admits the construction of an *evidence automaton* that classifies words $w \in \Sigma^*$ equivalently to the observation table's classification function f.

Definition 4 (Evidence Automaton). *An evidence automaton is a tuple* $(\Sigma, Q, q_{init}, F, \Delta)$ *where* Σ *is a set;* Q *is a finite set of states;* $q_{init} \in Q$ *is the* initial state*;* $F \subseteq Q$ *is the set of* final states*;* $\Delta \subseteq Q \times \Sigma \times Q$ *is the* transition relation.

A move $\rho = (q_1, a, q_2)$, also denoted $q_1 \xrightarrow{a} q_2$, is a transition from q_1 to q_2 using the character a. A word $w = a_1 a_2 \dots a_k$ is *accepted at state* q if for $1 \leq i \leq k$ there exist moves $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q$ and $q_k \in F$. Conversely, if that

[2] We use $a \in \Sigma^*$ for the definition of *consistent*, but since the table is prefix-closed by definition, it is equivalent to consider only single-characters $a \in \Sigma$.

$q_k \notin F$, then w is *not* accepted at q. If there is no path through the automaton for w, then the acceptance is undefined.

An evidence automaton differs from an s-FA in that the transitions are characters in Σ instead of predicates in a Boolean algebra over the domain Σ. Additionally, the evidence automaton can be deliberately sparse: it is not *complete*, and we avoid the notion that a state q does not accept a character a if there is no q' such that $(q, a, q') \in \Delta$—as stated above, such a case simply indicates the behavior of a at q is undefined.

Given a cohesive observation table $T = (\Sigma, S, R, E, f)$, we build an evidence automaton $A = (\Sigma, Q, q_{\text{init}}, F, \Delta)$ over that same alphabet as follows: for each $s \in S$, we introduce a state $q_s \in Q$, and q_{init} is assigned to q_ϵ. The final states F are all q_s for $s \in S$ such that $f(s) = 1$. Since the observation table is closed and reduced, there exists a well-defined function $g : S \cup R \to S$ such that $g(w) = s$ if and only if $row(w) = row(s)$. This function allows us to define the transition relation of A: if $w \cdot a \in S \cup R$, then $(q_{g(w)}, a, q_{g(w \cdot a)}) \in \Delta$. In Figure 1, the automaton $\mathbf{M_1^e}$ (resp $\mathbf{M_{11}^e}$) is the evidence automaton corresponding to cohesive table $\mathbf{T_1}$ (resp. $\mathbf{T_{11}}$).

Theorem 1 (Evidence compatibility). *Given a cohesive observation table $T = (\Sigma, S, R, E, f)$, if $M_{evid} = (\Sigma, Q, q_{init}, F, \Delta)$ is the evidence automaton construction of T, then for all $w \cdot e \in (S \cup R) \cdot E$, if $f(w \cdot e) = 1$ then M_{evid} accepts $w \cdot e$, and if $f(w \cdot e) = 0$ then M_{evid} does not accept $w \cdot e$.*

3.2 Separating Predicates

Given an evidence automaton with an alphabet Σ, we require two pieces to build an s-FA:

- a Boolean algebra $\mathcal{A}$ with $\mathfrak{D}_\mathcal{A} = \Sigma$
- a partitioning function P for $\mathcal{A}$, which we define below.

This latter component, the partitioning function, is the key insight to Λ^*'s generalization of L^*.

Definition 5 (Partitioning function). *A* partitioning function *for a Boolean algebra $\mathcal{A} = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ is a function $P : (2^\mathfrak{D})^* \to \Psi^*$ that takes as input a list $L_\mathfrak{D} = \ell_1 \dots \ell_k$ of disjoint sets of elements in $\mathfrak{D}$, and returns a parallel list $L_\Psi = \varphi_1 \dots \varphi_k$ of predicates in Ψ such that*

- *for all $a \in \mathfrak{D}$, $a \in [\![\varphi_i]\!]$ for some $\varphi_i \in L_\Psi$*
- *$\bigvee_{\varphi_i \in L_\Psi} \varphi_i = \top$*
- *$\varphi_i \wedge \varphi_j = \bot$ for all $\varphi_i, \varphi_j \in L_\Psi$ with $i \neq j$*
- *for each $\ell_i \in L_\mathfrak{D}$ corresponding to $\varphi_i \in L_\Psi$, all $a \in \ell_i$ are such that $a \in [\![\varphi_i]\!]$.*

Example 3 (Equality Algebra Separating Predicates). We can construct a partitioning function for the equality algebra: given a list $L_\mathfrak{D} = \ell_1 \dots \ell_k$ we construct a list $L_\Psi = \varphi_1 \dots \varphi_k$ where each φ_i has $[\![\varphi_i]\!] = \ell_i$. We choose a j with maximal $|\ell_j|$ and update $\varphi_j \leftarrow \varphi_j \vee \bigwedge_{1 \leq i \leq k} \neg \varphi_i$. In the concrete case of $\mathfrak{D} = \mathbb{Z}$ and $L_\mathfrak{D} = [\{2\}, \{3\}, \emptyset, \{0, 5\}]$, the partitioning function would produce (after simplification) $L_\Psi = [a = 2, a = 3, \bot, a \neq 2 \wedge a \neq 3]$.

At a high level, as long as the s-FA is consistent with the evidence automaton, it will be consistent with the observation table. The words in the remainder of Σ^*—for which the evidence automaton has unspecified classification—can be assigned to paths in a largely arbitrary manner. The partitioning function handles generalizing the concrete evidence by creating *separating predicates*, in effect specifying the behavior for the remaining words. Ideally, this generalization allows an automaton to be learned with a relatively small observation table, even if the alphabet is large—or even infinite.

