The complexity of learning linear temporal formulas from examples

Nathanaël Fijalkow NFIJALKOW@TURING.AC.UK
CNRS, LaBRI, Université de Bordeaux, France, and The Alan Turing Institute, London, United Kingdom

Guillaume Lagarde GUILLAUME.LAGARDE@LABRI.FR
CNRS, LaBRI, Université de Bordeaux

Editors: Jane Chandlee, Rémi Eyraud, Jeffrey Heinz, Adam Jardine, and Menno van Zaanen

Abstract

In this paper we initiate the study of the computational complexity of learning linear temporal logic (LTL) formulas from examples. We construct approximation algorithms for fragments of LTL and prove hardness results; in particular we obtain tight bounds for the fragment containing only the next operator and conjunctions, and prove NP-hardness results for many fragments.

Keywords: passive learning, automata learning, linear temporal logic, approximation algorithms

1. Introduction

We are interested in the complexity of learning formulas of Linear Temporal Logic (**LTL**) from examples, in a passive scenario: from a set of positive and negative words, the objective is to construct a formula, as small as possible, which satisfies the positive words and does not satisfy the negative words.

Passive learning of languages has a long history paved with negative results. Learning automata is notoriously difficult from a theoretical perspective, as witnessed by the original NP-hardness result of learning a Deterministic Finite Automaton (DFA) from examples by Gold (1978). This line of hardness results culminates with the inapproximability result of Pitt and Warmuth (1993) stating that there is no polynomial time algorithm (unless $\mathsf{P} = \mathsf{NP}$) for learning a DFA from examples even up to a polynomial approximation of their size.

One approach to cope with such hardness results is to change representation, replacing automata by logical formulas: their syntactic structures make them more amenable to principled search algorithms. There is a range of potential logical formalisms to choose from depending on the application domain. Linear Temporal Logic (Pnueli, 1977) is a prominent logic for specifying temporal properties over words. It has become a de facto standard in many fields such as model checking, program analysis, and motion planning for robotics. A key property making **LTL** a strong candidate as a concept class is that its syntax does not include variables, contributing to the fact that **LTL** formulas are typically easy to interpret and therefore useful as explanations.

Over the past five to ten years learning temporal logics (of which **LTL** is the core) has become an active research area, with applications in program specification (Lemieux et al., 2015) and anomaly and fault detections (Bombara et al., 2016). A number of different approaches have been proposed, leveraging SAT solvers (Neider and Gavran, 2018), automata (Camacho and McIlraith, 2019), and Bayesian inference (Kim et al., 2019), and extended to more expressive logics such as Property Specification Language (PSL) (Roy et al., 2020) and Computational Tree Logic (CTL) (Ehlers et al., 2020).

Nothing is known about the computational complexity of the underlying problem; indeed the works cited above focused on constructing efficient algorithms for practical applications. The goal of this paper is to initiate the study of the complexity of learning **LTL** formulas from examples.

Our contributions. We present a set of results for three fragments of **LTL**. For all three fragments we show that the learning problem is NP-complete. As usual for negative complexity statements of this form, they are conditional in that they assume that $\mathsf{P} \neq \mathsf{NP}$. The parameter n below is the number of positive (or negative) words in the input.

- In Section 3 we study $\mathbf{LTL}(\mathbf{X}, \wedge)$, which is the fragment containing only the *next* operator and conjunctions. We obtain matching upper and lower bounds on approximation algorithms: we show that there exists a polynomial time $\log(n)$-approximation algorithm for learning $\mathbf{LTL}(\mathbf{X}, \wedge)$, and that the approximation ratio cannot be improved for polynomial time algorithms.

- In Section 4 we study $\mathbf{LTL}(\mathbf{F}, \wedge)$, which is the fragment containing only the *eventually* operator and conjunctions. We construct an n-approximation algorithm and show that there is no polynomial time $(1 - o(1)) \cdot \log(n)$-approximation algorithm.

- In Section 5 we study $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$, which is the fragment containing the *eventually* and *next* operators, conjunctions and disjunctions.

We conclude in Section 6, listing remaining open problems.

2. Preliminaries

Unless otherwise specified we use the alphabet $\Sigma = \{a, b\}$ of size 2. We index words from position 1 (not 0) and the letter at position i in the word w is $w(i)$, so $w = w(1) \dots w(\ell)$. The empty word is ε.

