The Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI-23)

Learning Interpretable Temporal Properties from Positive Examples Only

Rajarshi Roy[1], **Jean-Raphaël Gaglione**[2], **Nasim Baharisangari**[3],
Daniel Neider[4, 5*], **Zhe Xu**[3], **Ufuk Topcu**[2]

[1] Max Planck Institute for Software Systems, Kaiserslautern, Germany
[2] University of Texas at Austin, Texas, USA
[3] Arizona State University, Arizona, USA
[4] TU Dortmund University, Dortmund, Germany
[5] Center for Trustworthy Data Science and Security, University Alliance Ruhr, Germany

Abstract

We consider the problem of explaining the temporal behavior of black-box systems using human-interpretable models. Following recent research trends, we rely on the fundamental yet interpretable models of *deterministic finite automata* (DFAs) and *linear temporal logic* (LTL_f) formulas. In contrast to most existing works for learning DFAs and LTL_f formulas, we consider learning from only positive examples. Our motivation is that negative examples are generally difficult to observe, in particular, from black-box systems. To learn meaningful models from positive examples only, we design algorithms that rely on *conciseness* and *language minimality* of models as regularizers. Our learning algorithms are based on two approaches: a symbolic and a counterexample-guided one. The symbolic approach exploits an efficient encoding of language minimality as a constraint satisfaction problem, whereas the counterexample-guided one relies on generating suitable negative examples to guide the learning. Both approaches provide us with effective algorithms with minimality guarantees on the learned models. To assess the effectiveness of our algorithms, we evaluate them on a few practical case studies.

1 Introduction

The recent surge of complex black-box systems in Artificial Intelligence has increased the demand for designing simple explanations of systems for human understanding. Moreover, in several areas such as robotics, healthcare, and transportation (Bundy et al. 2019; Gunning et al. 2019; Molnar 2022), inferring human-interpretable models has become the primary focus to promote human trust in systems.

To enhance the interpretability of systems, we aim to explain their temporal behavior. For this purpose, models that are typically employed include, among others, finite state machines and temporal logics (Weiss, Goldberg, and Yahav 2018; Roy, Fisman, and Neider 2020). Our focus is on two fundamental models: deterministic finite automata (DFAs) (Rabin and Scott 1959); and formulas in the de facto standard temporal logic: linear temporal logic (LTL) (Pnueli 1977). These models not only possess a host of desirable theoretical properties, but also feature easy-to-grasp syntax and intuitive semantics. The latter properties make them particularly suitable as interpretable models with many applications, e.g., as task knowledge for robotic agents (Kasenberg and Scheutz 2017; Memarian et al. 2020), as a formal specification for verification (Lemieux, Park, and Beschastnikh 2015), as behavior classifier for unseen data (Shvo et al. 2021), and several others (Camacho and McIlraith 2019).

The area of learning DFAs and LTL formulas is well-studied with a plethora of existing works (see related work). Most of them tackle the typical binary classification problem (Gold 1978) of learning concise DFAs or LTL formulas from a finite set of examples partitioned into a positive and a negative set. However, negative examples are hard to obtain in my scenarios. In safety-critical areas, often observing negative examples from systems (e.g., from medical devices and self-driving cars) can be unrealistic (e.g., by injuring patients or hitting pedestrians). Further, often one only has access to a black-box implementation of the system and thus, can extract only its possible (i.e., positive) executions.

In spite of being relevant, the problem of learning concise DFAs and LTL formulas from positive examples, i.e., the corresponding *one class classification* (OCC) problem, has garnered little attention. The primary reason, we believe, is that, like most OCC problems, this problem is an ill-posed one. Specifically, a concise model that classifies all the positive examples correctly is the trivial model that classifies all examples as positive. This corresponds to a single state DFA or, in LTL, the formula $true$. These models, unfortunately, convey no insights about the underlying system.

To ensure a well-defined problem, Avellaneda and Petrenko (2018), who study the OCC problem for DFAs, propose the use of the (accepted) *language* of a model (i.e., the set of allowed executions) as a regularizer. Searching for a model that has minimal language, however, results in one that classifies only the given examples as positive. To avoid this overfitting, they additionally impose an upper bound on the size of the model. Thus, the OCC problem that they state is the following: given a set of positive examples P and a size bound n, learn a DFA that (a) classifies P correctly, (b) has size at most n, and (c) is language minimal. For language comparison, the order chosen is set inclusion.

To solve this OCC problem, Avellaneda and Petrenko (2018) then propose a *counterexample-guided* algorithm.

*Part of this work has been conducted when the author was at the Carl von Ossietzky University of Oldenburg, Germany

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

This algorithm relies on generating suitable negative examples (i.e., counterexamples) iteratively to guide the learning process. Since only the negative examples dictate the algorithm, in many iterations of their algorithm, the learned DFAs do not have a language smaller (in terms of inclusion) than the previous hypothesis DFAs. This results in searching through several unnecessary DFAs.

To alleviate this drawback, our first contribution is a *symbolic* algorithm for solving the OCC problem for DFA. Our algorithm converts the search for a language minimal DFA symbolically to a series of satisfiability problems in Boolean propositional logic, eliminating the need for counterexamples. The key novelty is an efficient encoding of the language inclusion check of DFAs in a propositional formula, which is polynomial in the size of the DFAs. We then exploit an off-the-shelf SAT solver to check satisfiability of the generated propositional formulas and, thereafter, construct a suitable DFA. We expand on this algorithm in Section 3.

We then present two novel algorithms for solving the OCC problem for formulas in LTL_f (LTL over finite traces). While our algorithms extend smoothly to traditional LTL (over infinite traces), our focus here is on LTL_f due to its numerous applications in AI (Giacomo and Vardi 2013). Also, LTL_f being a strict subclass of DFAs, the learning algorithms for DFAs cannot be applied directly to learn LTL_f formulas.

Our first algorithm for LTL_f is a *semi-symbolic* algorithm, which combines ideas from both the symbolic and the counterexample-guided approaches. Roughly, this algorithm exploits negative examples to overcome the theoretical difficulties of symbolically encoding language inclusion for LTL_f. (LTL_f inclusion check is known to be inherently harder than that for DFAs (Sistla and Clarke 1985)). Our second algorithm is simply a counterexample-guided algorithm that relies solely on the generation of negative examples for learning. Section 4 details both algorithms.

To further study the presented algorithms, we empirically evaluate them in several case studies. We demonstrate that our symbolic algorithm solves the OCC problem for DFA in fewer (approximately one-tenth) iterations and runtime comparable to the counterexample-guided algorithm, skipping thousands of counterexample generations. Further, we demonstrate that our semi-symbolic algorithm solves the OCC problem for LTL_f (in average) thrice as fast as the counterexample-guided algorithm. All our experimental results can be found in Section 5. For the supplementary material of this paper, refer to the full version (Roy et al. 2022).

Related Work. The OCC problem described in this paper belongs to the body of works categorized as passive learning (Gold 1978). As alluded to in the introduction, in this topic, the most popular problem is the binary classification problem for learning DFAs and LTL formulas. Notable works include the works by Biermann and Feldman (1972); Grinchtein, Leucker, and Piterman (2006); Heule and Verwer (2010) for DFAs and Neider and Gavran (2018); Camacho and McIlraith (2019); Raha et al. (2022) for LTL/LTL_f.

