Automata Learning with an Incomplete Teacher

Mark Moeller
Cornell University, Ithaca, NY, USA

Thomas Wiener
Cornell University, Ithaca, NY, USA

Alaia Solko-Breslin
University of Pennsylvania, Philadelphia, PA, USA

Caleb Koch
Stanford University, Stanford, CA, USA

Nate Foster
Cornell University, Ithaca, NY, USA

Alexandra Silva
Cornell University, Ithaca, NY, USA

Abstract

The preceding decade has seen significant interest in use of active learning to build models of programs and protocols. But existing algorithms assume the existence of an idealized oracle—a so-called Minimally Adequate Teacher (MAT)—that cannot be fully realized in practice and so is usually approximated with testing. This work proposes a new framework for active learning based on an incomplete teacher. This new formulation, called iMAT, neatly handles scenarios in which the teacher has access to only a finite number of tests or otherwise has gaps in its knowledge. We adapt Angluin's $L^\star$ algorithm for learning finite automata to incomplete teachers and we build a prototype implementation in OCaml that uses an SMT solver to help fill in information not supplied by the teacher. We demonstrate the behavior of our iMAT prototype on a variety of learning problems from a standard benchmark suite.

2012 ACM Subject Classification Theory of computation → Active learning

Keywords and phrases Finite Automata, Active Learning, SMT Solvers

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.20

Supplementary Material *Software (Source Code)*: `https://github.com/cornell-pl/nerode-public`

Funding This material is based upon work supported by the Office of Naval Research under Contract No. N68335-22-C-0411.

Acknowledgements We thank Marijn Heule, Martin Leucker, and Arlindo Oliveira for their efforts in providing us access to their code and benchmarks. We also thank Akshat Singh and Sheetal Athrey, with whom this project began as an undergraduate research project.

1 Introduction

Automata are among the most basic structures in computer science, yet they continue to offer profound insights for modeling and analyzing systems. Recent years have seen renewed interest in the problem of *closed-box inference* of automata, often motivated by applications in verification and security—see [33] for an overview.

Many of the algorithms developed for closed-box inference are based on the *minimally adequate teacher* (MAT) framework, in which a Learner interacts with a Teacher (also sometimes referred to as an oracle) in an attempt to learn a finite automaton known by the Teacher. The Learner can pose two types of queries to the Teacher: membership queries ("does the automaton accept a given input word?") and equivalence queries ("is a given automaton correct?"). It turns out that these primitives are sufficient to show that the MAT framework terminates and produces the correct automaton. Numerous variants of the basic algorithm have been proposed (e.g., $\mathtt{L}^\star$ [2], `KV` [19], `TTT` [18] or $\mathtt{L}^\sharp$ [34]), using different data structures to collect and organize gathered information. The MAT framework has also been instantiated for other kinds of automata—e.g Mealy machines, weighted automata [5], and nominal automata [28] to name a few.

Contrary to its name, however, in many practical settings, even finding a Minimally Adequate Teacher is challenging. For example, consider learning a model of a closed-box system. How would the Teacher determine whether a given automaton supplied by the Learner captures the behavior of the closed-box system? The best the Teacher can do is check whether they agree on a finite set of tests. Likewise, in settings where the Teacher only has access to a set of positive and negative examples (also known as passive learning) it is unclear how to answer membership queries for inputs that lie outside of the example set.

This paper presents an alternative to MAT: the *incomplete* Minimally Adequate Teacher (iMAT) framework. An iMAT is allowed to answer "don't care" in response to membership queries and it does not answer equivalence queries at all. We show that Angluin's classic $\mathtt{L}^\star$ algorithm can be instantiated with an incomplete Teacher, by using an SMT solver to help fill in the missing information without needlessly expanding the size of the automaton. More precisely, we show—under modest assumptions—that our algorithm infers the minimal automaton compatible with the information provided by the Teacher. We present an OCaml prototype and demonstrate its behavior on well-known benchmarks [23, 29].

We see the framework developed in this paper as a first step towards building connections between several different areas. As we explain later, iMAT provides a bridge between active and passive learning. In active learning, additional information can always be gathered (i.e., from the Teacher), whereas in passive learning, the assumption is that all of the information that will be used for learning has been gathered in advance. With iMAT, one can use the passive information as an incomplete Teacher and proceed to learn the automaton using active techniques like $\mathtt{L}^\star$. A lack of perfect information about the language in question also arises in program synthesis, in particular in the "programming by example" paradigm [7, 22, 23, 25, 36]. Here, the goal is to infer a model that is correct for the examples, and hope that it generalizes to other scenarios—i.e., that the examples are somehow representative of a more general pattern. We explore this connection in our use of benchmarks from AlphaRegex [23].

The problem of using finite information to infer a DFA was first studied by Gold in the 1970s [14], and has since been studied by many others [32, 30, 21, 20]. Gold showed that finding a DFA with the minimum number of states is NP-complete [15]. As iMAT subsumes DFA inference from finite data, its scalability is necessarily limited.

Others have studied problems similar to iMAT in prior work, either in their use of incomplete information or SAT solvers [8, 29, 17, 16, 24]. Leucker and Neider's paper [24] includes an "inexperienced teacher" and surveys several learners with similar capabilities as ours. Section 10 presents a detailed survey of their paper and other related work. No previous work has studied iMAT in full generality, including: formulating the problem, establishing correctness, building an open-source implementation, and exploring alternate formulations. Hence, compared to prior work, this paper makes the following contributions:

1. We review the MAT framework (Section 2) which sets the stage for our formulation of active learning based on an incomplete Teacher (Section 3). We also adapt the notion of a Learner, giving it access to a Solver.

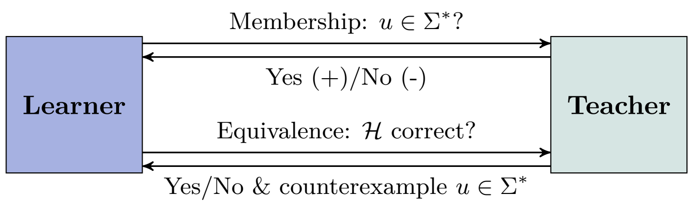

Membership: $u \in \Sigma^*$?
Yes (+)/No (-)
Learner
Teacher
Equivalence: $\mathcal{H}$ correct?
Yes/No & counterexample $u \in \Sigma^*$

Figure 1 The Minimally Adequate Teacher framework.

2. We instantiate the iMAT framework in the context of DFAs, using an SMT solver to fill in missing information (Section 4).
3. We prove that the Learner terminates and returns a minimal automaton compatible with the known information (Section 5).
4. We show how to modify our learning algorithm in the presence of the more limited Teacher that uses a simpler interface for compatibility checks (Section 6), and prove a (partial) correctness result (Section 7).
5. We implement the Learner in OCaml and present optimizations to accelerate the search for a DFA. We evaluate our proposed optimizations with an ablation study, and present performance data on a standard benchmark suite [23, 29].

Next, we review MAT and L*, to set the stage for the iMAT framework in later sections.

2 The MAT framework

In the MAT framework, a Learner seeks to discover an unknown language by interacting with a Teacher who can answer two types of queries:

Membership: The Learner sends a word u to the Teacher, who answers "yes" if u belongs to the language or "no" if it does not.

Equivalence: The Learner sends a hypothesis automaton $\mathcal{H}$ to the Teacher, who either confirms that $\mathcal{H}$ is correct or provides a counterexample.

The MAT framework has been the basis for active model learning since its introduction in the late 1980s, providing the basis for a variety of algorithms, data structures, and proof techniques (e.g. L* [2], KV [19], TTT [18], $L^\sharp$ [34], etc.). The rest of this section presents Angluin's L* algorithm, which introduces the notation and concepts used in this paper.

2.1 Strings, Languages, and Automata

Fix an alphabet of symbols, Σ. We let $a \in \Sigma$ denote an arbitrary symbol and ε the empty string. For strings (also called words) $u, v \in \Sigma^*$, we write uv for the concatenation of u and v (we will occasionally write $u.v$ for emphasis). We use capital letters $U, V \subseteq \Sigma^*$ to denote languages—i.e. sets of strings. Concatenation of languages U, V is written $U \cdot V = \{uv \mid u \in U, v \in V\}$. The set of prefixes for a string s is written $\mathrm{prefixes}(s) = \{u \mid s = uv, u, v \in \Sigma^*\}$. Likewise, $\mathrm{suffixes}(s) = \{v \mid s = uv, u, v \in \Sigma^*\}$. We sometimes even take prefixes of a set $S \subseteq \Sigma^*$, written $\mathrm{prefixes}(S) = \{u \mid u \in \mathrm{prefixes}(s), s \in S\}$. A set $S \subseteq \Sigma^*$ is called *prefix-closed* (*suffix-closed*) if $S = \mathrm{prefixes}(S)$ ($S = \mathrm{suffixes}(S)$). Finally, we denote the symmetric difference of two sets by $S \oplus S' = S - S' \cup S' - S$.

An important class of languages is that of *regular languages*, which are the languages accepted by deterministic finite automata (DFAs). A DFA is a five-tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta \colon Q \times \Sigma \to Q$ is the transition function,

$q_0 \in Q$ is the start state, and $F \subseteq Q$ are the accepting states. The transition function is extended to strings inductively by $\hat{\delta}\colon Q \times \Sigma^* \to Q$, where for any $q \in Q$, $\hat{\delta}(q, \varepsilon) = q$ and for any $a \in \Sigma, u \in \Sigma^*$, $\hat{\delta}(q, au) = \hat{\delta}(\delta(q, a), u)$. A string u is accepted by the automaton D iff $\hat{\delta}(q_0, u) \in F$. Moreover, the language accepted by D, written $L(D)$, is the set of all such strings: $L(D) = \{u \in \Sigma^* | \ \hat{\delta}(q_0, u) \in F\}$.

Regular languages have unique minimal representatives: for every regular language, there is a unique DFA with a minimal number of states. There are different ways to build such minimal DFA corresponding to a regular language, but the one that is used to prove correctness of L* is based on so-called Myhill-Nerode equivalence classes. Given a language L and a word $s \in \Sigma^*$, the Myhill-Nerode equivalence class of s, denoted $[s]_L$ is the set of all words s' satisfying: $s \equiv_L s' \overset{\triangle}{\iff} \forall_{e \in \Sigma^*} \cdot (se \in L \iff s'e \in L)$. Myhill-Nerode equivalence classes provide an alternative characterization of regular languages: a language is regular iff it has a finite number of Myhill-Nerode equivalence classes. Furthermore, there is a one-to-one correspondence between the states of the minimal automaton accepting a regular language L and its Myhill-Nerode equivalence classes.