Given an evidence automaton $A = (\Sigma, Q, q_{\text{init}}, F, \Delta)$, a Boolean algebra $\mathcal{A}$ with domain Σ, and an appropriate partitioning function P, we build an s-FA $M = (\mathcal{A}, Q, q_{\text{init}}, F, \Delta_M)$ using that Boolean algebra and that exact configuration of states. All that remains is the construction of the transition relation Δ_M.

For each state $q \in Q$, we perform the following: gather all evidence transitions out of that state into a set $\Delta_q = \{(q, a, q') \in \Delta\}$ and then construct a list L_Σ indexed over the states $q_i \in Q$, where each set in the list is $\{a \mid (q, a, q_i) \in \Delta_q\}$. We apply the partitioning function to get a list of separating predicates $L_{\Psi_\mathcal{A}} = P(L_\Sigma)$ which is also indexed over $q_i \in Q$, and then add (q, φ_i, q_i) to Δ_M for each $\varphi_i \in L_{\Psi_\mathcal{A}}$.

Theorem 2 (s-FA evidence compatibility). *Given a cohesive observation table $T = (\Sigma, S, R, E, F)$, if $M_{evid} = (\Sigma, Q, q_{init}, F, \Delta)$ is the evidence automaton construction of T, and $M = (\mathcal{A}, Q, q_{init}, F, \Delta)$ is produced from M_{evid} using a partitioning function, then for all $w \cdot e \in (S \cup R) \cdot E$, if $f(w \cdot e) = 1$ then $w \cdot e \in \mathcal{L}(M)$, and if $f(w \cdot e) = 0$ then $w \cdot e \notin \mathcal{L}(M)$.*

An example observation table, its corresponding evidence automaton, and a resultant s-FA are shown in the last row of Figure 1.

3.3 Algorithm Description

L: TODO flowchart showing L^* versus our algorithm: loop table, (intermediary), s-FA, counterexample.

The algorithm begins by initializing an observation table with $S = \{\epsilon\}$, $R = \{a\}$ for an arbitrary $a \in \Sigma$, and $E = \{\epsilon\}$. f is initially undefined. The knowledge of the table is grown using the operations *fill*, *close*, *evidence-close*, and *make-consistent*.

The operation *fill* asks a membership query for all $w \cdot e \in (S \cup R) \cdot E$ for which f is undefined and then adds those results to f; in this way, it ensures f is defined over the entire domain of the observation table.

The operation *close* checks for the existence of an $r \in R$ such that for all $s \in S$, $row(r) \neq row(s)$. Such an r is moved from R to S, and $r \cdot a$ is added to R for some arbitrary $a \in \Sigma$.

The operation *evidence-close* ensures for all $e \in E$ and $s \in S$ that $s \cdot e \in S \cup R$ by adding to R all $s \cdot e$ that are not. It also adds to R any necessary prefixes so that $S \cup R$ is prefix-closed.

The operation *make-consistent* operates as follows: if there exist words $w_1, w_2 \in S \cup R$ and $w_1 \cdot a, w_2 \cdot a \in S \cup R$ for some $a \in \Sigma$ such that $row(w_1) = row(w_2)$ but

Algorithm 1 Learning algorithm

```
Initialize table T
repeat
  while T is not cohesive do
    make-consistent T
    evidence-close T
    close T
  end while
  construct evidence automaton M_evid from T
  construct s-FA M from M_evid
  if M is not equivalent to the target automaton then
    process counterexample
  end if
until equivalent
```

$row(w_1 \cdot a) \neq row(w_2 \cdot a)$, then w_1 and w_2 actually lead to different states; using the $e \in E$ such that $f(w_1 \cdot a \cdot e) \neq f(w_2 \cdot a \cdot e)$, it is clear $a \cdot e$ thus differentiates those states. Accordingly, $a \cdot e$ is added to E.

Upon receiving a counterexample $c \in \Sigma^*$ from an equivalence query sent to the oracle, all prefixes of c are added to R (except those already present in S). There are two cases for a counterexample: one of the predicates in the s-FA needs refinement, which is facilitated by adding those new words to the table, or a new state must exist in the automaton, which is handled by the operation *make-consistent*.

Algorithm 1 shows the learning algorithm: after the table is initialized, the operations *make-consistent*, *evidence-close*, and *close* are applied until the table is cohesive.[3] (*Fill* is applied throughout whenever a change is made to the table.) An s-FA M is then conjectured from the table, and an equivalence query is performed: if M is equivalent to the target automaton, then the algorithm terminates. Otherwise, a counterexample is produced and processed, and the procedure repeats.

3.4 Worked Example

Suppose we invoke Λ^* to learn the automaton over non-negative integers that accepts all words *except* those that contain a number between 51 and 100 that is not immediately followed by two numbers between 0 and 20.

The Boolean algebra we use is the union of left-closed right-open intervals. We fix a partitioning function P that assumes that if $a \in \ell$, $b \in \ell'$, and there are no c in the input sets such that $a < c < b$, then the whole interval $[a, b)$ behaves equivalently to a (this is a great assumption if the oracle is always guaranteed to provide lexicographically minimal counterexamples, as we will discuss later).

[3] It is an invariant of the initialization of the table and of the operations applied to it that the observation table is always reduced.