The OCC problem of learning formal models from positive examples was first studied by Gold (1967). This work showed that the exact identification (in the limit) of certain models (that include DFAs and LTL_f formulas) from positive examples is not possible. Thereby, works have mostly focussed on models that are learnable easily from positive examples, such as pattern languages (Angluin 1980), stochastic finite state machines (Carrasco and Oncina 1999), and hidden Markov Models (Stolcke and Omohundro 1992). None of these works considered learning DFAs or LTL formulas, mainly due to the lack of a meaningful regularizer.

Recently, Avellaneda and Petrenko (2018) proposed the use of language minimality as a regularizer and, thereafter, developed an effective algorithm for learning DFAs. While their algorithm cannot overcome the theoretical difficulties shown by Gold (1967), they still produce a DFA that is a concise description of the positive examples. We significantly improve upon their algorithm by relying on a novel encoding of language minimality using propositional logic.

For temporal logics, there are a few works that consider the OCC problem. Notably, Ehlers, Gavran, and Neider (2020) proposed a learning algorithm for a fragment of LTL which permits a representation known as universally very-weak automata (UVWs). However, since their algorithm relies on UVWs, which has strictly less expressive power than LTL, it cannot be extended to full LTL. Further, there are works on learning LTL (Chou, Ozay, and Berenson 2022) and STL (Jha et al. 2019) formulas from trajectories of high-dimensional systems. These works based their learning on the assumption that the underlying system optimizes some cost functions. Our method, in contrast, is based on the natural notion of language minimality to find tight descriptions, without any assumptions on the system.

A problem similar to our OCC problem is studied in the context of inverse reinforcement learning (IRL) to learn temporal rewards for RL agents from (positive) demonstrations. For instance, Kasenberg and Scheutz (2017) learn concise LTL formulas that can distinguish between the provided demonstrations from random executions of the system. To generate the random executions, they relied on a Markov Decision Process (MDP) implementation of the underlying system. Our regularizers, in contrast, assume the underlying system to be a black-box and need no access to its internal mechanisms. Vazquez-Chanlatte et al. (2018) also learn LTL-like formulas from demonstrations. Their search required a pre-computation of the lattice of formulas induced by the subset order, which can be a bottleneck for scaling to full LTL. Recently, Hasanbeig et al. (2021) devised an algorithm to infer automaton for describing high-level objectives of RL agents. Unlike ours, their algorithm relied on user-defined hyper-parameters to regulate the degree of generalization of the inferred automaton.

2 Preliminaries

In this section, we set up the notation for the rest of the paper.

Let $\mathbb{N} = \{1, 2, \ldots\}$ be the set of natural numbers and $[n] = \{1, 2, \ldots, n\}$ be the set of natural numbers up to n.

Words and Languages. To formally represent system executions, we rely on the notion of words defined over a finite and nonempty *alphabet* Σ. The elements of Σ, which denote relevant system states, are referred to as *symbols*.

A *word* over Σ is a finite sequence $w = a_1 \dots a_n$ where $a_i \in \Sigma$, $i \in [n]$. The *empty word* ε is the empty sequence. The length $|w|$ of w is the number of its symbols (note that $|\varepsilon| = 0$). Moreover, Σ^* denotes the set of all words over Σ. We use $w[i] = a_i$ to denote the i-th symbol of w and $w[i:] = a_i \cdots a_n$ to denote the suffix of w starting from position i.

A *language* L is any set of words from Σ^*. We allow the standard set operations on languages such as inclusion $L_1 \subseteq L_2$, strict inclusion $L_1 \subset L_2$, and difference $L_1 \setminus L_2$. Moreover, we define $\mathit{Pref}(L) \coloneqq \{u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in L\}$.

Propositional logic. All our algorithms rely on propositional logic and, thus, we introduce it briefly. Let Var be a set of propositional variables, which take Boolean values $\{0, 1\}$ (0 represents *false*, 1 represents *true*). Formulas in propositional logic—which we denote by Greek capital letters—are defined recursively as: $\Phi \coloneqq x \in \mathit{Var} \mid \neg\Phi \mid \Phi \vee \Phi$. As syntax sugar, we allow the following standard formulas: true, false, $\Phi \wedge \Psi$, $\Phi \rightarrow \Psi$ and $\Phi \leftrightarrow \Psi$.

An *assignment* $v\colon \mathit{Var} \mapsto \{0, 1\}$ maps propositional variables to Boolean values. Based on an assignment v, we define the semantics of propositional logic using a valuation function $V(v, \Phi)$, which is inductively defined as follows: $V(v, x) = v(x)$, $V(v, \neg\Psi) = 1 - V(v, \Psi)$, and $V(v, \Psi \vee \Phi) = \mathit{max}\{V(v, \Psi), V(v, \Phi)\}$. We say that v satisfies Φ if $V(v, \Phi) = 1$, and we call v a model of Φ. A formula Φ is satisfiable if there exists a model v of Φ.

Arguably, the most well-known problem in propositional logic—the satisfiability (SAT) problem—is the problem of determining whether a propositional formula is satisfiable or not. With the rapid development of SAT solvers (Li and Manyà 2021) (which we exploit in our algorithms), checking the satisfiability of formulas with even millions of variables has become feasible. Most solvers can also return a model when a formula is satisfiable.

3 Learning DFA from Positive Examples

In this section, we present our symbolic algorithm for learning DFAs from positive examples. We begin by formally introducing DFAs.

A *deterministic finite automaton* (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ where Q is a finite set of states, Σ is the alphabet, $q_I \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta\colon Q \times \Sigma \rightarrow Q$ is the transition function. We define the size $|\mathcal{A}|$ of a DFA as its number of states $|Q|$.

Given a word $w = a_1 \dots a_n \in \Sigma^*$, the run of $\mathcal{A}$ on w, denoted by $\mathcal{A}\colon q_1 \xrightarrow{w} q_{n+1}$, is a sequence of states and symbols $q_1 a_1 q_2 a_2 \cdots a_n q_{n+1}$, such that $q_1 = q_I$ and for each $i \in [n]$, $q_{i+1} = \delta(q_i, a_i)$. Moreover, we say w is accepted by $\mathcal{A}$ if the last state in the run $q_{n+1} \in F$. Finally, we define the language of $\mathcal{A}$ as $L(\mathcal{A}) = \{w \in \Sigma^* \mid w \text{ is accepted by } \mathcal{A}\}$.

To introduce the OCC problem for DFAs, we first describe the learning setting. The OCC problem relies on a set of positive examples, which we represent using a finite set of words $P \subset \Sigma^*$. Additionally, the problem requires a bound n to restrict the size of the learned DFA. The role of this size bound is two-fold: (1) it ensures that the learned DFA does not overfit P; and (2) using a suitable bound, one can enforce the learned DFAs to be concise and, thus, interpretable.