2.2 L*: Data Structures

L* is an algorithm that implements a Learner according to the interfaces in the MAT framework (see Figure 1). The core data structure used in the algorithm is an observation table, which records the information gathered from membership queries and the counterexamples obtained from equivalence queries.

▶ **Definition 1** (Observation Table). *Given a language $L \subseteq \Sigma^*$, an* observation table *(wrt L) is a triple (S, E, T), where:*

- *$S \subseteq \Sigma^*$ is a prefix-closed set of "accessor strings";*
- *$E \subseteq \Sigma^*$ is a suffix-closed set of "distinguishing strings";*
- *$T\colon (S \cup S \cdot \Sigma) \cdot E \to \{+, -\}$ is a map on a finite set of words defined by*

$$T(u) = \begin{cases} + & u \in L \\ - & u \notin L \end{cases}$$

Note that once L is fixed, T is fully determined by S and E, so sometimes we refer to the observation table as simply (S, E). We can think of $L \subseteq \Sigma^*$ as a function $L\colon \Sigma^* \to \{+, -\}$ and then simply write $T(u) = L(u)$. Every u in the domain of T is obtained as a concatenation of a string $s \in S \cup S \cdot \Sigma$ and $e \in E$.

Intuitively, one can think of an observation table as a snapshot of a language L. When L is regular, we will show how the table is organized in a way that its rows can be used to recover the Myhill-Nerode equivalence classes of L, and therefore the minimal deterministic automaton. The use of accessor strings and distinguishing strings to name the elements of S and E will become clear when we explain how to build a DFA from a table.

Consider the regular language $L = \{w \in \{a, b\}^* \mid w \text{ has at least one } a\}$, and sets $S = \{\varepsilon, a\}$, and $E = \{\varepsilon, b\}$. The observation table (S, E) is given on the left in Figure 2. When depicting the table, we divide it into an upper part and a lower part. The rows in the upper part are labelled by strings in S, whereas the rows of the lower part are labelled by strings in $S \cdot \Sigma - S = \{sa \mid s \in S, a \in \Sigma, \ sa \notin S\}$. The strings in E label the columns of the table. The entries of the table correspond to the function T.

It is important to note that depicting the finite map $T\colon (S \cup S \cdot \Sigma) \cdot E \to \{+, -\}$ as a table leads to some repetition. For example, the entry in row a, column b must match the

	ε	b
ε	-	-
a	+	+
b	-	-
aa	+	+
ab	+	+

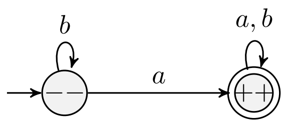


Figure 2 An observation table and corresponding automaton.

entry in row ab, column ε—i.e., since $a.b = ab.\varepsilon$ we also have $T(a.b) = T(ab.\varepsilon)$. It will be convenient to access the rows of the table as follows:

$$\text{row}\colon S \cup S \cdot \Sigma \to E \to \{+, -\} \qquad \text{row}(s)(e) = T(se).$$

We will sometimes abuse notation and apply row to a set of strings to obtain a set of rows. For example, in the table above row(S) is the set of distinct rows in the upper part of the table—row$(S) = \{\{\varepsilon \mapsto +, b \mapsto +\}, \{\varepsilon \mapsto -, b \mapsto -\}\}$. Additionally, when E is clear from the context, we will sometimes omit the column indices, writing row$(S) = \{++, --\}$.

To build a DFA from an observation table, we will use as states the upper rows of the table, but the well-definedness of the construction requires two properties to be satisfied[1].

▶ **Definition 2** (Closedness and Distinctness). *A table (S, E, T) is* closed *if for each string $s \in S$ and letter $a \in \Sigma$, we have $row(sa) \in row(S)$. It is* distinct *if all the rows labelled by $s \in S$ are distinct: $s, s' \in S \Rightarrow row(s) \neq row(s')$.*

Note that the example table above is closed and distinct. We can now make the connection between observation tables and finite automata precise.

▶ **Definition 3** (DFA associated with a table). *Given a closed and distinct table (S, E, T), with distinct rows in S, we associate a DFA, $\mathcal{D}(S, E, T) = (Q, \Sigma, \delta, q_0, F)$, where:*

$$Q = row(S) \qquad \delta(row(s), a) = row(sa) \qquad q_0 = row(\varepsilon) \qquad F = \{row(s) \mid row(s)(\varepsilon) = +\}.$$

The above definition relies on the fact that S is prefix-closed and E is suffix-closed and so both contain ε. Moreover, the transition function δ is well-defined whenever the table is closed and distinct. The first property ensures that row$(sa) \in Q$, whereas the second ensures the well definedness of $\delta(\text{row}(s), a)$.

The DFA corresponding to the example observation table is depicted on the right in Figure 2. The start state is indicated with an incoming arrow, and final state with a double circle. Each state is labeled by its unique row vector.

2.3 L* learner

We are now ready to present L*, the algorithm in which a Learner incrementally builds an observation table based on interactions with a Teacher as depicted in Figure 1. The L* Learner is shown in Figure 3. It starts with $S = E = \{\varepsilon\}$ and then explores potential new rows by examining the rows in the lower part of the table (labelled by strings in $S \cdot \Sigma - S$). If

[1] Readers familiar with L* will notice we jettisoned consistency for a simpler property—keeping the upper rows of the table distinct. This optimization is due to Maler and Pnueli [27].

```
1. Initialize $S = \{\varepsilon\}$, and $E = \{\varepsilon\}$
2. While $(S, E, T)$ is not closed, do:
       $S \leftarrow S \cup \{sa\}$,
       where $s \in S, a \in \Sigma$, but $\text{row}(sa) \notin \text{row}(S)$.
3. Conjecture $M = \mathcal{D}(S, E, T)$:
   a. If $M$ is correct, halt.
   b. Otherwise, $E \leftarrow E \cup \text{suffixes}(c)$, where $c$ is the provided counterexample. Goto 2.
```

Figure 3 Angluin's L⋆ (Maler-Pnueli version). Whenever S and E change, T is updated using membership queries. Note the extensions of S and E preserve distinctness as an invariant.

no new rows are found (i.e., as compared to the ones in row(S)), then we conclude that the table is closed. If, on the other hand, a row in the lower part of the table does not appear in the upper part, there is a closedness defect, which can be repaired by adding another string to S. In essence, this repair moves a row from the lower part of the table to the upper part. Accordingly, we generate new rows in the lower part until there is a row in the table for each $s \in S\Sigma$. Once the Learner finds a closed table (distinctness is maintained by construction), it poses an equivalence query to the Teacher. If the hypothesis is wrong the Teacher returns a counterexample which is used to extend the columns of the table.

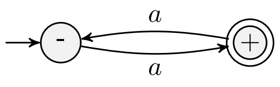
L⋆ can also be understood as incrementally discovering the Myhill-Nerode equivalence classes for the Teacher's language $\mathcal{L}$, which correspond to the states of the minimal DFA. Since the start state always corresponds to the equivalence class for the empty string ε, it starts with the observation table where $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$. Each hypothesis is a refinement of the previous guess, getting closer and closer to the correct minimal automaton. Termination of the algorithm follows as every closedness defect repaired or counterexample processed, results in an automaton with more states than the previous hypothesis.

2.4 L⋆ Example

Suppose we choose $\Sigma = \{a\}$ and the Teacher knows the language L described by the regular expression $a(aaa)^*$. The Learner begins by building the observation table below, on the left:

	ε
ε	-
a	+

	ε
ε	-
a	+
aa	-



(1)

The table on the left is not closed, because the row for a is not represented in the upper part of the table, hence the Learner extends S and this yields the table above in the middle. This table is closed so the Learner conjectures the corresponding DFA on the right.

The Teacher returns counterexample aaa (which is accepted by the automaton in Equation (1) but should be rejected).[2] The Learner processes this counterexample by extending E with its suffixes $\{\varepsilon, a, aa, aaa\}$, yielding the table on the left below:

[2] In general, there is no requirement for the Teacher to return the shortest counterexample and, indeed, the original complexity analysis of L⋆ takes into account that longer counterexamples might be returned resulting in larger than needed tables. Note that, however, minimality is never compromised as equal rows will be mapped to the same state in the automaton.

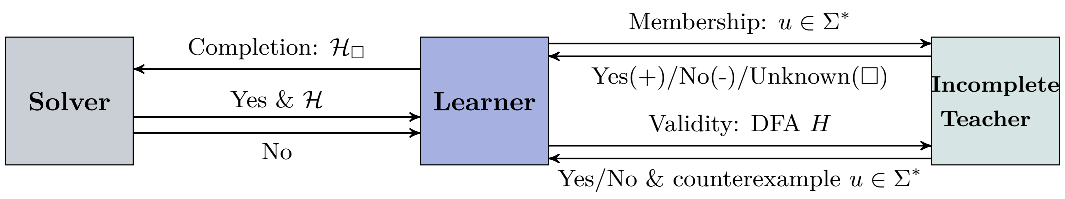
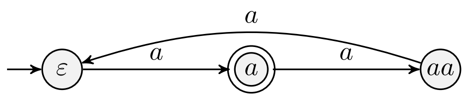

Solver
Completion: $\mathcal{H}_\square$
Yes & $\mathcal{H}$
No
Learner
Membership: $u \in \Sigma^*$
Yes(+)/No(-)/Unknown(□)
Validity: DFA H
Yes/No & counterexample $u \in \Sigma^*$
Incomplete Teacher

Figure 4 The Incomplete Minimally Adequate Teacher framework (iMAT).

	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-

	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-
aaa	-	+	-	-

The table on the left above is not closed, as row(aa) is different from all other rows. To fix this, the Learner adds aa to S yielding the table above, on the right. This table is closed and distinct, so the Learner makes the following conjecture which the Teacher accepts:



(2)

Angluin [2] showed that the L⋆ learner finds the minimal DFA for the target language using a polynomial number of membership queries—results which also apply to the Maler-Pnueli version. However, this property relies on the assumption that the Teacher can answer membership and equivalence queries.

The rest of this paper is devoted to introducing an alternative to the MAT framework, in which the Teacher is incomplete—it might not have answers to all the membership questions—but there is a third agent in the process that can help the Learner guess the missing information. We will then devise, implement, and evaluate an algorithm based on L⋆.

The challenges in devising and implementing such an algorithm are two-fold: on the one hand, since we do not have access to arbitrary membership queries, we will be building a table that has *holes*; on the other hand, the notion of minimality is now with respect to the existing information. Hence, progress towards this automaton is not as simple as in L⋆, requiring the use of heuristics to guess the missing information while still minimizing the size of the final automaton.