The syntax[1] of Linear Temporal Logic (**LTL**) includes atomic formulas $c \in \Sigma$, the boolean operators $\wedge$ and $\vee$, and the temporal operators **X** and **F**. The semantics of **LTL** over finite words is defined inductively over formulas, through the notation $w, i \models \phi$ where $w \in \Sigma^*$ is a word of length ℓ, $i \in [1, \ell]$ is a position in w, and ϕ an **LTL** formula. The definition is given below for the atomic formulas and temporal operators **X** and **F**, with boolean operators interpreted as usual.

1. **LTL** also includes a Globally operator **G** which is dual to **F**, and an Until operator **U** extending both **F** and **G**. In this paper we only consider fragments of $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$.

- $w, i \models c$ if $w(i) = c$.
- Next: $w, i \models \mathbf{X}\phi$ if $i < \ell$ and $w, i+1 \models \phi$.
- Eventually: $w, i \models \mathbf{F}\phi$ if $w, i' \models \phi$ for some $i' \in [i, \ell]$.

We then write $w \models \phi$ if $w, 1 \models \phi$ and say that w satisfies ϕ. We consider fragments of **LTL** by specifying which boolean connectives and temporal operators are allowed. For instance $\mathbf{LTL}(\mathbf{X}, \wedge)$ is the set of all **LTL** formulas using only atomic formulas, conjunctions, and the *next* operator. The full logic we consider here is $\mathbf{LTL} = \mathbf{LTL}(\mathbf{F}, \mathbf{G}, \mathbf{X}, \wedge, \vee)$. The size of a formula is the size of its syntactic tree, so for instance the size of $\mathbf{F}\phi$ is the size of ϕ plus one, and the size of $\phi_1 \wedge \phi_2$ is the sum of the sizes of ϕ_1 and ϕ_2 plus one. We say that two formulas are equivalent if they have the same semantics.

The LTL learning problem. The **LTL** learning decision problem is:

INPUT:	$u_1, \dots, u_n, v_1, \dots, v_m \in \Sigma^*$ and $k \in \mathbb{N}$,
QUESTION:	does there exist an **LTL** formula ϕ of size at most k such that for all $j \in [1, n]$, we have $u_j \models \phi$, and for all $j \in [1, m]$, we have $v_j \not\models \phi$?

In that case we say that ϕ separates $u_1, \dots, u_n$ from $v_1, \dots, v_m$, or simply that ϕ is a separating formula if the words are clear from the context. We call $u_1, \dots, u_n$ the positive words, and $v_1, \dots, v_m$ the negative words. The **LTL** learning problem is analogously defined for any fragment of **LTL**.

Parameters for complexity analysis. Without loss of generality we can assume that $n = m$ (adding duplicate identical words to have an equal number of positive and negative words). We also assume for technical convenience that all words have the same length ℓ. Therefore the three important parameters for the complexity of the **LTL** learning problem are: the number of words n, the length of the words ℓ, and the required formula size k.

Representation. The words given as input are represented in the natural way by listing the letters. We emphasise a subtlety on the representation of k: it can be given in binary (a standard assumption) or in unary.

In the first case, the input size is $O(n \cdot \ell + \log(k))$, so the formula ϕ we are looking for may be exponential in the input size. Therefore it is not clear a priori that the **LTL** learning problem is in NP. Opting for a unary encoding, the input size becomes $O(n \cdot \ell + k)$, and in that case an easy argument shows that the **LTL** learning problem is in NP.

We follow the standard representation: k is given in binary, and therefore it is not immediate that the **LTL** learning problem is in NP.

Convention. Typically $i \in [1, \ell]$ is a position in a word and $j \in [1, n]$ is used for indexing words.

A naive algorithm. Let us start our complexity analysis of the learning **LTL** problem by constructing a naive algorithm for the whole logic.

Theorem 1 *There exists an algorithm for learning **LTL** in time and space $O(exp(k) \cdot n \cdot \ell)$, where $exp(k)$ is exponential in k.*

Notice that the dependence of the algorithm presented in Theorem 1 is linear in n and ℓ, and it is exponential only in k, but since k is represented in binary this is potentially a doubly-exponential algorithm.

Proof For a formula $\phi \in \mathbf{LTL}$, we write $\langle \phi \rangle : \{u_1, \dots, u_n, v_1, \dots, v_n\} \to \{0,1\}^\ell$ for the function defined by

$$\langle \phi \rangle(w)(i) = \begin{cases} 1 & \text{if } w, i \models \phi, \\ 0 & \text{if } w, i \not\models \phi, \end{cases}$$

for $w \in \{u_1, \dots, u_n, v_1, \dots, v_n\}$.