Algorithm 1: Symbolic Algorithm for Learning DFA

Input: Positive words P, bound n

1: $\mathcal{A} \leftarrow \mathcal{A}_{\Sigma^*}$, $\Phi^{\mathcal{A}} \coloneqq \Phi_{\texttt{DFA}} \wedge \Phi_P$
2: **while** $\Phi^{\mathcal{A}}$ is satisfiable (with model v) **do**
3: $\quad\mathcal{A} \leftarrow$ DFA constructed from v
4: $\quad\Phi^{\mathcal{A}} \coloneqq \Phi_{\texttt{DFA}} \wedge \Phi_P \wedge \Phi_{\subseteq\mathcal{A}} \wedge \Phi_{\not\supseteq\mathcal{A}}$
5: **end while**
6: **return** $\mathcal{A}$

Finally, we define a DFA $\mathcal{A}$ to be an n-description of P if $P \subseteq L(\mathcal{A})$ and $|\mathcal{A}| \leq n$. When P is clear from the context, we simply say $\mathcal{A}$ is an n-description.

We can now state the OCC problem for DFAs:

Problem 1 (OCC problem for DFAs) *Given a set of positive words P and a size bound n, learn a DFA $\mathcal{A}$ such that: (1) $\mathcal{A}$ is an n-description; and (2) for every DFA $\mathcal{A}'$ that is an n-description, $L(\mathcal{A}') \not\subset L(\mathcal{A})$.*

Intuitively, the above problem asks to search for a DFA that is an n-description and has a minimal language. Note that several such DFAs can exist since the language inclusion is a partial order on the languages of DFA. We, here, are interested in learning only one such DFA, leaving the problem of learning all such DFAs as interesting future work.

3.1 The Symbolic Algorithm

We now present our algorithm for solving Problem 1. Its underlying idea is to reduce the search for an appropriate DFA to a series of satisfiability checks of propositional formulas. Each satisfiable propositional formula enables us to construct a guess, or a so-called *hypothesis* DFA $\mathcal{A}$. In each step, using the hypothesis $\mathcal{A}$, we construct a propositional formula $\Phi^{\mathcal{A}}$ to search for the next hypothesis $\mathcal{A}'$ with a language smaller (in the inclusion order) than the current one. The properties of the propositional formula $\Phi^{\mathcal{A}}$ we construct are: (1) $\Phi^{\mathcal{A}}$ is satisfiable if and only if there exists a DFA $\mathcal{A}'$ that is an n-description and $L(\mathcal{A}') \subset L(\mathcal{A})$; and (2) based on a model v of $\Phi^{\mathcal{A}}$, one can construct a such a DFA $\mathcal{A}'$.

Based on the main ingredient $\Phi^{\mathcal{A}}$, we design our learning algorithm as sketched in Algorithm 1. Our algorithm initializes the hypothesis DFA $\mathcal{A}$ to be $\mathcal{A}_{\Sigma^*}$, which is the one-state DFA that accepts all words in Σ^*. Observe that $\mathcal{A}_{\Sigma^*}$ is trivially an n-description, since $P \subset \Sigma^*$ and $|\mathcal{A}_{\Sigma^*}| = 1$. The algorithm then iteratively exploits $\Phi^{\mathcal{A}}$ to construct the next hypothesis DFAs, until $\Phi^{\mathcal{A}}$ becomes unsatisfiable. Once this happens, we terminate and return the current hypothesis $\mathcal{A}$ as the solution. The correctness of this algorithm follows from the following theorem:

Theorem 1 *Given positive words P and a size bound n, Algorithm 1 learns a DFA $\mathcal{A}$ that is an n-description and for every DFA $\mathcal{A}'$ that is an n-description, $L(\mathcal{A}') \not\subset L(\mathcal{A})$.*

We now expand on the construction of $\Phi^{\mathcal{A}}$. To achieve the aforementioned properties, we define $\Phi^{\mathcal{A}}$ as follows:

$$\Phi^{\mathcal{A}} \coloneqq \Phi_{\texttt{DFA}} \wedge \Phi_P \wedge \Phi_{\subseteq\mathcal{A}} \wedge \Phi_{\not\supseteq\mathcal{A}} \tag{1}$$

The first conjunct $\Phi_{\texttt{DFA}}$ ensures that the propositional variables we will use encode a valid DFA $\mathcal{A}'$. The second conjunct Φ_P ensures that $\mathcal{A}'$ accepts all positive words. The

third conjunct $\Phi_{\subseteq \mathcal{A}}$ ensures that $L(\mathcal{A}')$ is a subset of $L(\mathcal{A})$. The final conjunct $\Phi_{\not\supseteq \mathcal{A}}$ ensures that $L(\mathcal{A}')$ is not a superset of $L(\mathcal{A})$. Together, conjuncts $\Phi_{\subseteq \mathcal{A}}$ and $\Phi_{\not\supseteq \mathcal{A}}$ ensure that $L(\mathcal{A}')$ is a proper subset of $L(\mathcal{A})$. In what follows, we detail the construction of each conjunct.

To encode the hypothesis DFA $\mathcal{A}' = (Q', \Sigma, \delta', q_I', F')$ symbolically, following Heule and Verwer (2010), we rely on the propositional variables: (1) $d_{p,a,q}$ where $p, q \in [n]$ and $a \in \Sigma$; and (2) f_q where $q \in [n]$. The variables $d_{p,a,q}$ and f_q encode the transition function δ' and the final states F', respectively, of $\mathcal{A}'$. Mathematically speaking, if $d_{p,a,q}$ is set to true, then $\delta'(p, a) = q$ and if f_q is set to true, then $q \in F'$. Note that we identify the states Q' using the set $[n]$ and the initial state q_I' using the numeral 1.

Now, to ensure $\mathcal{A}'$ has a deterministic transition function δ', $\Phi_{\texttt{DFA}}$ asserts the following constraint:

$$\bigwedge_{p\in[n]} \bigwedge_{a\in\Sigma} \Big[\bigvee_{q\in[n]} d_{p,a,q} \wedge \bigwedge_{q\neq q'\in[n]} \big[\neg d_{p,a,q} \vee \neg d_{p,a,q'}\big]\Big].$$

Based on a model v of the variables $d_{p,a,q}$ and f_q, we can simply construct $\mathcal{A}'$. We set $\delta'(p, a)$ to be the unique state q for which $v(d_{p,a,q}) = 1$ and $q \in F'$ if $v(f_q) = 1$.

Next, to construct conjunct Φ_P, we introduce variables $x_{u,q}$ where $u \in \mathit{Pref}(P)$ and $q \in [n]$, which track the run of $\mathcal{A}'$ on all words in $\mathit{Pref}(P)$, which is the set of prefixes of all words in P. Precisely, if $x_{u,q}$ is set to true, then there is a run of $\mathcal{A}'$ on u ending in the state q, i.e., $\mathcal{A}'\colon {q'}_I \xrightarrow{u} q$.

Using the introduced variables, Φ_P ensures that the words in P are accepted by imposing the following constraints:

$$x_{\varepsilon,1} \wedge \bigwedge_{q\in\{2,\ldots,n\}} \neg x_{\varepsilon,q}$$

$$\bigwedge_{u\in \mathit{Pref}(P)} \bigwedge_{p,q\in[n]} \bigwedge_{a\in\Sigma} [x_{u,p} \wedge d_{p,a,q}] \rightarrow x_{ua,q}$$

$$\bigwedge_{w\in P} \bigwedge_{q\in[n]} x_{w,q} \rightarrow f_q$$

The first constraint above ensures that the runs start in the initial state q_I' (which we denote using 1) while the second constraint ensures that they adhere to the transition function. The third constraint ensures that the run of $\mathcal{A}'$ on every $w \in P$ ends in a final state and is, hence, accepted.