3 The iMAT Framework

This section introduces iMAT, a new framework for automata learning in which the Teacher is *incomplete* (see Figure 4). In iMAT, the Learner still wants to infer a regular language L, but the teacher only has partial information about the language. In particular, instead of holding an explicit language L, the Teacher is assumed to hold disjoint, possibly infinite sets of positive L_+ and L_- negative examples. The Learner seeks to find any language L such that $L_+ \subseteq L$ and $L_- \cap L = \varnothing$. In the literature, such an L is said to *separate* L_+ and L_- [8]. We assume that at least one regular L exists, but there may be several.[3]

[3] Note that we do *not* require that L_+ or L_- be regular. For example, neither $L_+ = a^n b^n, n > 0$ nor $L_- = b^n a^n, n > 0$ are regular, but they are separated by $L = a^* b^*$.

```
1. worklist $\leftarrow [(\{\varepsilon\}, \{\varepsilon\})]$
2. Call to SMT solver to fill in $\square$s with $+$ or $-$ in $\mathbf{hd}$(worklist).
   a. if UNSAT then:
      i. worklist $\leftarrow \mathbf{tl}$(worklist)@$[(S \cup \{s'\}, E) \mid s' \in S \cdot \Sigma - S]$
      ii. Goto step (2).
   b. if SMT solver returns table $(S, E)$, build the corresponding DFA and make a validity query.
      i. If the validity query succeeds, return the DFA.
      ii. Otherwise, get a counterexample $c$, set $E' = E \cup$ suffixes($c$), build table with $\square$'s $(S, E')$
          and set worklist $\leftarrow (S, E')$ :: worklist; goto step (2).
```

Figure 5 $\mathtt{L}^\star$ with blanks algorithm, $\mathtt{L}^\star_\square$. We use '@' and '::' to denote standard list operations (i.e., append and cons). As before, we omit membership queries; these occur whenever S and E are changed. Since observation tables are fully determined by S and E, we elide T.

The iMAT framework has three components: a Learner, an incomplete Teacher, and a Solver. The Teacher answers two types of queries, like MAT, but with weaker assumptions:

Membership: The Learner sends a word u to the incomplete Teacher, who answers with "yes" (+), "no" (-), or "unknown" ($\square$).

Validity: Given a hypothesis automaton, $\mathcal{H}$, the teacher determines whether $L_+ \subseteq L(\mathcal{H})$ and $L_- \cap L(\mathcal{H}) = \varnothing$. If so, it returns *None*, otherwise, it returns a counterexample string either in $L_+ - L(\mathcal{H})$ or in $L_- \cap L(\mathcal{H})$.

The other component in the iMAT framework is the Solver which the Learner can ask for help in completing the hypothesis construction. The solver answers just one type of query:

Completion: The Learner sends an incomplete hypothesis to the solver and asks whether there is a way to complete it into a full automaton. The Solver either says "yes" and returns a complete hypothesis or "no".

iMAT's Validity query is obviously similar to MAT's Equivalence query, but it is different in an important way: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns "yes" whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly. So while Validity may seem just as complex as Equivalence, we include it here as a first step toward more practical queries.

▶ Remark 4. Note that if the incomplete Teacher happens to be a complete oracle, then the iMAT setup can be used for MAT. Consider an iMAT instance where (i) the membership queries always return "yes" or "no" and (ii) the validity query returns a counterexample iff one exists. It follows that the hypothesis $\mathcal{H}$ can be built without ever calling the Solver (and with the same membership query complexity as MAT) and Validity queries correspond precisely to Equivalence queries.

4 $\mathtt{L}^\star_\square$: an iterative iMAT Learner

We now present an iMAT Learner, closely following the approach used in $\mathtt{L}^\star$. In essence, looking back at the schema in Figure 4 we want to devise a learner that exploits the loop of membership-validity queries to promote a steady construction of a minimum-size automaton with the minimum amount of information. For the Solver component of iMAT we use an SMT solver (i.e., Z3 [11]).

The Learner holds a (sorted, in increasing size) worklist of candidate observation tables with $\Box$'s. The algorithm works by constructing a worklist of potential hypotheses of increasing size, starting from a table with just ε as row and column labels.

We iteratively take each of these candidate tables and use an SMT solver to attempt filling in the $\Box$'s (Completion) so that the table is closed and distinct (see Figure 6). Based on the result of the SMT query, we will either pass the completed table to the Teacher (Validity) or try another candidate table. If the Validity query succeeds, we return it. Otherwise, if we get a counterexample c, we refine the current hypothesis table by adding all the suffixes of c to E, and add the table (with $\Box$'s) back to the worklist. The resulting iterative algorithm, $\mathtt{L}^\star$ with blanks ($\mathtt{L}^\star_\Box$), is shown in Figure 5.

Given an observation table, (S, E, T):

1. Declare a Boolean variable b_s for each string $s \in (S \cup S \cdot \Sigma) \cdot E$ for which $T(s) = \Box$.
2. For strings in L_+ or L_-, we fix the Boolean constraints:

$$\bigwedge_{s \in L_+} b_s = \text{true} \qquad \text{(Positive Evidence)}$$

$$\bigwedge_{s \in L_-} b_s = \text{false} \qquad \text{(Negative Evidence)}$$

3. Define a predicate to assert rows equal:

$$\text{Eq}(s, s') \triangleq \bigwedge_{e \in E} b_{se} = b_{s'e} \qquad \text{(Rows equal)}$$

4. Assert the table is closed:

$$\bigwedge_{s' \in S \cdot \Sigma - S} \bigvee_{s \in S} \text{Eq}(s, s') \qquad \text{(Closed)}$$

5. Define a predicate to represent the property that a row is unique (more precisely, that it is the first time the row appears):

$$\text{Unique}(s) \triangleq \bigwedge_{s' \in S:\ s' <_{\text{lex}} s} \neg\text{Eq}(s, s') \qquad \text{(Row unique)}$$

6. Declare a Boolean u_s for each string $s \in S$. We set u_s to true if row(s) is the first time that row appears:

$$u_s = \text{Unique}(s)$$

7. Assert the uniqueness of the rows in S (to maintain distinctness invariant):

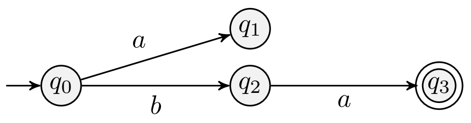
$$\bigwedge_{s \in S} \text{Unique}(s)$$

Figure 6 SAT encoding for table constraints

When the SMT solver finds that the instance is unsatisfiable, we know that the table cannot be made closed, but we do not know which s from $S \cdot \Sigma - S$ we need to promote to S to fix things as we would in classic $L^\star$. So we make progress by extending separate "copies" of S with each string in $S \cdot \Sigma - S$, and adding these tables to the worklist.

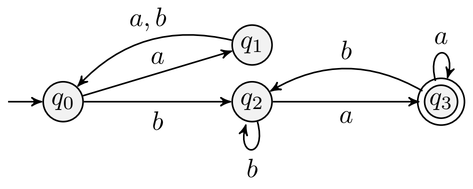
▶ Remark 5. By adopting the Maler-Pnueli variation of $L^\star$, the rows in S will always be distinct and correspond to nodes in a prefix tree. The worklist of tables with □'s is providing an enumeration of the search space of prefix trees that correspond to DFAs of increasing size. The SMT solver will essentially be checking for every entry of the worklist whether, for a given prefix tree corresponding to a table, the other transitions not in the tree can be filled in to be consistent with the data. For example, consider the following snapshot of a table in the worklist and the corresponding prefix tree:

	ε	a	aa	baa
ε	□	-	□	+
a	-	□	-	□
b	-	+	+	+
ba	+	+	+	□
aa	□	-	□	□
ab	□	-	□	□
baa	+	+	□	□
bab	-	+	□	□
bb	-	+	+	□



It turns out this table can be made closed and distinct—which indeed means the remaining transitions can be filled in. Here is the filled-in table and the resulting DFA:

	ε	a	aa	baa
ε	⊟	-	⊟	+
a	-	⊟	-	⊟
b	-	+	+	+
ba	+	+	+	⊞
aa	⊟	-	⊟	⊞
ab	⊟	-	⊟	⊞
baa	+	+	⊞	⊞
bab	-	+	⊞	⊞
bb	-	+	+	⊞



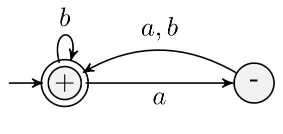
4.1 $L^\star_\Box$ Example

We illustrate the $L^\star_\Box$ algorithm (from Figure 5) with a teacher that starts with the following data: $L_+ = \{\epsilon, aa, ab\}$ and $L_- = \{a, bb, abb\}$. We begin with the following table on the left as the only item in the worklist.

	ε
ε	+
a	-
b	□

	ε
ε	+
a	-
ab	+
aa	+
b	□

	ε
ε	+
a	-
ab	+
aa	+
b	⊞



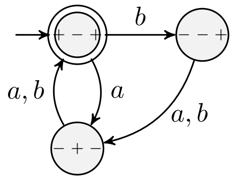
Next, we pop the table on the left off of the worklist and send it to the solver. We can see that even without filling in the blanks that the solver will return UNSAT (observe row(a) cannot possibly match row(ε)). So we add a to S and b to S and append the two new tables to the worklist (which was empty). We pop the middle table above off of the worklist and send it to the SMT solver. The solver finds that filling the single blank with a + satisfies the constraints, resulting in the table on the right. We conjecture the corresponding machine on the right and get counterexample bb. We then add the suffixes of bb to E, and enqueue the horizontally extended (S, E, T) to the head the worklist. Thus we again pop the updated table and attempt to fill it in:

	ε	b	bb
ε	+	$\square$	+
a	-	+	-
ab	+	-	$\square$
aa	+	$\square$	$\square$
b	$\square$	-	$\square$

	ε	b	ab
ε	+	-	+
b	-	-	$\square$
a	$\square$	+	$\square$
bb	-	$\square$	$\square$
ba	$\square$	$\square$	$\square$
ab	+	-	$\square$
aa	+	$\square$	$\square$

The table on the left is unsat. So we branch our search out by add ab, aa, and b to S on three new tables and append each of them to the worklist (which currently only holds the table with $S = \{\epsilon, b\}$). For brevity, we fast forward to the table from the last step where b was added, shown above on the right. We take the table off the worklist and the solver successfully fills in the blanks. We then conjecture the machine that corresponds to this table and this is correct on all data in L_+, L_- so the algorithm returns it.