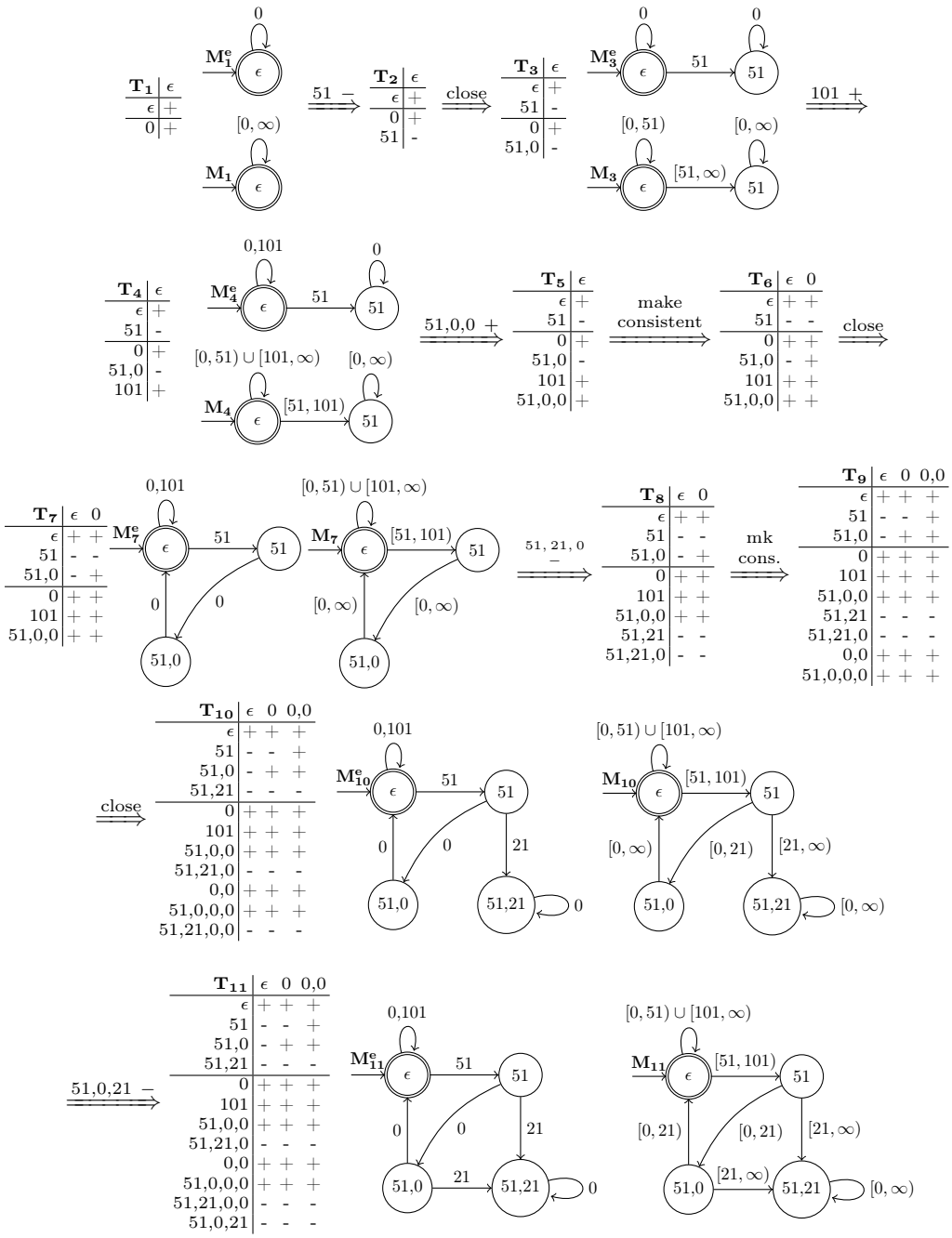

Fig. 1. An example run of the Λ^* algorithm.

For example, $P(\{0\},\{10\}) = [0,10),[10,\infty)$ and $P(\{0,20\},\{10\}) = [0,10) \cup [20,\infty),[10,\infty)$.

The trace of the algorithm is illustrated in Figure 1. Λ^* begins by initializing an observation table so that $S = \{\epsilon\}$, $R = \{0\}$ (the 0 is an arbitrary character from Σ and is used purely so that the table contains $\epsilon \cdot a$ for some a), and $E = \{\epsilon\}$. The appropriate membership queries are made to the oracle, resulting in the table $\mathbf{T_1}$. $\mathbf{T_1}$ is cohesive, so it is used to construct the evidence automaton $\mathbf{M_1^e}$, and by calling the partitioning function P on the outgoing transitions of each state in $\mathbf{M_1^e}$—in this case just $P(\{0\}) = [0,\infty)$—the s-FA $\mathbf{M_1}$ is conjectured. The oracle is given $\mathbf{M_1}$ as an equivalence query, and it returns the single-character word 51 as a counterexample. 51 is added to R in the observation table, as would all of its prefixes if it were a word of length greater than one, and a membership query is asked for $51 \cdot \epsilon$, resulting in table $\mathbf{T_2}$.

$\mathbf{T_2}$ is not closed, since $row(51) = -$ and there is no $s \in S$ with $row(s) = -$. Accordingly, 51 represents a path to a new state, so it is moved from S to R, and a continuation $51, 0$ is added to R. This produces table $\mathbf{T_3}$, which is now cohesive and thus admits the construction of the evidence automaton $\mathbf{M_3^e}$ and ultimately the s-FA $\mathbf{M_3}$ through the use of the partitioning function: for example, for the outgoing transitions of the initial state, $P(\{0\},\{51\}) = [0,51),[51,\infty)$. An equivalence query sent to the oracle returns the counterexample of 101.

Adding 101 to R results in the cohesive table $\mathbf{T_4}$ and the s-FA $\mathbf{M_4}$, and the oracle provides the counterexample $51, 0, 0$. $51, 0, 0$ is added to R (all of its prefixes are already present in $S \cup R$), resulting in the table $\mathbf{T_5}$ which is not consistent: observe that $row(51) = - = row(51, 0)$, but $row(51 \cdot 0) = - \neq + = row(51, 0 \cdot 0)$. This means that 51 and $51, 0$ actually lead to different states, which will be addressed in two stages. First, following the rule *make-consistent*, since $f(51 \cdot 0 \cdot \epsilon) \neq f(51, 0 \cdot 0 \cdot \epsilon)$, we add $0 \cdot \epsilon$ to E to distinguish the states led to by 51 and $51, 0$, which produces table $\mathbf{T_6}$. Applying *close* to $\mathbf{T_6}$ results in $\mathbf{T_7}$, which is then cohesive (we added an element to E, which would normally require applying *evidence-close*, but it happens to be that $\mathbf{T_7}$ is already evidence-closed) and produces an s-FA $\mathbf{M_7}$. The counterexample $51, 21, 0$ requires adding it as well as the prefix $51, 21$ to R, producing table $\mathbf{T_8}$.