For the third conjunct $\Phi_{\subseteq \mathcal{A}}$, we must track the synchronized runs of the current hypothesis $\mathcal{A}$ and the next hypothesis $\mathcal{A}'$ to compare their behavior on all words in Σ^*. To this end, we introduce auxiliary variables, $y^{\mathcal{A}}_{q,q'}$ where $q, q' \in [n]$. Precisely, $y^{\mathcal{A}}_{q,q'}$ is set to true, if there exists a word $w \in \Sigma^*$ such that there are runs $\mathcal{A}\colon q_I \xrightarrow{w} q$ and $\mathcal{A}'\colon q_I' \xrightarrow{w} q'$.

To ensure $L(\mathcal{A}') \subseteq L(\mathcal{A})$, $\Phi_{\subseteq \mathcal{A}}$ imposes the following constraints:

$$y^{\mathcal{A}}_{1,1}$$

$$\bigwedge_{q=\delta(p,a)} \bigwedge_{p',q'\in[n]} \bigwedge_{a\in\Sigma} \Big[\big[y^{\mathcal{A}}_{p,p'} \wedge d_{p',a,q'}\big] \rightarrow y^{\mathcal{A}}_{q,q'}\Big]$$

$$\bigwedge_{p\notin F} \bigwedge_{p'\in[n]} \Big[y^{\mathcal{A}}_{p,p'} \rightarrow \neg f_{p'}\Big]$$

The first constraint ensures that the synchronized runs of $\mathcal{A}$ and $\mathcal{A}'$ start in the respective initial states, while the second constraint ensures that they adhere to their respective transition functions. The third constraint ensures that if the synchronized run ends in a non-final state in $\mathcal{A}$, it must also end in a non-final state in $\mathcal{A}'$, hence forcing $L(\mathcal{A}') \subseteq L(\mathcal{A})$.

For constructing the final conjunct $\Phi_{\not\supseteq \mathcal{A}}$, the variables we exploit rely the following result:

Lemma 1 *Let $\mathcal{A}$, $\mathcal{A}'$ be DFAs such that $|\mathcal{A}| = |\mathcal{A}'| = n$ and $L(\mathcal{A}') \subset L(\mathcal{A})$, and let $K = n^2$. Then there exists a word $w \in \Sigma^*$ such that $|w| \leq K$ and $w \in L(\mathcal{A}) \setminus L(\mathcal{A}')$.*

This result provides an upper bound to the length of a word that can distinguish between DFAs $\mathcal{A}$ and $\mathcal{A}'$

Based on this result, we introduce variables $z_{i,q,q'}$ where $i \in [n^2]$ and $q, q' \in [n]$ to track the synchronized run of $\mathcal{A}$ and $\mathcal{A}'$ on a word of length at most $K = n^2$. Precisely, if $z_{i,q,q'}$ is set to true, then there exists a word w of length i, with the runs $\mathcal{A}\colon q_I \xrightarrow{w} q$ and $\mathcal{A}' : q_I' \xrightarrow{w} q'$.

Now, $\Phi_{\not\supseteq \mathcal{A}}$ imposes the following constraints:

$$z_{0,1,1}$$

$$\bigwedge_{i\in[n^2]} \Big[\bigvee_{q,q'\in[n]} z_{i,q,q'} \wedge \big[\bigwedge_{\substack{p\neq q\in[n]\\ p'\neq q'\in[n]}} \neg z_{i,p,p'} \vee \neg z_{i,q,q'}\big]\Big]$$

$$\bigwedge_{\substack{p,q\in[n]\\ p',q'\in[n]}} \Big[\big[z_{i,p,p'} \wedge z_{i+1,q,q'}\big] \rightarrow \bigvee_{\substack{a\in\Sigma \text{ where}\\ q=\delta(p,a)}} d_{p',a,q'}\Big]$$

$$\bigvee_{i\in[n^2]} \bigvee_{\substack{q\in F\\ q'\in[n]}} \Big[z_{i,q,q'} \wedge \neg f_{q'}\Big]$$

The first three constraints above ensure that words up to length n^2 have a valid synchronized run on the two DFAs $\mathcal{A}$ and $\mathcal{A}'$. The final constraint ensures that there is a word of length $\leq n^2$ on which the synchronized run ends in a final state in $\mathcal{A}$ but not in $\mathcal{A}'$, ensuring $L(\mathcal{A}) \not\subseteq L(\mathcal{A}')$.

4 Learning LTL$_\mathrm{f}$ from Positive Examples

We now switch our focus to algorithms for learning LTL$_\mathrm{f}$ formulas from positive examples. We begin with a formal introduction to LTL$_\mathrm{f}$.

Linear temporal logic (over finite traces) (LTL$_\mathrm{f}$) is a logic that reasons about temporal behavior of systems using temporal modalities. While originally LTL$_\mathrm{f}$ is built over propositional variables $\mathcal{P}$, to unify the notation with DFAs, we define LTL$_\mathrm{f}$ over an alphabet Σ. It is, however, not a restriction since an LTL$_\mathrm{f}$ formula over $\mathcal{P}$ can always be translated to an LTL$_\mathrm{f}$ formula over $\Sigma = 2^{\mathcal{P}}$. Formally, we define LTL$_\mathrm{f}$ formulas—denoted by Greek small letters—inductively as:

$$\varphi \coloneqq a \in \Sigma \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

As syntactic sugar, along with additional constants and operators used in propositional logic, we allow the standard temporal operators $\mathbf{F}$ ("finally") and $\mathbf{G}$ ("globally"). We define $\Lambda = \{\neg, \vee, \wedge, \rightarrow, \mathbf{X}, \mathbf{U}, \mathbf{F}, \mathbf{G}\} \cup \Sigma$ to be the set of all operators (which, for simplicity, also includes symbols). We define the size $|\varphi|$ of φ as the number of its unique subformu-

las; e.g., size of $\varphi = (a\,\mathbf{U}\,\mathbf{X}\,b) \vee \mathbf{X}\,b$ is five, since its five distinct subformulas are $a, b, \mathbf{X}\,b, a\,\mathbf{U}\,\mathbf{X}\,b$, and $(a\,\mathbf{U}\,\mathbf{X}\,b) \vee \mathbf{X}\,b$.

To interpret LTL_f formulas over (finite) words, we follow the semantics proposed by Giacomo and Vardi (2013). Given a word w, we define recursively when a LTL_f formula holds at position i, i.e., $w, i \models \varphi$, as follows:

$$w, i \models a \in \Sigma \text{ if and only if } a = w[i]$$
$$w, i \models \neg\varphi \text{ if and only if } w, i \not\models \varphi$$
$$w, i \models \mathbf{X}\,\varphi \text{ if and only if } i < |w| \text{ and } w, i+1 \models \varphi$$
$$w, i \models \varphi\,\mathbf{U}\,\psi \text{ if and only if } w, j \models \psi \text{ for some } i \leq j \leq |w| \text{ and } w, i' \models \varphi \text{ for all } i \leq i' < j$$

We say w *satisfies* φ or, alternatively, φ *holds on* w if $w, 0 \models \varphi$, which, in short, is written as $w \models \varphi$.