	ε	b	ab
ε	+	-	+
b	-	-	$\boxplus$
a	$\boxminus$	+	$\boxminus$
bb	-	$\boxplus$	$\boxminus$
ba	$\boxminus$	$\boxplus$	$\boxminus$
ab	+	-	$\boxplus$
aa	+	$\boxminus$	$\boxplus$



The reader may have noticed that several times we naively added all row labels from the lower part of the table, when we could have localized the problem to a single row causing the unsat (we skipped ahead to these tables). This intuition leads to an optimization based on unsatisfiable cores, which we describe in Section 8.1.

5 Correctness of $L^\star_\square$

In this section, we show that the Learner $L^\star_\square$ we introduced in the previous section is correct: the algorithm terminates with the minimum size automaton compatible with L_+, L_- (Theorems 10 and 11). We first show that the search proceeds monotonically in the size of S (and therefore in the size of automata).

▶ **Lemma 6** (Size of work lists). *The worklists generated in Figure 5 can contain tables of at most 2 sizes, n and $n + 1$. When both are present, all the tables of size n are at the front.*

Proof. The initial worklist is $[(\{\varepsilon\}, \{\varepsilon\})]$. So there is one size, $|S| = 1$, and claim is trivially satisfied. As we enter the cycle with a worklist of size n in step (2) note that we add to the worklist in two places. In step $2(a)$, we add a list of items of size $n + 1$ so the property is maintained in both cases (it may be that the item we popped was the last one of size n, in which case the list is now a single size). In step $2(b)$, we process the counterexample and add the table back to the front of the worklist—since we do not modify S this is an item of size n and he property is maintained. ◀

The following definition expresses the notion that an observation table is "on the path" to identifying some target DFA, which will be useful shortly:

▶ **Definition 7** (Compatible Observation Table). *An observation table* (S, E, T) *is* compatible *with a DFA* $M = (Q, \Sigma, \delta, q_0, F)$ *if:*

1. *The function* $q \colon S \to Q$ *defined by* $q(s) = \hat{\delta}(q_0, s)$ *is injective,*
2. *The blanks can be filled in* by M *to make the rows of* S *distinct, and*
3. *For each* s *such that* $T(s) \neq \square$*, then* $T(s) = +$ *if and only if* $s \in L(M)$*. (i.e.* M *agrees with the non-blank entries of* (S, E, T)*).*

In other words, a table which is compatible with a target DFA can be extended to find the target DFA by adding zero or more strings to S while maintaining the invariant that each row corresponds to a unique state. The next lemma ensures that the worklist will always have some compatible table with a minimum DFA.

▶ **Lemma 8.** *Let* $M = (Q, \Sigma, \delta, q_0, F)$ *be a minimum-size DFA consistent with* L_+, L_-*. All worklists generated in the algorithm in Figure 5 contain at least one table* (S, E, T) *that is compatible with* M*.*

Proof. First observe that M has no unreachable states since it is minimal. Initially, the table $(\{\varepsilon\}, \{\varepsilon\})$ is compatible with M since (a) any function from $S = \{\varepsilon\}$ is injective and (b) there is only one row, so it is distinct. We now check that the algorithm maintains the invariant of having a compatible table in the worklist. As we enter the cycle with a worklist of length n, the only interesting case is when the compatible table (S, E, T) is the head of the list, since in all other cases it will clearly still be on the worklist at the next iteration. We proceed to analyse what happens while processing (S, E, T):

- In Step 2(a), we know that (S, E, T) could not be made made closed by filling in blanks, but by induction hypothesis that the rows of S are distinguished by M. Hence, for at least one $s \in S \cdot \Sigma - S$ we have that s accesses a new state in M (if this were not the case, then SAT instance could be satisfied using M as an oracle, since then each row in the lower part would match the row in the upper part corresponding to the state accessed). By the same argument, after adding this s to S, we know it will still be distinguishable from the other rows, because if it were not, then using M as an oracle would have satisifed the core. Hence, the table obtained by adding s to S is compatible with M.
- Step 2(b)(i) does not return, so we need not maintain the invariant.
- In Step 2(b)(ii), we do not change S, so that condition (a) of Definition 7 is immediate. Any rows which can be distinguished by M using only the suffixes in E are still distinguished by additional suffixes in E'. So (S, E', T') is compatible with M. ◀

We now prove a lemma which illustrates that the crux of the search is finding the correct prefix set, S. At this point, we will make a validity query (possibly incorrectly—but the table will be satisfiable) until we are correct. If we have a particular oracle in mind (i.e., any correct automaton), then the lemma says that if S contains accessor prefixes for the states of the oracle (and E is large enough to distinguish states), the table will be satisfiable. To be clear, in the algorithm we course do not use an oracle to fill in the blanks—we use an SMT solver as we do not yet know the automaton. The point is simply that the satisfiability of the SMT instance when a solution exists will be used to establish correctness.

▶ **Lemma 9** (Existence of a solution). *Let* L_+, L_- *be example sets, and let* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *be a minimum state DFA consistent with* L_+, L_-*. Let* (S, E, T) *be an observation table that is compatible with* $\mathcal{A}$ *and such that* $|S| = |Q|$*. Then there is a closed, distinct, complete observation table* (S, E, T') *where* $T'(s) = T(s)$ *for all* $T(s) \neq \square$*.*

Proof. Use the DFA $\mathcal{A}$ as an oracle to complete the table: let $T'(s) = +$ if $s \in L(\mathcal{A})$, otherwise $T'(s) = -$. Since $\mathcal{A}$ is consistent with the datasets, so $T(s) = T'(s)$ whenever $T(s) \neq \Box$.

(closed) Let $s \in S, a \in \Sigma$. If $sa \in S$, we are done, so assume $sa \notin S$. Let $q = \hat{\delta}(q_0, sa)$. By hypothesis, there is a string $s' \in S$ such that $\hat{\delta}(q_0, s') = q$. Then for each $e \in E$, $T'(sae) = \hat{\delta}(q_0, sae) = \hat{\delta}(\hat{\delta}(q_0, sa), e) = \hat{\delta}(q, e) = \hat{\delta}(\hat{\delta}(q_0, s'), e) = \hat{\delta}(q_0, s'e) = T'(s'e)$. Thus $\text{row}(sa) = \text{row}(s')$.

(distinct) By the fact that (S, E, T) is compatible with $\mathcal{A}$, we have that using $\mathcal{A}$ to fill in the blanks of T distinguishes all of the rows labeled by S. ◀

▶ **Theorem 10** (Partial Correctness). *If the algorithm terminates, then it returns a DFA of minimal size consistent with L_+, L_-.*

Proof. Let M be any minimal DFA for L_+, L_+. Then by Lemma 8, at each point there is an observation table on the worklist that is compatible with M. Assume the algorithm eventually terminates. In the worst case, it is when we find the table (S, E, T) with $|S| = n$, for n the number of states of M, (and potentially with $E = \text{suffixes}(L_+ \cup L_-)$). By Lemma 6, we traverse the worklist in nondecreasing order. Hence, if we terminated earlier, it was by conjecturing a correct machine smaller than n, which is a contradiction. ◀

All that remains to be shown is that we make progress towards this solution and indeed the algorithm terminates.

▶ **Theorem 11** (Termination). *The $\mathtt{L}^\star_\Box$ algorithm terminates.*

Proof. Let M be a minimal DFA for L_+, L_-, and let (S, E, T) be a table in the worklist compatible with M guaranteed to exist by Lemma 8. Step $2(a)$ makes progress by increasing the size of S. Moreover, since there are finitely many automata M' whose size is less than or equal to M, there are only finitely many counterexamples we can get in Step $3(b)$ (because we never again misclassify an example we have seen). Each time we reach the compatible table, we enqueue a table with larger size by 1. When, at latest, we reach the compatible table whose size matches M, then we may need many, but only finitely many more validity queries before we are correct and terminate. ◀

6 Weakening the Teacher: iMAT with Distinguish

Next, we explore another kind of incomplete teacher that is not required to implement Validity queries, but only a weaker set of Distinguish queries. Two strings $s_1, s_2 \in \Sigma^*$ are said to be *distinguished* if there exists $e \in \Sigma^*$ such that,

- $s_1 e \in L_+$ and $s_2 e \notin L_+$ (or vice versa), or
- $s_1 e \in L_-$ and $s_2 e \notin L_-$ (or vice versa).

Membership: As before, the Learner sends a string u and the Teacher responds with "+" if $u \in L_+$, "−" if $u \in L_-$, and "□" otherwise.

Distinguish: The learner sends two strings s_1 and s_2, together with a finite "exclusion set" E, and the teacher responds "yes" with a suffix $e \notin E$ that distinguishes these two strings or "no" if it cannot distinguish the strings.

It should be clear that Distinguish queries are less powerful than Validity queries, as they only concern a pair of strings and not the full language. However, it turns out that when L_+ and L_- are finite, we can adapt the learner to still have a terminating procedure to learn a correct automaton. We prove these results in Section 7.

iMAT with Distinguish has close ties to foundational concepts in formal languages. In particular, the distinguish query is closely related to the Myhill-Nerode equivalence classes $\equiv_L$ of the target language L.

▶ **Lemma 12.** *Given strings $s_1, s_2 \in \Sigma^*$ with distinguish $\varnothing$ s_1 $s_2 = e$ for some $e \in \Sigma^*$, then there exists some language L that separates L_+ and L_- such that $s_1 \not\equiv_L s_2$. Moreover if, in addition, the result of membership queries to s_1e and s_2e are not $\square$, then for* every *L separating L_+ and L_-, we have that $s_1 \not\equiv_L s_2$.*

Proof. For the second claim, suppose $T(s_1e) \neq T(s_2e)$ and both are not $\square$. Without loss of generality, $T(s_1e) = +$ and $T(s_2e) = -$. Let L be any language separating L_+ and L_-. Clearly $s_1e \in L$ and $s_2e \notin L$. That $s_1 \not\equiv_L s_2$ follows from the definition of the Myhill-Nerode equivalence. Now for the first part, assume $T(s_1e) = -$ and $T(s_2e) = \square$ (the other cases are similar), and assume L is a regular language that separates L_+ and L_-. Then $L' = L \cup \{s_2e\}$ is also regular and separates L_+ and L_-. Moreover, for this L we have $s_1 \not\equiv_L s_2$. ◀

Although iMAT with Distinguish is guaranteed to terminate only when L_+ and L_- are finite, we can show that arbitrary problem instances can be modeled using finite instances. Hence, finite example sets are in a sense complete. More formally, suppose we are given possibly infinite sets of positive and negative examples L_+ and L_-. Then we can find finite subsets of L_+ and L_- that are sufficient for the Learner to recover the minimum-size DFA compatible with L_+ and L_-.