$\mathbf{T_8}$ is also inconsistent, since $row(51) = -,- = row(51, 21)$ but $row(51 \cdot 0) = -,+ \neq -,- = row(51, 21 \cdot 0)$. Since $f(51 \cdot 0 \cdot 0) \neq f(51, 21 \cdot 0 \cdot 0)$, we add $0 \cdot 0$ to E to distinguish 51 from $51, 21$, and evidence-close the table to get $\mathbf{T_9}$. Closing and evidence-closing this table results in $\mathbf{T_{10}}$, the conjecture $\mathbf{M_{10}}$, the counterexample $51, 0, 21$, the table $\mathbf{T_{11}}$, and finally the automaton $\mathbf{M_{11}}$ which passes the equivalence query.

4 Learnability

Whether an s-FA can be learned and, if so, the complexity of learning are dependent on a more fundamental property concerning the *learnability* of the underlying Boolean algebra. We first first provide a definition for an algebra's learnabil-

ity with respect to the inputs given to a partitioning function and then connect these inputs to the queries given to the oracle during the learning algorithm.

Fix a partitioning function P over a Boolean algebra $\mathcal{A}$. Let C denote a concept class for which each concept $c \in C$ is a finite partition of $\mathfrak{D}_{\mathcal{A}}$ using predicates $\Psi_{\mathcal{A}}$, and let G denote the set of *generators* which, informally, provide a sequence of counterexamples—elements in $\mathfrak{D}_{\mathcal{A}}$—to update the sets given to P. We're concerned with how many times a generator g must make an update before P learns a desired partition.

A generator $g \in G$ can be thought of as a function that takes as input a tuple $(L, c_{\text{guess}}, c_{\text{target}})$—where L is the list of subsets of $\mathfrak{D}_{\mathcal{A}}$ given as input to P, $c_{\text{guess}} \in C$ is a partition of $\mathfrak{D}_{\mathcal{A}}$ consistent with L, and $c_{\text{target}} \in C$ is the target partition—and outputs a new list L' of $\mathfrak{D}_{\mathcal{A}}$-subsets. We say g provides sets to P to refer to the iterative process in which $L_0 = \{\emptyset\}$ and $L_{i+1} = g(L_i, P(L_i), c_{\text{target}})$. Intuitively, a generator iteratively updates a list of sets to be given to a partitioning function so that the output of that function approaches the target partition.

Additionally, the generators are subject to the following restrictions that ensure a sense of monotonicity: (*i*) the output L' is *greater than* the input L in the sense that $\forall a \in \mathfrak{D}_{\mathcal{A}}(\exists \ell \in L.\, a \in \ell \rightarrow \exists \ell' \in L'.\, a \in \ell')$ (a character present in the input will always be present in future iterations); (*ii*) if $a_1 \in \ell_i \in L$ and $a_2 \in \ell_j \in L$ and $i \neq j$, then it cannot be that there is some $\ell' \in L'$ and both $a_1 \in \ell'$ and $a_2 \in \ell'$ (if the generator says two elements belong to different sets in a partition, that must be true for all future iterations); and (*iii*) either

- the number of sets in L' is larger than the number of sets in L
- at least one $a \in \mathfrak{D}_{\mathcal{A}}$ has been added to a set $\ell_i \in L$ so that $a \in \ell'_i \in L'$ and for the $\varphi_i \in c_{\text{guess}}$ corresponding to ℓ_i, $a \notin [\![\varphi_i]\!]$ (on the next application of P, a superset of $\{a\}$ that was not partitioned correctly will be fixed)

Also, the inputs to the generator are subject to a notion of consistency: if $a_1 \in \ell_i \in G$ and $a_2 \in \ell_j \in G$ such that $i \neq j$, then there is no $\varphi \in c_{\text{target}}$ such that $\{a_1, a_2\} \subseteq [\![\varphi]\!]$.

This definition of a generator exactly captures the high-level process of updating the observation table in our algorithm via queries to the oracle and *projecting* those changes onto the individual lists of sets that are given to the partitioning function for the creation of the conjectured s-FA. Below we will formalize a notion of learnability with respect to these generators, and it will thus bear a close correspondence to the complexity of learning an s-FA.

Definition 6 (s_g-learnability). *Given a Boolean algebra $\mathcal{A}$, a partitioning function P, and an generator $g \in G$, we say the pair $(\mathcal{A}, P)$ is* s_g-learnable *if there exists an implicit function $s_g : C \rightarrow \mathbb{N}$ such that P needs as input a list of sets, provided by g, with total size at most $s_g(c)$ to discover a target partition $c \in C$. Furthermore, we say $\mathcal{A}$ itself is s_g-learnable if there exists a partitioning function P such that $(\mathcal{A}, P)$ is s_g-learnable.*

We also classify $\mathcal{A}$ as belonging to a *learning class* that depends on these s_g functions—but first we need an auxiliary notion of the *size* of a partition.

Definition 7 (Size of a partition). *Let C be the set of partitions of $\mathcal{A}$. Each $c \in C$ is a list $\varphi_1, \ldots, \varphi_n$: we can expand each φ_i to a minimal disjunctive-normal-form formula $\bigvee_j \psi_{i,j}$ such that $c' = \psi_{1,1}, \ldots, \psi_{1,m_1}, \ldots, \psi_{n,1}, \ldots, \psi_{n,m_n}$ is a partition of $\mathcal{A}$ that is at least as fine as c. We say the* size *of c is the length of the list of such a minimal c'.*

Example 4. The partition $\{x < 5 \vee x > 10, 5 \leq x \wedge x \leq 10\}$ has size 3.