The OCC problem for LTL_f formulas, similar to Problem 1, relies upon a set of positive words $P \subset \Sigma^*$ and a size upper bound n. Moreover, an LTL_f formula φ is an n-description of P if, for all $w \in P$, $w \models \varphi$, and $|\varphi| \leq n$. Again, we omit P from n-description when clear. Also, in this section, an n-description refers only to an LTL formula.

We state the OCC problem for LTL_f formulas as follows:

Problem 2 (OCC problem for LTL_f formulas) *Given a set of positive words P and a size bound n, learn an LTL_f formula φ such that: (1) φ is an n-description; and (2) for every LTL_f formula φ' that is an n-description, $\varphi' \not\rightarrow \varphi$ or $\varphi \rightarrow \varphi'$.*

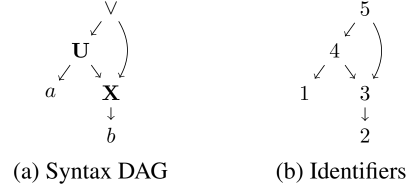
Intuitively, the above problem searches for an LTL_f formula φ that is an n-description and holds on a minimal set of words. Once again, like Problem 1, there can be several such LTL_f formulas, but we are interested in learning exactly one.

4.1 The Semi-Symbolic Algorithm

Our *semi*-symbolic, does not only depend on the current hypothesis, an LTL_f formula φ here, as was the case in Algorithm 1. In addition, it relies on a set of negative examples N, accumulated during the algorithm. Thus, using both the current hypothesis φ and the negative examples N, we construct a propositional formula $\Psi^{\varphi,N}$ to search for the next hypothesis φ'. Concretely, $\Psi^{\varphi,N}$ has the properties that: (1) $\Psi^{\varphi,N}$ is satisfiable if and only if there exists an LTL_f formula φ' that is an n-description, does not hold on any $w \in N$, and $\varphi \not\rightarrow \varphi'$; and (2) based on a model v of $\Psi^{\varphi,N}$, one can construct such an LTL_f formula φ'.

The semi-symbolic algorithm follows a paradigm similar to the one illustrated in Algorithm 1. However, unlike the previous algorithm, the current guess φ', obtained from a model of $\Psi^{\varphi,N}$, may not always satisfy the relation $\varphi' \rightarrow \varphi$. In such a case, we generate a word (i.e., a negative example) that satisfies φ, but not φ' to eliminate φ' from the search space. For the generation of words, we rely on constructing DFAs from the LTL_f formulas (Zhu et al. 2017) and then performing a breadth-first search over them. If, otherwise, $\varphi' \rightarrow \varphi$, we then update our current hypothesis and continue until $\Psi^{\varphi,N}$ is unsatisfiable. Overall, this algorithm learns an appropriate LTL_f formula with the following guarantee:

Theorem 2 *Given positive words P and size bound n, the semi-symbolic algorithm learns an LTL_f formula φ that is an n-description and for every LTL_f formulas φ' that is an n-description, $\varphi' \not\rightarrow \varphi$ or $\varphi \rightarrow \varphi'$.*



(a) Syntax DAG (b) Identifiers

Figure 1: Syntax DAG and identifiers for $(a\,\mathbf{U}\,\mathbf{X}\,b) \vee \mathbf{X}\,b$

We now focus on the construction of $\Psi^{\varphi,N}$, which is significantly different from that of $\Phi^{\mathcal{A}}$. It is defined as follows:

$$\Psi^{\varphi,N} := \Psi_{\text{LTL}} \wedge \Psi_P \wedge \Psi_N \wedge \Psi_{\not\leftarrow\varphi}. \quad (2)$$

The first conjunct Ψ_{LTL} ensures that propositional variables we exploit encode a valid LTL_f formula φ'. The second conjunct Ψ_P ensures that φ' holds on all positive words, while the third, Ψ_N, ensures that it does not hold on the negative words. The final conjunct $\Phi_{\not\leftarrow\varphi}$ ensures that $\varphi \not\rightarrow \varphi'$.

Following Neider and Gavran (2018), all of our conjuncts rely on a canonical syntactic representation of LTL_f formulas as *syntax DAGs*. A syntax DAG is a directed acyclic graph (DAG) that is obtained from the syntax tree of an LTL_f formula by merging its common subformulas. An example of a syntax DAG is illustrated in Figure 1a. Further, to uniquely identify each node of a syntax DAG, we assign them unique identifiers from $[n]$ such that every parent node has an identifier larger than its children (see Figure 1b).

To construct the hypothesis φ', we encode its syntax DAG, using the following propositional variables: (1) $x_{i,\lambda}$ for $i \in [n]$ and $\lambda \in \Lambda$; and (2) $l_{i,j}$ and $r_{i,j}$ for $i \in [n]$ and $j \in [i{-}1]$. The variable $x_{i,\lambda}$ tracks the operator label of the Node i of the syntax DAG of φ', while variables $l_{i,j}$ and $r_{i,j}$ encode the left and right child of Node i, respectively. Mathematically, $x_{i,\lambda}$ is set to true if and only if Node i is labeled with operator λ. Moreover, $l_{i,j}$ (resp. $r_{i,j}$) is set to true if and only if Node i's left (resp. right) child is Node j.

To ensure variables $x_{i,\lambda}$, $l_{i,j}$ and $r_{i,j}$ have the desired meaning, Ψ_{LTL} imposes certain structural constraints. For instance, to ensure that each node of the syntax DAG of φ' is uniquely labeled by an operator, we have the following constraints:

$$\bigwedge_{i \in [n]} \Big[\bigvee_{\lambda \in \Lambda} x_{i,\lambda} \wedge \bigwedge_{\lambda \neq \lambda' \in \Lambda} \big[\neg x_{i,\lambda} \vee \neg x_{i,\lambda'}\big]\Big].$$

Ψ_{LTL} includes additional structural constraints for which we refer the readers to Neider and Gavran (2018).

We now describe the construction of Ψ_P and Ψ_N. Both use variables $y^i_{w,t}$ where $i \in [n]$, $w \in P \cup N$, and $t \in [|w|]$. The variable $y^i_{w,t}$ tracks whether φ'_i holds on the suffix $w[t{:}]$, where φ'_i is the subformula of φ' rooted at Node i. Formally, $y^i_{w,t}$ is set to true if and only if $w[t{:}] \models \varphi'_i$.

To ensure desired meaning of variables $y^i_{w,t}$, Ψ_P and Ψ_N impose semantic constraints, again similar to ones proposed by Neider and Gavran (2018). Exemplarily, the constraint

implementing the semantics of the $\mathbf{X}$-operator is as follows:

$$\bigwedge_{\substack{i\in[n]\\ j\in[i-1]}} x_{i,\mathbf{X}} \wedge l_{i,j} \rightarrow \Big[\big[\bigwedge_{t\in[|w|-1]} y^i_{w,t} \leftrightarrow y^i_{w,t+1}\big] \wedge \neg y^i_{w,|w|}\Big].$$

Intuitively, this constraint states that if Node i is labeled with $\mathbf{X}$ and Node j is the left child of Node i, then φ'_i holds on $w[t{:}]$ if and only if φ'_j holds on $w[t+1{:}]$ (i.e., if $t < |w|$). For the other $\mathrm{LTL_f}$ operators, we impose similar semantic constraints for which we refer the readers to Roy et al. (2022).