▶ **Theorem 13.** *Let $L_+, L_- \subseteq \Sigma^*$ be infinite, disjoint example sets (not necessarily regular), and let M be a minimum-size DFA that separates L_+ and L_-. Then there are* finite *subsets L_+, L_- such that M is also a minimum-size DFA separating L_+, L_-.*

Proof. Consider the set $\mathcal{M}$ of all DFAs over Σ that have strictly fewer states than M. (Observe that $\mathcal{M}$ is potentially very large, but finite). By the minimality of M on L_+, L_-, each $M' \in \mathcal{M}$ misclassifies a string from either L_+ or L_-. So we can define a function $c\colon \mathcal{M} \to \Sigma^*$ that maps each M' to one such counterexample string. Then the set $S = \{s \mid s = c(M'), M' \in \mathcal{M}\}$ is finite (in fact, $|S| \leq |\mathcal{M}|$). Moreover, we have by construction that for $S_+ = L_+ \cap S$ and $S_- = L_- \cap S$ that each $M' \in \mathcal{M}$ fails to separate S_+, S_- by misclassifying $c(M')$, so that M is a minimum-size separating automaton for S_+, S_- as needed. ◀

7 Correctness of $\mathtt{L}^\star_\square$ with Distinguish

We now show how to adapt the $\mathtt{L}^\star_\square$ to the situation that we only have an iMAT with Distinguish, not Validity. The idea is that we perform $\mathtt{L}^\star_\square$ as before, but when we have a hypothesis machine, we do a series of distinguish queries instead of a validity query. Intuitively, we will ask a distinguish query for each pair of rows which are "made the same" by the hypothesis:

▶ **Definition 14** (Distinguish queries for a hypothesis). *Given a closed, distinct, complete table, (S, E, T) and associated DFA M, we say the* distinguish queries for hypothesis *M are:*

distinguish E s' s

for each $s' \in S \cdot \Sigma - S$ and for the unique s such that $row(s) = row(s')$.

Observe these distinguish queries are well-defined because each $s' \in S \cdot \Sigma - S$ is guaranteed to have a unique matching row in S.

The first theorem shows that if all of the distinguish queries for a hypothesis return None, then the hypothesis is correct.

▶ **Theorem 15** (Validity using Distinguish). *Suppose a learner has a hypothesis DFA $M = (Q, \Sigma, \delta, q_0, F)$ constructed from a closed, distinct, complete observation table (S, E, T). Suppose also that the teacher returns None for all of the distinguish queries for M. Then $L_+ \subseteq L(M)$ and $L(M) \cap L_- = \varnothing$.*

Proof. We show the contrapositive. Suppose there is a counterexample string $c \in L_+$ such that $c \notin L(M)$ (without loss of generality; the other case is similar). It suffices to show that there is a distinguish query that is not None. First of all c cannot be in T since we inherit from $\mathtt{L}^\star$ that hypotheses are correct on all evidence already considered. So we must have $c = s'_0 v_0$ for a prefix $s'_0 \in S \cdot \Sigma - S$, since $S \cdot \Sigma - S$ includes exactly one prefix of each word not in $S\Sigma$. By closedness of the table, there is an $s_0 \in S$ such that $\mathrm{row}(s'_0) = \mathrm{row}(s_0)$. Consider distinguish E s_0 s'_0. If this query returns a suffix, we are done, so assume it is None. In particular, this means that $s_0 v_0 \in L_+$, since otherwise v_0 would be a response to the query. Moreover, since $\mathrm{row}(s_0) = \mathrm{row}(s'_0)$ we must have that $\hat{\delta}(\mathrm{row}(s_0), v_0) = \hat{\delta}(\mathrm{row}(s'_0), v_0)$, implying that $s_0 v_0 \notin L(M)$ and $s_0 v_0$ is also a counterexample. So, as before, there must be an $s'_1 \in S \cdot \Sigma - S$ and suffix v_1 such that $s_0 v_0 = s'_1 v_1$. Crucially, s_0 (as $s_0 \in S$) is a proper prefix of s'_1, making v_1 a proper suffix of v_0. Since, again, there must be an $s_1 \in S$ for which $\mathrm{row}(s_1) = \mathrm{row}(s'_1)$ We proceed by considering the query distinguish E s_1 s'_1 and applying the same reasoning, except that we have made progress because v_1 is a suffix of v_0. Clearly we will see a distinguishing suffix after a finite number of these iterations: in particular by the time that $v_i = \varepsilon$, then $s'_i v_i = s'_i \in S \cdot \Sigma - S$. But that means s'_i is a string we have already seen and is also a counterexample (i.e., $s'_i \in L_+$ but rejected), which is impossible by the construction of the table (which means the distinguishing suffix must be at latest the previous round). ◀

We modify $\mathtt{L}^\star_\square$ as follows:

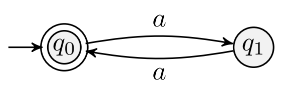
1. Run $\mathtt{L}^\star_\square$ until ready to make a validity query for hypothesis H.
2. In place of the validity query, make the distinguish queries for H.
   a. If they are all None, stop and return H.
   b. Otherwise, add the first suffix returned by the teacher to E, and return to Step 1.

▶ **Corollary 16.** *If the modified $\mathtt{L}^\star_\square$ terminates, the learned machine is correct.*

Proof. This follows immediately from Theorem 15. ◀

Of course, this modified procedure is not guaranteed to terminate in general. For some sparse, infinite L_+ and L_-, it is possible for the teacher to have infinitely many suffixes that the learner must consider without making progress. For example, if we have $L_+ = (aa)^*$ and $L_- = (aaaa)^* a$, the learner will soon reach:

	ε
ε	+
a	-
aa	+



At this point, the Learner will ask the query distinguish E ε aa and the trouble is that the teacher has an infinite set of suffixes to give in response to distinguish E ε aa, namely those strings in the set $(aa)^* a$. As a result, the procedure above will not terminate in this case. We are unaware of a query weaker than Validity that still guarantees termination—settling this is an interesting question for future work.

We do have, however, as a special case, guaranteed termination for finite example sets.

▶ **Corollary 17.** *Let L_+ and L_- be finite, disjoint sets of strings. A learner can still learn a correct automaton for L_+, L_- with only* membership *and* distinguish *queries.*

Proof. Run $\mathtt{L}^\star_\square$ modified for distinguish. Because of Theorem 15, we need only show that it terminates. Each time through Step 2 that we do not terminate, the teacher returns a suffix e. But since we add e to E, the teacher can never return this suffix again, and since the examples are finite, they can only have finitely many suffixes. Termination follows. ◀

7.1 A Landscape of Teacher Queries

As we have already presented two variants of iMAT, it is natural to ask how it relates to other formulations of automata learning problems. In this section, we summarize some known results for a variety of queries provided by a Teacher including:

Query name	**Type**
membership	$\Sigma^* \to 2$
equivalence	$\text{DFA} \to \Sigma^* + 1$
membership$_\square$	$\Sigma^* \to \{+, -, \square\}$
distinguish	$\mathcal{P}_{\text{fin}}(\Sigma^*) \times \Sigma^* \times \Sigma^* \to \{+, -, \square\}$
validity	$\text{DFA} \to \Sigma^* + 1$

Recall that though equivalence and validity have the same type, they are provided by Teachers with different capabilities: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns "yes" whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly, which is what a complete teacher proves through an Equivalence query.

We distinguish two classes of learning problems. For the first class, the Teacher knows a specific regular language and will not return "unknown" in response to Membership queries. Conversely, in the second class, the teacher is incomplete and may return "unknown" in response to Membership queries.

Teacher holds concrete regular language (i.e., no blanks)

Only Membership. Suppose we run $\mathtt{L}^\star$, but replace each equivalence query with a loop that queries all strings looking for a counterexample. It is a consequence of the Myhill-Nerode Theorem that this process will eventually discover the correct automaton. However, the Learner will never be able to determine that it has converged. If the Learner is told the number of states, a terminating procedure is possible [1].

Only Equivalence. We can obtain a trivial terminating procedure with only the equivalence query simply by conjecturing all automata in increasing order of states. Angluin [3] showed that we cannot improve the learner to a polynomial time procedure.

Membership and Equivalence. This is Angluin's MAT: $\mathtt{L}^\star$ is an efficient terminating procedure for determining the correct DFA [2]. See Section 2.

Teacher is incomplete (i.e., may return blanks)

Only Membership. Just as in the non-blanks case, a learner can identify a minimum-size correct machine eventually, but cannot determine that it has done so—so there is no terminating procedure. This is Gold's notion of "identification in the limit" [13].

Membership and Validity. Because this problem subsumes all instances of DFA inference from finite data, it is NP-Hard [15] and cannot be approximated in polynomial time (unless $P = NP$) [32]. We give a terminating procedure, $\mathtt{L}^\star_\square$, in Section 4 and prove it correct in Section 5. The complexity comments from "Only Equivalence" also hold here.

Membership and Distinguish. We consider a modification of $\mathtt{L}^\star_\square$ which produces a minimum-size automaton if it terminates. We explore this in Section 6.

8 Implementation

We have built an OCaml implementation of the learner (using Validity) presented in this paper. The goal of our implementation is to provide a reusable implementation of our framework in a functional language that can be used to explore the iMAT framework, run our algorithm, and implement other algorithms that use similar primitives. Our implementation consists of two packages: (i) `nerode`, a general library for regular languages, with data structures for regular expressions, DFAs, and NFAs, as well as conversions between them, and (ii) `nerode-learn`, which implements Angluin's $\mathtt{L}^\star$ algorithm, as well as the iMAT variants developed in this paper. The `nerode` package is approximately 2,400 lines of code, while the `nerode-learn` package is approximately 3,000 lines of code. We use Z3 as the backend solver for checking all SMT problems.

Data Structures

Our OCaml implementation uses the following types.

- Strings are encoded in a module called `Word`, where their representation is a list of Alphabet symbols. Alphabet symbols internally are `int`.[4]

```
type Word.t = Alphabet.symbol list
```

  In particular, defining modules for Word and Alphabet allowed for a modular design and allows us to support different alphabets.
- The rows of the observation table (or more precisely, the labels of the rows) are encoded as a set of Words:

```
type RowLabels.t = WordSet.t
```

- The columns of the observation table are also encoded as a set of words:

```
type ColLabels.t = WordSet.t
```

- Individual entries in the observation table are encoded using the following data type:

```
type entry = True | False | Blank
```

  Here, the constructors include `Plus`, `Minus`, and `Blank`.
- Thus we can encode the actual lookup function of the table as a map from words to entries:

```
type TblMap.t = entry WordMap.t
```

[4] We abuse OCaml syntax slightly: `type Word.t` is not a valid type definition, but we keep the module names to show our module boundaries, and concisely differentiate all of our types `t`.