Definition 8 (Learning Class). *For a fixed Boolean algebra $\mathcal{A}$ if there exists a $g \in G$ such that $\mathcal{A}$ is s_g-learnable, then*

- *if s_g is a constant function, i.e. $\exists k \forall c.\, s_g(c) = k$, we say $\mathcal{A} \in \mathcal{C}^{\exists}_{const}$*
- *if s_g is a function only of the size of c, we say $\mathcal{A} \in \mathcal{C}^{\exists}_{size}$*
- *if s_g is otherwise unconstrained, we say $\mathcal{A} \in \mathcal{C}^{\exists}_{finite}$*

Additionally, for some fixed paritioning function P, if for all $g \in G$, $(\mathcal{A}, P)$ is s_g-learnable, then

- *if each s_g is a constant function, we say $\mathcal{A} \in \mathcal{C}^{\forall}_{const}$*
- *if each s_g is a function only of the size of c, we say $\mathcal{A} \in \mathcal{C}^{\forall}_{size}$*
- *if each s_g is otherwise unconstrained, we say $\mathcal{A} \in \mathcal{C}^{\forall}_{finite}$*

Observe that learning classes are partially-ordered by the subset relation:

$$\begin{array}{ccccc} \mathcal{C}^{\forall}_{const} & \subset & \mathcal{C}^{\forall}_{size} & \subset & \mathcal{C}^{\forall}_{finite} \\ \cap & & \cap & & \cap \\ \mathcal{C}^{\exists}_{const} & \subset & \mathcal{C}^{\exists}_{size} & \subset & \mathcal{C}^{\exists}_{finite} \end{array}$$

When $\mathcal{A} \in \mathcal{C}^{\forall}_{const}$, learning a partition over $\mathfrak{D}_{\mathcal{A}}$ is equivalent to the machine-learning notion of a mistake-bound [11]. The equality algebra for any finite alphabet is in $\mathcal{C}^{\forall}_{const}$. The Boolean algebra formed from unions of intervals over the integers, reals, or rationals, is in $\mathcal{C}^{\exists}_{size}$. The integer case is also in $\mathcal{C}^{\forall}_{finite}$. Using enumeration, linear rational arithmetic is in $\mathcal{C}^{\forall}_{finite}$. L: TODO elaborate.

L: TODO talk about the generator as a projection of the oracle onto the individual partitions. Since for each state in an s-FA, the set of outgoing transitions forms a partition of the alphabet, i.e. a concept in C, the number of counterexamples needed to learn the entire automaton is simply the sum of $s_g(c)$ for each state's outgoing transitions.

Theorem 3 (SFA Learnability). *If M is an s-FA over a learnable Boolean algebra $\mathcal{A}$, then the number of equivalence queries needed to learn M is polynomial in $\Sigma_{q_i \in Q} s_{g_i}(c_i)$, where s_{g_i} is the projection of the oracle onto learning the partition c_i for the outgoing transitions of state q_i.*

4.1 Composing Algebras

The definition of learnability described prior has the remarkable property that it is preserved by some constructions that combine Boolean algebras, such as the *disjoint union* and the *cartesian product*. In these cases, a partitioning function for the resultant algebra can be constructed by using partitioning functions for the original algebras as black boxes; This allows us to phrase the learnability of the constructed algebra in terms of the learnability of the individual algebras.

Definition 9 (Disjoint Union Algebra). *Let $\mathcal{A}_1, \mathcal{A}_2$ be boolean algebras. Their* disjoint union algebra $\mathcal{A}_\uplus = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$*, which we denote $\mathcal{A}_\uplus = \mathcal{A}_1 \uplus \mathcal{A}_2$, is constructed as follows:*[4]

$$\begin{aligned}
\mathfrak{D} &= \mathfrak{D}_{\mathcal{A}_1} \uplus \mathfrak{D}_{\mathcal{A}_2} \\
\Psi &= \Psi_{\mathcal{A}_1} \times \Psi_{\mathcal{A}_2} \\
[\![(\varphi_{\mathcal{A}_1}, \varphi_{\mathcal{A}_2})]\!] &= [\![\varphi_{\mathcal{A}_1}]\!]_{\mathcal{A}_1} \uplus [\![\varphi_{\mathcal{A}_2}]\!]_{\mathcal{A}_2} \\
\bot &= (\bot_{\mathcal{A}_1}, \bot_{\mathcal{A}_2}) \\
\top &= (\top_{\mathcal{A}_1}, \top_{\mathcal{A}_2}) \\
(\varphi_{\mathcal{A}_1}, \varphi_{\mathcal{A}_2}) \vee (\varphi'_{\mathcal{A}_1}, \varphi'_{\mathcal{A}_2}) &= ((\varphi_{\mathcal{A}_1} \vee_{\mathcal{A}_1} \varphi'_{\mathcal{A}_1}), (\varphi_{\mathcal{A}_2} \vee_{\mathcal{A}_2} \varphi'_{\mathcal{A}_2})) \\
(\varphi_{\mathcal{A}_1}, \varphi_{\mathcal{A}_2}) \wedge (\varphi'_{\mathcal{A}_1}, \varphi'_{\mathcal{A}_2}) &= ((\varphi_{\mathcal{A}_1} \wedge_{\mathcal{A}_1} \varphi'_{\mathcal{A}_1}), (\varphi_{\mathcal{A}_2} \wedge_{\mathcal{A}_2} \varphi'_{\mathcal{A}_2})) \\
\neg(\varphi_{\mathcal{A}_1}, \varphi_{\mathcal{A}_2}) &= (\neg_{\mathcal{A}_1}\varphi_{\mathcal{A}_1}, \neg_{\mathcal{A}_2}\varphi_{\mathcal{A}_2})
\end{aligned}$$