To ensure that φ' holds on positive words, $\Psi_P := \bigwedge_{w\in P} y^n_{w,0}$ and to ensure φ' does not hold on negative words, $\Psi_N := \bigwedge_{w\in N} \neg y^n_{w,0}$.

Next, to construct $\Psi_{\not\leftarrow\varphi}$, we symbolically encode a word u that distinguishes formulas φ and φ'. We bound the length of the symbolic word by a *time horizon* $K = 2^{2^{n+1}}$. The choice of K is derived from Lemma 1 and the fact that the size of the equivalent DFA for an $\mathrm{LTL_f}$ formulas can be at most doubly exponential (Giacomo and Vardi 2015).

Our encoding of a symbolic word u relies on variables $p_{t,a}$ where $t \in [K]$ and $a \in \Sigma$. If $p_{t,a}$ is set to true, then $u[t] = a$. To ensure that the variables $p_{t,a}$ encode their desired meaning, we generate a formula $\Psi_{\texttt{word}}$ that consists of the following constraint:

$$\bigwedge_{t\in[K]} \Big[\bigvee_{a\in\Sigma\cup\{\epsilon\}} p_{t,a} \wedge \bigwedge_{a\neq a'\in\Sigma\cup\{\epsilon\}} \big[\neg p_{t,a} \vee \neg p_{t,a'}\big]\Big].$$

This constraint ensures that, in the word u, each position $t \leq K$ has a unique symbol from $\Sigma \cup \{\epsilon\}$.

Further, to track whether φ and φ' hold on u, we have variables $z^{\varphi,i}_{u,t}$ and $z^{\varphi',i}_{u,t}$ where $i \in [n]$, $t \in [K]$. These variables are similar to $y^i_{w,t}$, in the sense that $z^{\varphi,i}_{u,t}$ (resp. $z^{\varphi',i}_{u,t}$) is set to true, if φ (resp. φ') holds at position t. To ensure desired meaning of these variables, we impose semantic constraints $\Psi_{\texttt{sem}}$, similar to the semantic constraints imposed on $y^i_{w,t}$. For instance, for a symbol a, we have the following constraint:

$$\bigwedge_{i\in[n]} \bigwedge_{a\in\Sigma} \Big[x_{i,a} \rightarrow \big[\bigwedge_{t\in[K]} z^{\varphi',i}_{u,t} \leftrightarrow p_{t,a} \big]\Big].$$

Finally, we set $\Psi_{\not\leftarrow\varphi} := \Psi_{\texttt{word}} \wedge \Psi_{\texttt{sem}} \wedge z^{\varphi,n}_{u,0} \wedge \neg z^{\varphi',n}_{u,0}$. Intuitively, the above conjunction ensures that there exists a word on which φ holds and φ' does not.

4.2 The Counterexample-guided Algorithm

We now design a *counterexample-guided* algorithm to solve Problem 2. In contrast to the symbolic (or semi-symbolic) algorithm, this algorithm does not guide the search based on propositional formulas built out of the hypothesis $\mathrm{LTL_f}$ formula. Instead, this algorithm relies entirely on two sets: a set of negative words N and a set of discarded $\mathrm{LTL_f}$ formulas D. Based on these two sets, we design a propositional formula $\Omega^{N,D}$ that has the properties that: (1) $\Omega^{N,D}$ is satisfiable if and only if there exists an $\mathrm{LTL_f}$ formula φ that is an n-description, does not hold on $w \in N$, and is not one of the formulas in D; and (2) based on a model v of $\Omega^{N,D}$, one can construct such an $\mathrm{LTL_f}$ formula φ'.

Being a counterexample-guided algorithm, the construction of the sets N and D forms the crux of the algorithm. In each iteration, these sets are updated based on the relation between the hypothesis φ and the current guess φ' (obtained from a model of $\Omega^{N,D}$). There are exactly three relevant cases, which we discuss briefly.

- First, $\varphi' \leftrightarrow \varphi$, i.e., φ' and φ hold on the exact same set of words. In this case, the algorithm discards φ', due to its equivalence to φ, by adding it to D.
- Second, $\varphi' \rightarrow \varphi$ and $\varphi \not\rightarrow \varphi'$, i.e., φ' holds on a proper subset of the set of words on which φ hold. In this case, our algorithm generates a word that satisfies φ and not φ', which it adds to N to eliminate φ.
- Third, $\varphi' \not\leftarrow \varphi$, i.e., φ' does not hold on a subset of the set of words on which φ hold. In this case, our algorithm generates a word w that satisfies φ' and not φ, which it adds to N to eliminate φ'.

By handling the cases mentioned above, we obtain an algorithm with guarantees (formalized in Theorem 2) exactly the same as the semi-symbolic algorithm in Section 4.1.

5 Experiments

In this section, we evaluate the performance of the proposed algorithms using three case studies. First, we evaluate the performance of Algorithm 1, referred to as $\texttt{SYM}_{\texttt{DFA}}$, and compare it to a baseline counterexample-guided algorithm by Avellaneda and Petrenko (2018), referred to as $\texttt{CEG}_{\texttt{DFA}}$ in our first case study. Then, we evaluate the performance of the proposed semi-symbolic algorithm (Section 4.1), referred to as $\texttt{S-SYM}_{\texttt{LTL}}$, and the counterexample-guided algorithm (Section 4.2), referred to as $\texttt{CEG}_{\texttt{LTL}}$, for learning $\mathrm{LTL_f}$ formulas in our second and third case studies.

In $\texttt{S-SYM}_{\texttt{LTL}}$, we fixed the time horizon K to a natural number, instead of the double exponential theoretical upper bound of $2^{2^{n+1}}$. Using this heuristic means that $\texttt{S-SYM}_{\texttt{LTL}}$ does not solve Problem 2, but we demonstrate that we produced good enough formulas in practice.

In addition, we implemented two existing heuristics from Avellaneda and Petrenko (2018) to all the algorithms. First, in every algorithm, we learned models in an incremental manner, i.e., we started by learning DFAs (resp. $\mathrm{LTL_f}$ formulas) of size 1 and then increased the size by 1. We repeated the process until bound n. Second, we used a set of positive words P' instead of P that starts as an empty set, and at each iteration of the algorithm, if the inferred language does not contain some words from P, we then extended P' with one of such words, preferably the shortest one. This last heuristic helped when dealing with large input samples because it used as few words as possible from P.

We implemented every algorithm in Python 3[1], using PySAT (Ignatiev, Morgado, and Marques-Silva 2018) for learning DFA, and an ASP (Baral 2003) encoding that we solve using clingo (Gebser et al. 2017) for learning $\mathrm{LTL_f}$ formulas. Overall, we ran all the experiments using 8 GiB of RAM and two CPU cores with a clock speed of 3.6 GHz.