Putting these all together, an observation table is encoded as a record s, sa, e, t where s, sa, and e are sets of strings, and *tmap* is a map from strings to entries:

```
    type Table.t = {s : RowLabels.t; sa : RowLabels.t;
                    e : ColLabels.t; tmap : TblMap.t}
```

When implementing algorithms on observation tables, there is an interesting data structure choice to be made. An obvious approach is to keep the entire table explicitly as a two-dimensional array. The advantage to this approach is that the row function corresponds precisely to rows in the array. The downside is that to update the entry for a string, one has to visit each location in the table corresponding to how the string can be decomposed into prefix and suffix. To avoid this, instead of keeping the whole table, we keep S, $S \cdot \Sigma - S$, and E as sets and T as a map from strings in $(S \cup S \cdot \Sigma) \cdot E$ to table entries. Thus entries for a string are updated everywhere they appear by a single update to T. The downside is that we must generate the row function "on the fly" by repeatedly accessing T.

Algorithms

We now present our OCaml implementation of $\mathtt{L}^{\star}_{\Box}$. We encode the worklist as a list of tables, which is maintained as an argument in the main recursive loop. A collection of all entries in the observation tables is also maintained globally as an argument in the loop.

```
let wl : Table.t list = ...
let entry_map : entry WordMap.t = ...
```

The main algorithm is a recursive function that searches for the smallest table that is compatible with the given positive and negative examples:

```
let rec lstar_blanks wl (e: ColLabels.t) entry_map: Dfa.t =
  let t, entry_map = List.hd wl |> Table.extend_cols entry_map e in
  match fill_blanks_smt t with
  | None →
    let new_ts, entry_map' =
        List.fold
            ~f:(fun (ts_acc, em_acc) sa →
                let new_t, em_acc' = (Table.move_up em_acc sa t) in
                new_t::ts_acc, em_acc')
            ~init:([], entry_map) (Table.lower_rows t) in
    lstar_blanks (wl @ new_ts |> List.tl) e entry_map'
  | Some obs →
    let dfa = table_to_dfa obs in
    match conjecture dfa with
    | None → dfa
    | Some cex →
        lstar_blanks wl (ColLabels.union e (suffixes cex)) entry_map
```

For the most part, the OCaml implementation follows the pseudocode given in Figure 5, but there are a few differences. In particular, the columns of the tables (`e`) in the worklist are not updated until right before we attempt to fill them in. We keep a cumulative mapping from strings to table entries (`entry_map`) in the main loop, preventing redundant membership queries, and we also accumulate the suffixes learned from counterexamples (`e`) in the main loop. Intuitively, this approach is sound because it is safe to share counterexamples across tables and thus have the same column labels (`e`) for each (see Lemma 19 below).

Operationally, the `lstar_blanks` function proceeds as follows. First, it removes an item t from the worklist. It adds additional suffixes in e to the table t, using membership queries to determine new column entries in e that cannot be found in *entry_map*, filling in t with `Plus`, `Minus`, or `Blank`; it also updates the entry collection *entry_map* with any new queries. Next, it attempts to fill the blanks in the observation table encoded by t, calling `fill_blanks_smt`, which uses use an SMT solver as discussed below. If the SMT solver cannot find values for the blanks, the algorithm extends the worklist with each new table *new_t* obtained by adding one row from the bottom part of the table t, using `Table.add`, and recurses. It also updates the *entry_map* with any new query results while filling in new rows. Otherwise, the blanks can be filled, and it makes a conjecture to the teacher. If the conjecture is correct, we have the minimal DFA. Otherwise, the conjecture fails, so we add the suffixes for the new counterexample to e, keep t at the head of `wl`, and recurse.

Efficient SMT Encoding

The key to achieving good performance in any solver-aided algorithm, is having an efficient encoding of observation tables. We use the theory of bitvectors, which is widely supported by SMT solvers, to express the blank-filling constraints. Specifically, given a table with k columns, we encode the rows in the table as bitvector variables of length k. We add assertions to constrain the bitvectors so they correspond to the rows in the table[5]. For example, the bitvector variable for the jth row in the table would be declared as follows,

```
(declare-const s_j (_ BitVec k))
```

and if the entry in the ith column was positive, we would add an assertion:

```
(assert (= ((_ extract i i) s_j) #b1))
```

Alternatively, if the entry in the ith column were a blank, we would declare a bitvector variable of length 1 to encode the blank,

```
(declare-const b_ij (_ BitVec 1))
```

and add the corresponding constraint:

```
(assert (= ((_ extract i i) s_j) b_ij))
```

Hence, a model of the SMT formula corresponds to a way of filling in the blanks that is consistent with the positive and negative entries in the table.

Returning to our algorithm, recall that we need the rows in the upper part of the table to be distinct, and we need the table to be closed. For distinctness, we use the built-in `distinct` function from the bitvector theory. For closedness, we generate assertions stating that each row in the bottom part of the table must be equal to some row in the upper part.

For example, given the observation table on the left:

	ε	a
ε	+	□
a	□	-
b	□	+
aa	-	□
ab	□	□
ba	+	-
bb	□	□

$\Longrightarrow$

	ε	a
ε	+	⊟
a	⊟	-
b	⊞	+
aa	-	⊟
ab	⊞	⊟
ba	+	-
bb	⊞	⊟

Our SMT encoding would have 7 bitvectors of size 2, one for each row, and 7 bitvectors of size 1, one for each (unique) blank:

[5] An earlier version used SAT, with individual entries as boolean variables, but we found that using the bitvector theory performed better

```
(declare-const s1 (_ BitVec 2)) (declare-const b1 (_ BitVec 1))
(declare-const s2 (_ BitVec 2)) (declare-const b2 (_ BitVec 1))
(declare-const s3 (_ BitVec 2)) (declare-const b3 (_ BitVec 1))
(declare-const s4 (_ BitVec 2)) (declare-const b4 (_ BitVec 1))
(declare-const s5 (_ BitVec 2)) (declare-const b5 (_ BitVec 1))
(declare-const s6 (_ BitVec 2)) (declare-const b6 (_ BitVec 1))
(declare-const s7 (_ BitVec 2)) (declare-const b7 (_ BitVec 1))
```

Note that in the table above, the blank in the first row, second column and the blank in the second row, first column must have the same value as they both encode the membership of the string a in the language. Our SMT encoding account for such constraints between blanks. Next, we add assertions to encode the entries in the table:

```
(assert(=((_ extract 1 1) s1) #b1)) (assert(=((_ extract 2 2) s1) b1))
(assert(=((_ extract 1 1) s2) b1))  (assert(=((_ extract 2 2) s2) #b0))
(assert(=((_ extract 1 1) s3) b2))  (assert(=((_ extract 2 2) s3) #b1))
(assert(=((_ extract 1 1) s4) #b0)) (assert(=((_ extract 2 2) s4) b3))
(assert(=((_ extract 1 1) s5) b4))  (assert(=((_ extract 2 2) s5) b5))
(assert(=((_ extract 1 1) s6) #b1)) (assert(=((_ extract 2 2) s6) #b0))
(assert(=((_ extract 1 1) s7) b6))  (assert(=((_ extract 2 2) s7) b7))
```

Note that the assertions reflect the fact the blanks in the first rows must be filled in with the same value, since they are both associated with the string “`a`”. We would also assert distinctness for the upper part,

```
(assert(distinct s1 s2 s3))
```

Finally, we would add assertions for closedness:

```
(assert(or (= s1 s4) (= s2 s4) (= s3 s4)))
(assert(or (= s1 s5) (= s2 s5) (= s3 s5)))
(assert(or (= s1 s6) (= s2 s6) (= s3 s6)))
(assert(or (= s1 s7) (= s2 s7) (= s3 s7)))
```

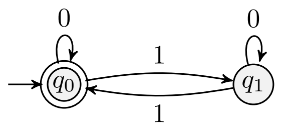
To fill in the rows in table, we can simply read off values assigned to the variables,

```
(define-fun b1 () (_ BitVec 1) #b0)
(define-fun b2 () (_ BitVec 1) #b1)
(define-fun b3 () (_ BitVec 1) #b0)
(define-fun b4 () (_ BitVec 1) #b1)
(define-fun b5 () (_ BitVec 1) #b0)
(define-fun b6 () (_ BitVec 1) #b1)
(define-fun b7 () (_ BitVec 1) #b0)
```

which corresponds to the table on the right above.

Teacher Module

Because our design relied on an abstract interface to the teacher, it was straightforward to implement the teacher in different ways. While the benchmarks we used for evaluation (Section 9) are lists of labeled examples, we also implemented a feature that allows the user to specify infinite L_+ and L_- using regular expressions. For example, we could specify $L_+ = (0101)^*$ and $L_- = 10^*$, and the system correctly identifies:



Note that the correct language has a smaller DFA than the one for L_+!

8.1 Optimizations

We now look at several optimizations we implemented. The first one will reduce the number of tables generated in step 2(a) of Figure 5 by making use of unsat-cores; the second prevents the learner from making needlessly similar conjectures by reusing counterexamples that it gets from the teacher, and the third explores the use of a priority queue instead of a list.

Unsat-cores

When a table's blanks cannot be filled in by the SMT solver we explore the tables that result by adding each string from $S \cdot \Sigma - S$ to S. In many cases, however, we might be able to identify a row, say $s' \in S \cdot \Sigma - S$ that is already distinguished from every row in S by suffixes in E. In this case, any filling of the blanks would result in an unclosed table *because of this row*, so we can promote *only* this one, as in classic $\mathtt{L}^\star$.