If $\mathcal{A}_1$ has partitioning function P_1 and $\mathcal{A}_2$ has partitioning function P_2, then we can construct a partitioning function $P_\uplus$ for $\mathcal{A}_\uplus = \mathcal{A}_1 \uplus \mathcal{A}_2$: $P_\uplus$ takes as input a list $L_\uplus$ of sets where each set $\ell_{\uplus_i} \subset \mathfrak{D}_{\mathcal{A}_1} \uplus \mathfrak{D}_{\mathcal{A}_2}$. We decompose $L_\uplus$ into $L_{\mathfrak{D}_1}$ and $L_{\mathfrak{D}_2}$, two lists of sets of $\ell_{1_i} \subset \mathfrak{D}_{\mathcal{A}_1}$ and $\ell_{2_i} \subset \mathfrak{D}_{\mathcal{A}_2}$, respectively: for each $a \in \ell_{\uplus_i}$, if $a \in \mathfrak{D}_{\mathcal{A}_1}$, then we add a to ℓ_{1_i}, and otherwise if $a \in \mathfrak{D}_{\mathcal{A}_2}$, then we add a to ℓ_{2_i}. We obtain $L_{\Psi_1} = P_1(L_{\mathfrak{D}_1})$ and $L_{\Psi_2} = P_2(L_{\mathfrak{D}_2})$. We construct $L_{\Psi_\uplus}$ by taking $\varphi_{\uplus_i} = \varphi_{1_i} \times \varphi_{2_i}$ for all i, return $L_{\Psi_\uplus}$, and terminate.

Theorem 4 (Disjoint Union Algebra Learnability). *Given Boolean algebras $\mathcal{A}_1, \mathcal{A}_2$ with partitioning functions P_1, P_2, $(\mathcal{A}_1, P_1)$ is s_{g_1}-learnable and $(\mathcal{A}_2, P_2)$ is s_{g_2}-learnable if and only if there exists $g_\uplus$ such that their disjoint union algebra $(\mathcal{A}_\uplus, P_\uplus)$ is $s_{g_\uplus}$-learnable, where $s_{g_\uplus}(c) = s_{g_1}(c_1) + s_{g_2}(c_2)$ and c_1 and c_2 are the restrictions of the subsets in c to $\mathfrak{D}_{\mathcal{A}_1}$ and $\mathfrak{D}_{\mathcal{A}_2}$, respectively.*

Corollary 1. *If $\mathcal{A}_1$ is in learning class $\mathcal{C}$ and $\mathcal{A}_2$ is in learning class $\mathcal{C}'$, then their disjoint union $\mathcal{A}_\uplus$ is in learning class $\mathcal{C} \cup \mathcal{C}'$.*

Definition 10 (Product Algebra). *Let $\mathcal{A}_1, \mathcal{A}_2$ be boolean algebras. Their* product algebra $\mathcal{A}_\times = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$*, which we denote $\mathcal{A}_\times = \mathcal{A}_1 \times \mathcal{A}_2$,*

[4] In our definition, we use $\mathfrak{D}_{\mathcal{A}_1} \uplus \mathfrak{D}_{\mathcal{A}_2}$ to denote the disjoint union of sets; rigorously, when the sets are not already disjoint, this is constructed by taking $(\mathfrak{D}_{\mathcal{A}_1} \times \{1\}) \cup (\mathfrak{D}_{\mathcal{A}_2} \times \{2\})$ and lifting all the remaining constructs appropriately.

is constructed as follows:

$$\begin{aligned}
\mathfrak{D} &= \mathfrak{D}_{\mathcal{A}_1} \times \mathfrak{D}_{\mathcal{A}_2} \\
\Psi &= 2^{\Psi_{\mathcal{A}_1} \times \Psi_{\mathcal{A}_2}} \\
[\![\{(\varphi_{\mathcal{A}_1 i}, \varphi_{\mathcal{A}_2 i})\}_i]\!] &= \textstyle\bigcup_i ([\![\varphi_{\mathcal{A}_1 i}]\!]_{\mathcal{A}_1} \times [\![\varphi_{\mathcal{A}_2 i}]\!]_{\mathcal{A}_2}) \\
\bot &= \{(\bot_{\mathcal{A}_1}, \bot_{\mathcal{A}_2})\} \\
\top &= \{(\top_{\mathcal{A}_1}, \bot_{\mathcal{A}_2})\} \\
\{(\varphi_{\mathcal{A}_1 i}, \varphi_{\mathcal{A}_2 i})\}_i \vee \{(\varphi'_{\mathcal{A}_1 j}, \varphi'_{\mathcal{A}_2 j})\}_j &= \{(\varphi_{\mathcal{A}_1 i}, \varphi_{\mathcal{A}_2 i})\}_i \cup \{(\varphi'_{\mathcal{A}_1 j}, \varphi'_{\mathcal{A}_2 j})\}_j \\
\{(\varphi_{\mathcal{A}_1 i}, \varphi_{\mathcal{A}_2 i})\}_i \wedge \{(\varphi'_{\mathcal{A}_1 j}, \varphi'_{\mathcal{A}_2 j})\}_j &= \{(\varphi_{\mathcal{A}_1 i} \wedge_{\mathcal{A}_1} \varphi'_{\mathcal{A}_1 j}, \varphi_{\mathcal{A}_2 i} \wedge_{\mathcal{A}_2} \varphi'_{\mathcal{A}_2 j}) \mid \forall i, j\} \\
\neg\{(\varphi_{\mathcal{A}_1 i}, \varphi_{\mathcal{A}_2 i})\}_i &= \textstyle\bigwedge_i \{(\neg_{\mathcal{A}_1} \varphi_{\mathcal{A}_1 i}, \top_{\mathcal{A}_2}), (\top_{\mathcal{A}_1}, \neg_{\mathcal{A}_2} \varphi_{\mathcal{A}_2 i})\}
\end{aligned}$$