[1]https://github.com/cryhot/samp2symb/tree/paper/posdata

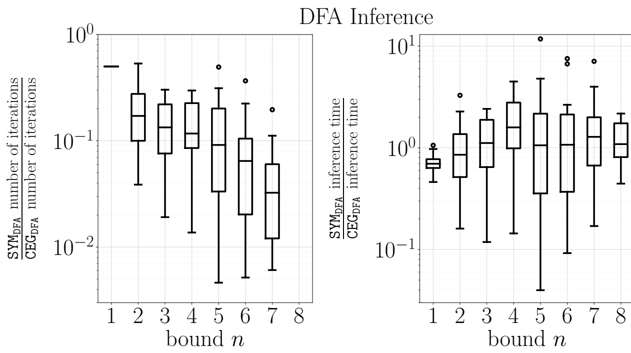


Figure 2: Comparison of $\mathtt{SYM_{DFA}}$ and $\mathtt{CEG_{DFA}}$ in terms of the runtime and the number of iterations of the main loop.

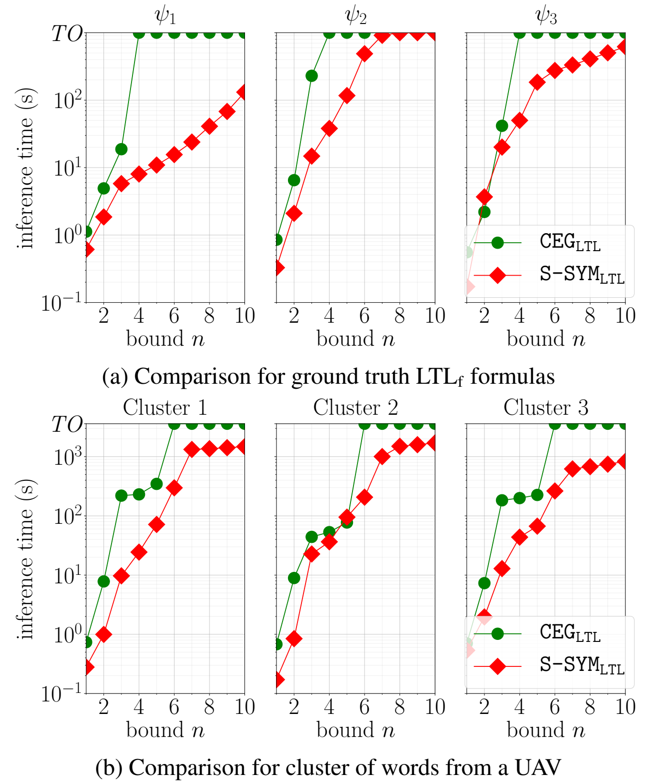
Learning DFAs For this case study, we considered a set of 28 random DFAs of size 2 to 10 generated using AALpy (Muškardin et al. 2022). Using each random DFA, we generated a set of 1000 positive words of lengths 1 to 10. We ran algorithms $\mathtt{CEG_{DFA}}$ and $\mathtt{SYM_{DFA}}$ with a timeout $TO = 1000$s, and for n up to 10.

Figure 2 shows a comparison between the performance of $\mathtt{SYM_{DFA}}$ and $\mathtt{CEG_{DFA}}$ in terms of the inference time and the required number of iterations of the main loop. On the left plot, the average ratio of the number of iterations is 0.14, which, in fact, shows that $\mathtt{SYM_{DFA}}$ required noticeably less number of iterations compared to $\mathtt{CEG_{DFA}}$. On the right plot, the average ratio of the inference time is 1.09, which shows that the inference of the two algorithms is comparable, and yet $\mathtt{SYM_{DFA}}$ is computationally less expensive since it requires fewer iterations.

Learning Common LTL$_\mathrm{f}$ Patterns In this case study, we generated sample words using 12 common LTL patterns (Dwyer, Avrunin, and Corbett 1999). For instance, $\psi_1 \coloneqq \mathbf{G}\, a_0$, $\psi_2 \coloneqq \mathbf{G}(a_1 \rightarrow \mathbf{G}(\neg a_0))$ and $\psi_3 \coloneqq \mathbf{G}(\neg a_0) \vee \mathbf{F}(a_0 \wedge (\mathbf{F}(a_1))$. We refer to the full paper for a complete list of LTL$_\mathrm{f}$ patterns. Using each of these 12 ground truth LTL$_\mathrm{f}$ formulas, we generated a sample of 10000 positive words of length 10. Then, we inferred LTL$_\mathrm{f}$ formulas for each sample using $\mathtt{CEG_{LTL}}$ and $\mathtt{S\text{-}SYM_{LTL}}$, separately. For both algorithms, we set the maximum formula size $n = 10$ and a timeout of $TO = 1000$s. For $\mathtt{S\text{-}SYM_{LTL}}$, we additionally set the time horizon $K = 8$.

Figure 3a represents a comparison between the mentioned algorithms in terms of inference time for the ground truth LTL$_\mathrm{f}$ formulas ψ_1, ψ_2, and ψ_3. On average, $\mathtt{S\text{-}SYM_{LTL}}$ ran 173.9% faster than $\mathtt{CEG_{LTL}}$ for all the 12 samples. Our results showed that the LTL$_\mathrm{f}$ formulas φ inferred by $\mathtt{S\text{-}SYM_{LTL}}$ were more or equally specific than the ground truth LTL$_\mathrm{f}$ formulas ψ (i.e., $\varphi \rightarrow \psi$) for five out of the 12 samples, while the LTL$_\mathrm{f}$ formulas φ' inferred by $\mathtt{CEG_{LTL}}$ were equally or more specific than the ground truth LTL$_\mathrm{f}$ formulas ψ (i.e., $\varphi' \rightarrow \psi$) for three out of the 12 samples.

Learning LTL from Trajectories of Unmanned Aerial Vehicle (UAV) In this case study, we implemented $\mathtt{S\text{-}SYM_{LTL}}$ and $\mathtt{CEG_{LTL}}$ using sample words of a simulated



(a) Comparison for ground truth LTL$_\mathrm{f}$ formulas

(b) Comparison for cluster of words from a UAV

Figure 3: Comparison of $\mathtt{S\text{-}SYM_{LTL}}$ and $\mathtt{CEG_{LTL}}$ in terms of the runtime on the two LTL$_\mathrm{f}$ case studies.

unmanned aerial vehicle (UAV) for learning LTL$_\mathrm{f}$ formulas. Here, we used 10000 words clustered into three bundles using the k-means clustering approach. Each word summarizes selective binary features such as x_0: "low battery", x_1: "glide (not thrust)", x_2: "change yaw angle", x_3: "change roll angle", etc. We set $n = 10$, $K = 8$, and a timeout of $TO = 3600$s. We inferred LTL$_\mathrm{f}$ formulas for each cluster using $\mathtt{CEG_{LTL}}$ and $\mathtt{S\text{-}SYM_{LTL}}$.

Figure 3b depicts a comparison between $\mathtt{CEG_{LTL}}$ and $\mathtt{S\text{-}SYM_{LTL}}$ in terms of the inference time for three clusters. Our results showed that, on average, $\mathtt{S\text{-}SYM_{LTL}}$ is 260.73% faster than $\mathtt{CEG_{LTL}}$. Two examples of the inferred LTL$_\mathrm{f}$ formulas from the UAV words were $(\mathbf{F}\, x_1) \rightarrow (\mathbf{G}\, x_1)$ which reads as "*either the UAV always glides, or it never glides*" and $\mathbf{G}(x_2 \rightarrow x_3)$ which reads as "*a change in yaw angle is always accompanied by a change in roll angle*".