To generalize this, we observe that the closedness assertion corresponding to the row s' above was unsatisfiable on its own. Modern SMT solvers can be configured to compute subets of constraints that are unsatisfiable alone, called unsat cores, when returning "unsat" [26]. To justify that we are still guaranteed to find the minimal automaton boils down to extending the argument of Lemma 8:

▶ **Lemma 18.** *The invariant of Lemma 8 holds, even when $\mathtt{L}^\star_\square$ is modified to add only tables resulting from promoting only strings in unsat core.*

Proof. The argument for Lemma 8 applies, but we need to justify that a string in the unsat core accesses a new state of M. This must be true, or else using M as an oracle would fill in the blanks to satisfy the unsat core. ◀

Suffix-set Sharing

The reader may have noticed that the $\mathtt{L}^\star_\square$ Algorithm described above cleverly reuses counterexamples, accumulating the suffix set E. Hence, instead of maintaining a worklist with elements (S, E, T), we only need a worklist that maintains prefix sets S for each individual table. For T, we maintain a cumulative mapping of strings to entries as a parameter in the main loop and simply update missing entries for columns of the popped table on each iteration. For E, we prove that sharing counterexamples is sound:

▶ **Lemma 19.** *Let M be a DFA, and let (S, E, T) be compatible with M and let (S, E', T') be a table such that $E \subseteq E'$. Then (S, E', T') is also compatible with M, where T' is the extension of T by membership queries for E'.*

Proof. Condition (a) of compatibility is immediate since S is unchanged. If M distinguishes the upper part rows with only the suffixes in E, they remain distinguished by E', implying condition (b). Condition (c) holds by hypothesis. ◀

▶ **Corollary 20.** *The algorithm in Figure 5 is still guaranteed to produce minimal machines when modified to share a single E across the worklist.*

Proof. Sharing E means instead of adding a counterexample to the current item's E, traverse the worklist updating all E. By Lemma 19 adding additional strings to E does not affect compatibility, and so the invariant of Lemma 8 still holds and minimality follows. ◀

Benchmark set	DFA Size	# Benchmarks	Mean Learn time (s)	Median Learn time (s)	Mean Z3 Time (s)	Mean worklist items
Lee, So, and Oh	2	1	0.0082	0.0082	0.0081	2.0000
	3	10	0.0234	0.0229	0.0226	5.0000
	4	9	0.0600	0.0417	0.0529	7.5556
	5	4	0.1676	0.1360	0.1401	19.0000
Oliveira and Silva	1	80	0.0056	0.0053	0.0055	1.0000
	2	15	0.0100	0.0093	0.0097	2.0667
	3	91	0.0226	0.0216	0.0213	4.0879
	4	84	0.0396	0.0338	0.0346	5.7619
	5	100	0.1237	0.0775	0.0935	8.6200
	6	105	0.3803	0.1684	0.2566	13.6381
	7	79	1.1251	0.6419	0.7583	20.0886
	8	93	10.6307	1.7830	6.1782	44.1613
	9	74	50.1672	7.7361	28.8381	96.8784
	10	25	98.0573	7.3782	65.2680	176.5200
	11	14	1498.4836	101.2503	988.9716	933.2857

Table 1 Performance results on established benchmarks. "Learn time" is the total time spent on an individual benchmark, while "Z3 time" is the time spent on a benchmark *within the SMT solver*. "Worklist items" are the number of tables processed from the worklist during learning.

Heuristic prioritization

Our algorithm as described above uses a list that is sorted only by the size of the prefix set. By instead using a priority queue, we can apply heuristics to investigate tables which are more likely to be compatible with a minimal DFA (in the sense of Definition 7). Because we search monotonically by size, this trick can only save effort on the "last size searched," since all smaller automata are checked first. Still, the number of prefix sets grows rapidly enough that the savings on even this last size justify the optimization.

9 Evaluation

To evaluate the performance of our implementation, we timed it on a portion of the benchmark sets created by Oliveira and Silva [29] (for their system Bica) and the benchmarks of Lee, So, and Oh[6] [23]. We present summary data of these runs in Table 1 and Figure 7. Table 1 shows that median total learning times were consistently shorter than the mean total learning times, suggesting that, at each size, the more expensive examples are less common. The mean Z3 time column suggests the system spends around two-thirds of its running time in SMT solving at all sizes. The last column shows how the number of worklist items processed increases with DFA solution size. All benchmarks use the alphabet $\Sigma = \{0, 1\}$. We ran the experiments on a 2.10GHz Intel Xeon Silver 4216 machine with 512GB of memory.

Oliveira and Silva's benchmark set is large: for each size from 4 states to 23 states, they produced 19 random DFAs. For each DFA, there are 5 sets of positive and negative example strings. Each example set is a set of 20 strings of length 30 produced by random walks on the generated DFA. Each problem also contains all the prefixes of those 20 strings.

[6] We omit benchmark #9, which has a DFA solution of size 16.

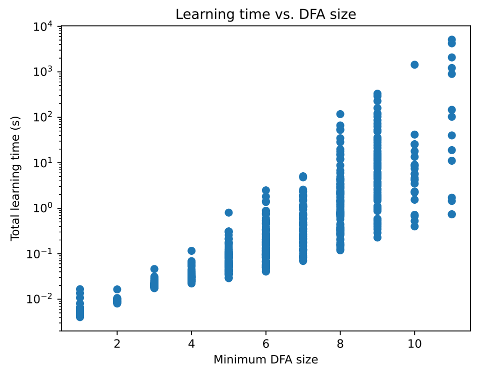


Figure 7 Performance on Oliveira and Silva benchmarks generated by automata of sizes 4 to 11.

Using this process, they generated a total of 95 problems using each size of random DFA. Importantly, as a result of this generation process, it is very often the case that there is a smaller consistent machine than the one used in generation. In Table 1, summary data is presented for benchmarks generated from size 4 to size 11 machines, however, note that for Table 1 and Figure 8, the size shown is the size of the *learned* (i.e., minimum-size) automaton, not the one used to generate the examples.

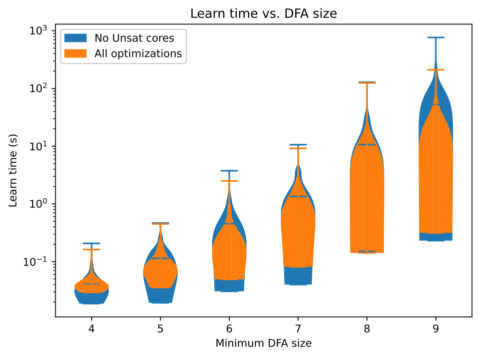
From the results of our experiments, we conclude that the scalability of our system is limited. It is practical in cases where the DFA to be learned is of size 11 or smaller. However, in its current form, learning larger DFAs is prohibitively expensive (see Section 11 for a discussion of future work in this direction). In contrast, Heule and Verwer [17] report solving all of the Oliveira and Silva benchmarks for sizes 4 to 21 "within 200 seconds per instance," although their method requires access to all of the finite examples at the start.

9.1 Evaluation of optimizations

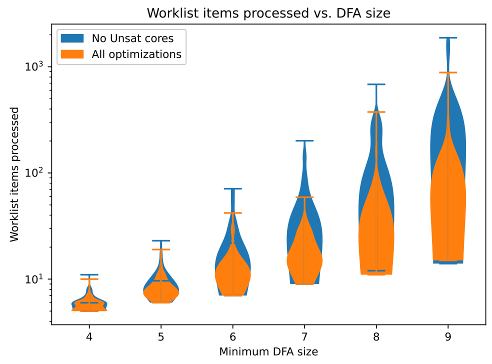
We evaluated the optimizations from Section 8.1 by conducting an ablation-style study, presented in Figure 8. The ablation study was performed in two stages. First, we ran the system on the Oliveira benchmarks (generated by sizes 4 to 9) with all optimizations turned on. Then, we ran the system on the same benchmarks a separate time for each of the three optimizations, with the optimization turned off. The results show significant time and search space savings for the unsat-cores optimization and heuristic prioritization. The "suffix-set sharing" optimization cannot affect the number of worklist items processed and turned out to result in an entirely negligible time improvement (these figures are omitted).

10 Related Work

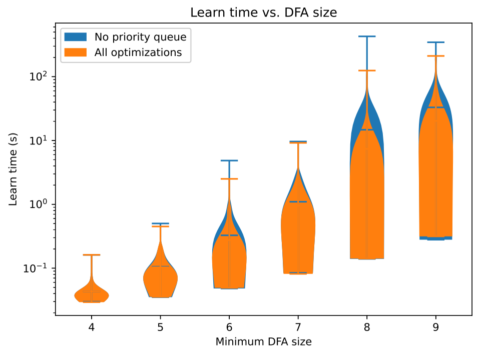
DFA Inference from finite data. Inference of DFAs from finite data is a long standing problem and different solutions have appeared in the literature (see e.g [10] for an overview). Gold introduced the observation table and considered blank entries ("holes") in the context of passive learning, but his algorithm does not guarantee minimality of the automaton produced [14]. Modern algorithms attack the problem from the perspective of "state merging" [30]. The main idea of these approaches is to build an automaton from the tree of all prefixes of positive examples, called the prefix tree accepter (PTA), and then attempt to merge states, using the negative examples to validate merges. It is invariant that the positive examples are



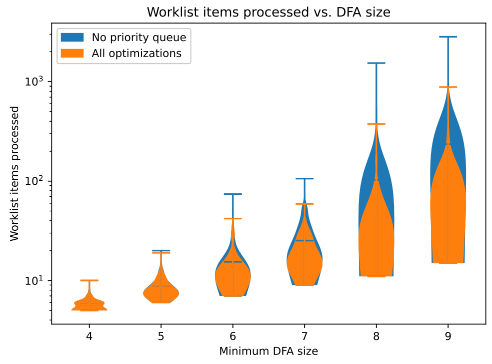
(a) Time reduction from unsat core optimization.



(b) Search space reduction of unsat core optimization.



(c) Time reduction from heuristic prioritization.



(d) Search space reduction of heuristic prioritization.

Figure 8 Ablation study of optimizations proposed in Section 8.1. Data for "Suffix-set sharing" is omitted because it produced only negligible speedup.

accepted, but a merge might cause a negative example to be accepted—in which case it is backtracked. Much research has gone into the selection of which merges to prioritize [21, 9]. State merging methods are much faster than the algorithm described in this paper but in general do not guarantee minimum size. A notable exception to this is the **exbar** Algorithm due to Lang [20], which gives a method for exhaustively exploring potential merges in a way that finding a minimum size DFA is guaranteed at exponential running time cost.

Inference of Separating Automata. Chen et al [8] use a similar notion of blanks in $\mathtt{L}^\star$ (which they call "Don't care"). The problem they solve is closely related, but it requires that the teacher start with two *regular* languages, for which they seek to find a minimal separating DFA. The algorithm they present, L^{sep}, computes the final DFA by first computing a 3-valued automaton (3DFA), which has "Don't care" as output. They then convert this automaton to the minimum-size consistent DFA. Crucially, the interface they define with the teacher requires that the positive and negative example sets be regular languages. Thus in light of this restriction, the problem solved by L^{sep} may be seen as a special case of the iMAT setting.