If $\mathcal{A}_1$ has partitioning function P_1 and $\mathcal{A}_2$ has partitioning function P_2, then we can construct a partitioning function $P_\times$ for $\mathcal{A}_\times = \mathcal{A}_1 \times \mathcal{A}_2$: $P_\times$ takes as input a list $L_\times$ of sets where each set $\ell_{\times_i} \subset \mathfrak{D}_{\mathcal{A}_1} \times \mathfrak{D}_{\mathcal{A}_2}$. We first take the set $D_1 = \{d_1 \mid (d_1, d_2) \in \ell_{\times_i}$ for some $\ell_{\times_i} \in L_\times\}$, turn it into a list $D'_1 = \{d_{1,1}\}, \ldots, \{d_{1,n}\}$, and compute a partition $L_1 = P_1(D'_1)$. Then for each $d_i \in D_1$, we construct a list D_{2,d_i} where the j-th element is the set $\{d_2 \mid (d_i, d_2) \in \ell_{\times_j}\}$ and compute a partition $L_{2,d_i} = P_2(D_{2,d_i})$. Finally, we initialize the list of predicates to be returned $L_{\Psi_\times} = \varphi_{\times_1}, \ldots, \varphi_{\times_k}$ so that initially each $\varphi_{\times_i} = \bot$. Then for all i and each $(d_1, d_2) \in \ell_{\times_i}$, let φ_{d_1} be the predicate in L_1 corresponding to $\{d_1\}$ in D'_1 and let φ_{d_2} be the predicate in L_{2,d_1} corresponding to the set of D_{2,d_1} that contains d_2; update $\varphi_{\times_i} \leftarrow \varphi_{\times_i} \vee (\varphi_{d_1}, \varphi_{d_2})$. Return $L_{\Psi_\times}$ and terminate.

Example 5. Suppose we want to find a partition over $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ with each component using the algebra of intervals, and suppose the input sets are $L_\times = [\{(0,0), (1,0), (1,2)\}, \{(0,2)\}]$. Then $D'_1 = [\{0\}, \{1\}]$ and perhaps $L_1 = P_1(D'_1) = [x \leq 0, x > 0]$. Then we have $D_{2,0} = [\{0\}, \{2\}]$ and $D_{2,1} = [\{0, 2\}, \emptyset]$. Perhaps $L_{2,0} = P_2(D_{2,0}) = [y \leq 1, y > 1]$ and $L_{2,1} = P_2(D_{2,1}) = [\top, \bot]$. Then (without simplification) $L_{\Psi_\times} = [(x \leq 0, y \leq 1) \vee (x > 0, \top) \vee (x > 0, \top), (x \leq 0, y > 1)]$

Theorem 5 (Product Algebra Learnability). *Given Boolean algebras $\mathcal{A}_1, \mathcal{A}_2$ with partitioning functions P_1, P_2, $(\mathcal{A}_1, P_1)$ is s_{g_1}-learnable and $(\mathcal{A}_2, P_2)$ is s_{g_2}-learnable if and only if there exists $g_\times$ such that their product algebra $(\mathcal{A}_\times, P_\times)$ is $s_{g_\times}$-learnable: let $c \in C_\times$ be the target partition over the product algebra, let $c_1 \in C_1$ be the minterms of the $\mathcal{A}_1$-components of c, and let $c_2 \in C_2$ be the minterms of the $\mathcal{A}_2$-components of c. Finally, $s_{g_\times}(c) = s_{g_1}(c_1) s_{g_2}(c_2)$.*

Corollary 2. *If $\mathcal{A}_1$ is in learning class $\mathcal{C}$ and $\mathcal{A}_2$ is in learning class $\mathcal{C}'$, then their product $\mathcal{A}_\times$ is in learning class $\mathcal{C} \cup \mathcal{C}'$.*

Since our notion of learnability is closed under the disjoint union and product constructions, it generalizes to arbitrary non-recursive data types; symbolic automata over these arbitrary types can thus be learned using partitioning functions for the component types, as opposed to necessitating highly specialized partitioning functions.

Fundamentally, learning a symbolic automaton efficiently requires leveraging knowledge about the oracle to construct a good partitioning function. We will discuss in later sections how several existing s-FA-learning techniques implicitly perform such a construction.

5 Implementation

We have implemented our algorithm using the open-source Java library Symbolic Automata.[5] The implementation is truly modular and only requires the programmer to provide learnable Boolean algebras as input to the learner; the disjoint union and product algebras are implemented meta-algebras that can be instantiated arbitrarily.

Maler and Mens [12] present an algorithm for s-FA learning over a totally ordered alphabet: union of intervals over a bounded subset of the integers. Their technique works by assuming the oracle always provides lexicographically minimal counterexamples, so every counterexample identifies a boundary in a partition. They learn an example automaton in L: todo equivalence and L: todo membership queries. We implemented a partitioning function equivalent to their characterization and are able to learn the same automaton in ? equivalence and ? membership queries.

Maler and Mens also demonstrate their technique can be applied to partially ordered alphabets: they show an example using the product of unions of intervals over a bounded subset of the integers that usees an ad-hoc requirement that the oracle always gives them minimal counterexamples with respect to the partial order. With this set up, they are able to learn an example automaton in 18 equivalence and 20 membership queries. We learned this automaton by using the meta-partitioning function for product algebras presented in Section 4.1, which is able to learn the automaton in 28 equivalence and 70 membership queries. The discrepency is due to the fact that Maler and Mens have a more specialized implicit partitioning function, the result in Theorem 5 suggests that using the meta-algebra should require about quadratically more equivalence queries. In fact, if we implement an analagous specialized partitioning function directly on the product algebra, we can learn the example in ? equivalence and ? membership queries.