6 Conclusion

We presented novel algorithms for learning DFAs and LTL$_\mathrm{f}$ formulas from positive examples only. Our algorithms rely on conciseness and language minimality as regularizers to learn meaningful models. We demonstrated the efficacy of our algorithms in three case studies.

A natural direction of future work is to lift our techniques to tackle learning from positive examples for other finite state machines (e.g., non-deterministic finite automata) and more expressive temporal logics (e.g., linear dynamic logic (LDL) (Giacomo and Vardi 2013)).

Acknowledgments

We are especially grateful to Dhananjay Raju for introducing us to Answer Set Programming and guiding us in using it to solve our SAT problem. This work has been financially supported by the Defense Advanced Research Projects Agency (DARPA) (Contract number HR001120C0032), Army Research Laboratory (ARL) (Contract number W911NF2020132 and ACC-APG-RTP W911NF), National Science Foundation (NSF) (Contract number 1646522), and Deutsche Forschungsgemeinschaft (DFG) (Grant number 434592664) and has been partly supported by the Research Center Trustworthy Data Science and Security (https://rc-trust.ai), one of the Research Alliance centers within the UA Ruhr (https://uaruhr.de).

References

Angluin, D. 1980. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1): 46–62.

Avellaneda, F.; and Petrenko, A. 2018. Inferring DFA without Negative Examples. In Unold, O.; Dyrka, W.; and Wieczorek, W., eds., *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, 17–29. PMLR.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Biermann, A. W.; and Feldman, J. A. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Computers*, 21(6): 592–597.

Bundy, A.; Crowcroft, J.; Ghahramani, Z.; Reid, N.; Weller, A.; McCarthy, N.; and Montgomery, J. 2019. Explainable AI: the basics. *The Royal Society. Available at: Explainable AI: the basics (royalsociety. org)(Accessed on April 21, 2021)*.

Camacho, A.; and McIlraith, S. A. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. In *ICAPS*, 621–630. AAAI Press.

Carrasco, R. C.; and Oncina, J. 1999. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO Theor. Informatics Appl.*, 33(1): 1–20.

Chou, G.; Ozay, N.; and Berenson, D. 2022. Learning temporal logic formulas from suboptimal demonstrations: theory and experiments. *Auton. Robots*, 46(1): 149–174.

Dwyer, M. B.; Avrunin, G. S.; and Corbett, J. C. 1999. Patterns in Property Specifications for Finite-State Verification. In Boehm, B. W.; Garlan, D.; and Kramer, J., eds., *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, 411–420. ACM.

Ehlers, R.; Gavran, I.; and Neider, D. 2020. Learning Properties in LTL $\cap$ ACTL from Positive Examples Only. In *FMCAD*, 104–112. IEEE.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2017. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811.

Giacomo, G. D.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*, 854–860. IJCAI/AAAI.

Giacomo, G. D.; and Vardi, M. Y. 2015. Synthesis for LTL and LDL on Finite Traces. In *IJCAI*, 1558–1564. AAAI Press.

Gold, E. M. 1967. Language Identification in the Limit. *Inf. Control.*, 10(5): 447–474.

Gold, E. M. 1978. Complexity of Automaton Identification from Given Data. *Inf. Control.*, 37(3): 302–320.

Grinchtein, O.; Leucker, M.; and Piterman, N. 2006. Inferring Network Invariants Automatically. In *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, 483–497. Springer.

Gunning, D.; Stefik, M.; Choi, J.; Miller, T.; Stumpf, S.; and Yang, G.-Z. 2019. XAI2014;Explainable artificial intelligence. *Science Robotics*, 4(37): eaay7120.

Hasanbeig, M.; Jeppu, N. Y.; Abate, A.; Melham, T.; and Kroening, D. 2021. DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning. In *AAAI*, 7647–7656. AAAI Press.

Heule, M.; and Verwer, S. 2010. Exact DFA Identification Using SAT Solvers. In *10th International Colloquium of Grammatical Inference: Theoretical Results and Applications, ICGI '10*, volume 6339 of *LNCS*, 66–79. Springer.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*, 428–437.

Jha, S.; Tiwari, A.; Seshia, S. A.; Sahai, T.; and Shankar, N. 2019. TeLEx: learning signal temporal logic from positive examples using tightness metric. *Formal Methods Syst. Des.*, 54(3): 364–387.

Kasenberg, D.; and Scheutz, M. 2017. Interpretable apprenticeship learning with temporal logic specifications. In *CDC*, 4914–4921. IEEE.

Lemieux, C.; Park, D.; and Beschastnikh, I. 2015. General LTL Specification Mining (T). In *ASE*, 81–92. IEEE Computer Society.

Li, C.; and Manyà, F., eds. 2021. *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*. Springer.

Memarian, F.; Xu, Z.; Wu, B.; Wen, M.; and Topcu, U. 2020. Active Task-Inference-Guided Deep Inverse Reinforcement Learning. In *CDC*, 1932–1938. IEEE.

Molnar, C. 2022. *Model-agnostic interpretable machine learning*. Ph.D. thesis, Ludwig Maximilian University of Munich, Germany.

Muškardin, E.; Aichernig, B.; Pill, I.; Pferscher, A.; and Tappler, M. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 1–10.

Neider, D.; and Gavran, I. 2018. Learning Linear Temporal Properties. In Bjørner, N.; and Gurfinkel, A., eds., *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, 1–10. IEEE.

Pnueli, A. 1977. The Temporal Logic of Programs. In *18th Annual Symposium of Foundations of Computer Science, FOCS '77*, 46–57. IEEE Computer Society.

Rabin, M.; and Scott, D. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3: 114–125.

Raha, R.; Roy, R.; Fijalkow, N.; and Neider, D. 2022. Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, 263–280. Springer.

Roy, R.; Fisman, D.; and Neider, D. 2020. Learning Interpretable Models in the Property Specification Language. In *IJCAI*, 2213–2219. ijcai.org.

Roy, R.; Gaglione, J.; Baharisangari, N.; Neider, D.; Xu, Z.; and Topcu, U. 2022. Learning Interpretable Temporal Properties from Positive Examples Only. *CoRR*, abs/2209.02650.

Shvo, M.; Li, A. C.; Icarte, R. T.; and McIlraith, S. A. 2021. Interpretable Sequence Classification via Discrete Optimization. In *AAAI*, 9647–9656. AAAI Press.

Sistla, A. P.; and Clarke, E. M. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3): 733–749.

Stolcke, A.; and Omohundro, S. 1992. Hidden Markov Model Induction by Bayesian Model Merging. In Hanson, S.; Cowan, J.; and Giles, C., eds., *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann.

Vazquez-Chanlatte, M.; Jha, S.; Tiwari, A.; Ho, M. K.; and Seshia, S. A. 2018. Learning Task Specifications from Demonstrations. In *NeurIPS*, 5372–5382.

Weiss, G.; Goldberg, Y.; and Yahav, E. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, 5244–5253. PMLR.

Zhu, S.; Tabajara, L. M.; Li, J.; Pu, G.; and Vardi, M. Y. 2017. Symbolic LTLf Synthesis. In *IJCAI*, 1362–1369. ijcai.org.