DFA Inference using SAT solvers. Oliveira and Silva use constraint solvers to find a mapping from the states of a PTA to a particular size [29]. If the search fails, the size is increased until a DFA is found. Their work extends the method of Biermann and Feldman [6], who first explored such mappings from the PTA. Heule and Verwer [17] also use SAT solvers to infer DFAs from examples but their approach uses a SAT formulation closely tied to graph coloring. They first build a compatibility graph for states of the PTA, where states are compatible if their merge is not immediately ruled out by examples. The colors assigned in the coloring instance thus correspond to states in the smaller DFA, with the edges of the incompatibility graph ruling out incorrect merges.

Active learning of Network Invariants. Grinchtein, Leucker, and Piterman [16] present an algorithm that augments $\mathtt{L}^\star$ to handle missing information with a SAT solver for inferring network invariants—i.e., in the sense of Wolper and Lovinfosse's inductive technique verifying for large compositions of finite-state systems [35]. Their work, which improves the earlier method of Pena and Oliveira [31], extends the Angluin notions of table closedness and consistency to tables with blanks (called "weakly closed" and "weakly consistent"). The learner can proceed by performing $\mathtt{L}^\star$ corresponding actions on tables while there are still blanks. Once the table is weakly closed and weakly consistent, they produce a series of SAT queries following Biermann and Feldmen's approach ([6], also mentioned below). The result of these SAT queries is a minimum-size automaton consistent with the examples in the table—in their scheme, they do not maintain the invariant that the minimum automaton is equal in size to the upper part of the table as in our approach, so they do not necessarily conjecture hypotheses monotonically.

In follow-on work Leucker and Neider [24] presented a framework which distills the essentials of the algorithm of [16] for learning DFAs. Their formal framework presents the teacher similar to our presentation in Section 3, but they do not highlight the solver as a first-class citizen in the framework and think of it as part of the Learner. They do give an overview of potential learners that work with inexperienced teachers. These encompass an approach without membership queries (a naive enumeration of all DFAs of increasing size) and the approach of [16] with a SAT solver, as well as the approach of [8], which we described above. They do not consider any modification similar to our iMAT with Distinguish, and they do not explore any implementation aspects or benchmarking.

11 Discussion

We have presented algorithms that solve the problem of automata inference from an incomplete teacher. The core ideas we applied to produce our algorithm are data structure agnostic and therefore a first direction for future work is whether we can adapt the work we did in this paper to recent optimizations of $\mathtt{L}^\star$, such as $L^\sharp$.

A particularly interesting optimization is the use of *discrimination trees*, an efficient replacement for observation tables [19, 18]. The first challenge will be to understand how the operations on discrimination trees can be generalized in the setting with $\square$'s.

Another interesting direction would be to look at $\mathtt{L}^\star$ variants developed for other automata models. We expect that the work in this paper applies directly to Mealy and Moore machines, with the caveat that the number of options for filling in blanks will be affected by the size of the output sets. But one could also explore how SMTs solvers could help in learning of weighted and symbolic automata, two models with interesting applications for which $\mathtt{L}^\star$-like algorithms were proposed [5, 4, 12].

Finally, there is another $\mathtt{L}^\star$ adaptation, due to Bollig et al., that learns non-deterministic finite automata with access to an Angluin MAT. Another interesting direction for future work would be to investigate the adaptation of their approach to the incomplete setting.

References

1 Dana Angluin. A note on the number of queries needed to identify regular languages. *Inf. Control.*, 51:76–87, 1981.

2 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

3 Dana Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5(2):121–150, jul 1990. doi:10.1023/A:1022692615781.

4 Borja Balle and Mehryar Mohri. Learning weighted automata. In Andreas Maletti, editor, *Algebraic Informatics*, pages 1–21, Cham, 2015. Springer International Publishing.

5 Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.*, 25(6):1268–1280, dec 1996. doi:10.1137/S009753979326091X.

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

7 Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 487–502, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385988.

8 Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating dfa's for compositional verification. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

9 Orlando Cicchello and Stefan C. Kremer. Beyond edsm. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '02, page 37–48, Berlin, Heidelberg, 2002. Springer-Verlag.

10 Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. doi:10.1017/CBO9781139194655.

11 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

12 Samuel Drews and Loris D'Antoni. Learning symbolic automata. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 173–189, 2017. doi:10.1007/978-3-662-54577-5_10.

13 E. Mark Gold. Language identification in the limit. *Inf. Control.*, 10:447–474, 1967.

14 E. Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, sep 1972. doi:10.1016/0005-1098(72)90033-7.

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

16 O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *3rd International Joint Conference on Automated Reasoning*, volume 4130 of *<A HREF=http://www.springer.de/comp/lncs/index.html>Lecture Notes in Computer Science</A>*, pages 483–497. Springer-Verlag, 2006.

17 Marijn J. H. Heule and Sicco Verwer. Exact dfa identification using sat solvers. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, pages 66–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

18 Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734, pages 307–322. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. URL: `http://link.springer.com/10.1007/978-3-319-11164-3_26`, `doi:10.1007/978-3-319-11164-3_26`.

19 Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA, 1994.

20 Kevin J. Lang. Faster algorithms for finding minimal consistent dfas. Technical report, NEC Research Institute, 1999.

21 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 *Proceedings of the 4th International Colloquium on Grammatical Inference*, ICGI '98, page 1–12, Berlin, Heidelberg, 1998. Springer-Verlag.

22 Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. *SIGPLAN Not.*, 49(6):542–553, jun 2014. `doi:10.1145/2666356.2594333`.

23 Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, page 70–80, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2993236.2993244`.

24 M. Leucker and Daniel Neider. Learning minimal deterministic automata from inexperienced teachers. In *Leveraging Applications of Formal Methods*, 2012.

25 Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. Transregex: Multi-modal regular expression synthesis by generate-and-repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1210–1222. IEEE, 2021.

26 Mark Liffiton and Karem Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40:1–33, 01 2008. `doi:10.1007/s10817-007-9084-z`.

27 Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. In *COLT 1991*, 1991.

28 Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, POPL '17, page 613–625, 2017.

29 Arlindo Oliveira and J.P.M. Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44, 07 2001.

30 Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. *World Scientific*, 01 1992. `doi:10.1142/9789812797902_0004`.

31 J.M. Pena and A.L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(11):1619–1632, 1999. `doi:10.1109/43.806807`.

32 Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, January 1993. `doi:10.1145/138027.138042`.

33 Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, jan 2017. `doi:10.1145/2967606`.

34 Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022. URL: `https://doi.org/10.1007/978-3-030-99524-9_12`, `doi:10.1007/978-3-030-99524-9_12`.

35 Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1990.

36 Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, 2020.

A Another $L^\star_\square$ Example

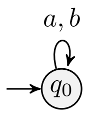
Suppose we start with the following input.

$$L_+ = \{ab, aab, bab, aaab, abab, baab, bbab\}$$
$$L_- = \{aa, ba, bb, aaa, baa, aba, bba, abb, bbb\}$$

We begin with the following table as the only item in the worklist.

	ε
ε	□
a	□
b	□

	ε
ε	⊟
a	⊟
b	⊟



We pop this table off of the worklist, and the SMT solver attempts to fill in its blanks. The solver finds that filling all the blanks with a - satisfies the constraints. We then conjecture the machine that corresponds to the filled-in table.

We get the counterexample $baab$, add its suffixes to E, and push the new (S, E, T), below on the left, onto the top of worklist.

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
b	□	-	+	+	□

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
aa	-	+	+	□	□
ab	+	-	+	□	□
b	□	-	+	+	□

Now the table on the left is unsatisfiable. Hence, we add each element in $S\Sigma - S$ to S and add each resulting (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a\}$ (table depicted on the right above), and the last item has $S = \{\varepsilon, b\}$. Once again, the solver fails because there is no way to fill in the blanks and maintain closedness. Once more, we add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, b\}$, the next item has $S = \{\varepsilon, a, aa\}$, and the last item has $S = \{\varepsilon, a, ab\}$.

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
b	□	-	+	+	□
a	□	+	+	+	□
ba	-	+	+	□	□
bb	-	-	+	□	□

We pop the table above off of the worklist. Again the solver fails because there is no way to fill in the blanks and maintain closedness. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a, aa\}$, the next item has $S = \{\varepsilon, a, ab\}$, the next item has $S = \{\varepsilon, b, ba\}$, and the last item has $S = \{\varepsilon, a, bb\}$.

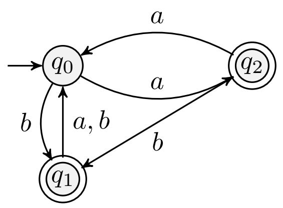
In the next step, we pop the table below on the left off of the worklist. Unsat again. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. At the head of the list we now have the table on the right.

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
aa	-	+	+	□	□
aaa	-	+	□	□	□
aab	+	□	□	□	□
ab	+	-	+	□	□
b	□	-	+	+	□

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
ab	+	-	+	□	□
aa	-	+	+	□	□
aba	-	+	□	□	□
abb	-	□	□	□	□
b	□	-	+	+	□

This time SMT solver successfully fills in the blanks:

	ε	b	ab	aab	$baab$
ε	⊟	⊞	+	+	+
a	⊞	+	+	+	⊞
ab	+	-	+	⊞	⊞
aa	-	+	+	⊞	⊞
aba	-	+	⊞	⊞	⊞
abb	-	⊞	⊞	⊞	⊞
b	⊞	-	+	+	⊞

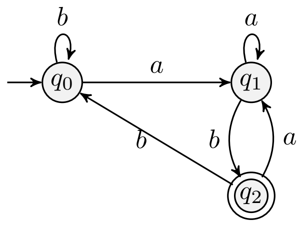


We get counterexample bbb, add its suffixes to E, and the head of the worklist becomes:

	ε	b	ab	aab	$baab$	bb	bbb
ε	□	□	+	+	+	-	-
a	□	+	+	+	□	-	□
ab	+	-	+	□	□	□	□
aa	-	+	+	□	□	□	□
aba	-	+	□	□	□	□	□
abb	-	□	□	□	□	□	□
b	□	-	+	+	□	-	□

Finally the SMT solver successfully fills in the blanks:

	ε	b	ab	aab	$baab$	bb	bbb
ε	⊟	⊟	+	+	+	-	-
a	⊟	+	+	+	⊞	-	⊟
ab	+	-	+	⊞	⊞	⊟	⊟
aa	-	+	+	⊞	⊞	⊟	⊟
aba	-	+	⊞	⊞	⊞	⊟	⊟
abb	-	⊟	⊞	⊞	⊞	⊟	⊟
b	⊟	-	+	+	⊞	-	⊟



The corresponding automaton is consistent with L_+, L_- so we return it as the result.