6 Related Work

Improvements of L^*. Our Λ^* algorithm builds on classic L^* [3], but since then, many extensions have been proposed, the most advanced one being the TTT algorithm [9]. The open source library *LearnLib* contains implementations of all such variants [1]. We expect existing improvements of L^* to be applicable to our Λ^*. However, while these optimizations can potentially improve the expected size of the observation table, the number of membership queries will be dictated

[5] Available at `https://github.com/lorisdanto/symbolicautomata`.

by the amount of evidence needed for the partitioning function to generalize; thus, the impact of such optimizations is not clear and is left for future work. Our algorithm opens many new research questions: How can we efficiently store intermediate predicates computed using the partitioning functions? For what theories can separating predicates be computed incrementally?

Automata Learning and Infinite Alphabets. While other algorithms for learning automata over infinite alphabets have been studied, our paper is the first one to provide: 1) an algorithm for learning symbolic automata over arbitrary alphabet theories, with a notion of learnability that is parametric in both the alphabet theory and the generator, and 2) compositionality properties that allow to combine learnable algebras. We detailed our comparison against the most relevant works.

Isberner et al proposed a variant of L^* for learning automata over large and potentially infinite alphabets using abstractions [8]. Given concrete symbols, the abstraction function generates a representative abstract symbol and the final automaton operates over the alphabet of abstract symbols. Overly coarse abstractions can cause non-determinism that is then resolved using refinement operators. This approach differs from ours in two aspects. First, while the final output of $\Lambda^*$is a symbolic automaton over the target Boolean algebra, the output in [8] is an automaton operating over a separate abstract alphabet that is discovered during the learning process and might not necessary form a Boolean algebra . Second, while our algorithm enjoys compositionality properties over the input Boolean algebras, the one in [8] does not provide any such guarantee as the abstractions are not known a priori.

Maler and Mens [12] instantiate the algorithm proposed in [8] and learn symbolic automata over the interval algebra for the integers. Where we adapt the observation table to contain only a subset of what would be included in L^*, they instead use an observation table whose entries are from a *symbolic alphabet*, or in other words, their table entries are elements in Ψ^*, each accompanied by a *concrete word* in Σ^* that served as evidence to create the symbolic word. This formulation of the observation table is possible due to the constraint they make on the oracle: it must provide lexicographically minimal counterexamples. This way, every counterexample uniquely determines the upper or lower bound for a symbolic letter. In our technique, which subsumes their method, the symbolic alphabet (i.e., the set of relevant predicates) is dynamically generated at each iteration of the algorithm. Using our technique, we can also drop the assumption that the the oracle provides lexicographically minimal counterexamples. Finally, [12] provides a modified version of the algorithm for learning automata over pairs of integers by constraining that the oracle gives them minimal counterexamples in the partial order of the pairs. Remarkably, we can obtain their modified algorithm for free using the Cartesian product of the interval algebra with itself.

Argyros et al [5] present an algorithm for learning symbolic automata that is similar to ours: the learnability is parametric with respect to their *guardgen* method, which is an equivalent formulation of the partitioning function expressed in this paper. Their definition of learnability is equivalent to our $\mathcal{C}^{\forall}_{const}$ and can

therefore only capture Boolean algebras operating over finite alphabets or with finitely many predicates. Our work introduces generators, proposes a deeper analysis of the learnability of a Boolean algebra, and shows how learnable algebras can be composed.

The Sigma* algorithm by Botinčan and Babić [6] studies the related but disjoint problem of learning symbolic transducers (automata with outputs). The paper does not propose a notion of learnability.

Algorithms have been proposed for learning nominal and register automata [13]. In these models, the alphabet is infinite but not structured (i.e., it does not form a Boolean algebra) and characters at different positions can be compared using binary relations (typically equality, inequality, or simple arithmetic relations such as $y = x + c$). These models are orthogonal to the ones studied in this paper

Oracle-guided synthesis. Our work is closely related to the notion of oracle-guided inductive synthesis (OGIS) [10], where automata learning and its termination are characterized in terms of the interactions with the oracle and the nature of its counterexamples. Our work more strongly connects the complexity of learning with the *learnability* of the underlying algebra, which is parametric in the selection of a partitioning function. Additionally, OGIS focuses on a distinction between the expressiveness of infinite-memory and finite-memory systems for learning; in this paper we assume learning is being done only in the context of infinite-memory, but we derive stronger results about the complexity of the learnability as opposed to simply whether or not an automaton is learnable.

References

1. Learnlib, a framework for automata learning. `http://learnlib.de/`.
2. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, Jan. 2005.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
4. D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3308–3314. AAAI Press, 2015.
5. G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in black: Towards formal, black box analysis of sanitizers and filters. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 91–109, 2016.
6. M. Botincan and D. Babic. Sigma*: symbolic learning of input-output specifications. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 443–456, 2013.
7. P. García, M. V. de Parga, G. I. Álvarez, and J. Ruiz. *Learning Regular Languages Using Nondeterministic Finite Automata*, pages 92–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
8. M. Isberner, F. Howar, and B. Steffen. *Inferring Automata with State-Local Alphabet Abstractions*, pages 124–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

9. M. Isberner, F. Howar, and B. Steffen. *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning*, pages 307–322. Springer International Publishing, Cham, 2014.
10. S. Jha and S. A. Seshia. A theory of formal synthesis via inductive learning. *CoRR*, abs/1505.03953, 2015.
11. N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.
12. I. Mens and O. Maler. Learning regular languages over large ordered alphabets. *Logical Methods in Computer Science*, 11(3), 2015.
13. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.
14. Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 20:1–20:7, New York, NY, USA, 2014. ACM.