Note that ϕ is separating if and only if $\langle \phi \rangle(u_j)(1) = 1$ and $\langle \phi \rangle(v_j)(1) = 0$ for all $j \in [1, n]$. The algorithm simply consists in enumerating all formulas ϕ of **LTL** of size at most k inductively, constructing $\langle \phi \rangle$, and checking whether ϕ is separating. Initially, we construct $\langle a \rangle$ and $\langle b \rangle$, and then once we have computed $\langle \phi \rangle$ and $\langle \psi \rangle$, we can compute $\langle \phi \wedge \psi \rangle$, $\langle \phi \vee \psi \rangle$, $\langle \mathbf{X}\phi \rangle$ and $\langle \mathbf{F}\phi \rangle$ in time $O(n \cdot \ell)$. To conclude, we note that the number of formulas[2] of **LTL** of size at most k is exponential in k. ■

Approximation algorithms. The goal of this paper is to understand the complexity of learning fragments of **LTL** and to construct efficient approximation algorithms. An α-approximation algorithm for learning **LTL** (or some fragment of **LTL**) does the following: the algorithm either determines that there are no separating formulas, or constructs a separating formula ϕ which has size at most $\alpha \cdot m$ with m the size of a minimal separating formula.

3. LTL(X, ∧)

Normalisation

We first state and prove a normalisation lemma for formulas in $\mathbf{LTL}(\mathbf{X}, \wedge)$.

We define the class of "patterns" as formulas generated by the following grammar:

$$P \doteq \mathbf{X}^i c \ \mid \ \mathbf{X}^i (c \wedge P) \text{ with } i \geq 0 \text{ and } c \in \Sigma.$$

Unravelling the definition we get the following general form for patterns:

$$P = \mathbf{X}^{i_1 - 1}(c_1 \wedge \mathbf{X}^{i_2 - i_1}(\cdots \wedge \mathbf{X}^{i_p - i_{p-1}} c_p) \cdots),$$

with $1 \leq i_1 < i_2 < \cdots < i_p$ and $c_1, \dots, c_p \in \Sigma$. It is equivalent to the (larger in size) formula $\bigwedge_{q \in [1,p]} \mathbf{X}^{i_q - 1} c_q$, which states that for each $q \in [1, p]$, the letter in position i_q is c_q.

2. The asymptotics can be obtained using classical techniques from Analytic Combinatorics, see Flajolet and Sedgewick (2008): the number of **LTL** formulas of size k is asymptotically equivalent to $\frac{\sqrt{14} \cdot 7^k}{2\sqrt{\pi k^3}}$.

To determine the size of a pattern P we look at two parameters: its last position $\mathbf{last}(P) = i_p$ and its width $\mathbf{width}(P) = p$. The size of P is $\mathbf{last}(P) + 2(\mathbf{width}(P) - 1)$. The two parameters of a pattern, last position and width, hint at the key trade-off we will have to face in learning $\mathbf{LTL}(\mathbf{X}, \wedge)$ formulas: do we increase the last position, to reach further letters in the words, or the width, to further restrict the set of satisfying words?

Lemma 2 *For every formula $\phi \in \boldsymbol{LTL}(\boldsymbol{X}, \wedge)$ there exists an equivalent pattern of size smaller than or equal to ϕ.*

Proof We proceed by induction on ϕ.

- Atomic formulas are already a special case of patterns.
- If $\phi = \mathbf{X}\phi'$, then by induction we get a pattern P equivalent to ϕ', and $\mathbf{X}P$ is a pattern and equivalent to ϕ.
- If $\phi = \phi_1 \wedge \phi_2$, then by induction we get two patterns P_1 and P_2 equivalent to ϕ_1 and ϕ_2. We use the inductive definition for patterns to show that $P_1 \wedge P_2$ is equivalent to another pattern. We focus on the case $P_1 = \mathbf{X}^{i_1}(c_1 \wedge P_1')$ and $P_2 = \mathbf{X}^{i_2}(c_2 \wedge P_2')$, the other cases are simpler instances of this one.

  There are two cases: $i_1 = i_2$ or $i_1 \neq i_2$.

  If $i_1 = i_2$, either $c_1 \neq c_2$ and then $P_1 \wedge P_2$ is equivalent to false, which is the pattern $c_1 \wedge c_2$, or $c_1 = c_2$, and then $P_1 \wedge P_2$ is equivalent to $\mathbf{X}^{i_1}(c_1 \wedge P_1' \wedge P_2')$. Since P_1' is smaller than ϕ_1 and P_2' is smaller than ϕ_2, $P_1' \wedge P_2'$ is smaller than ϕ, so we can apply the induction hypothesis; $P_1' \wedge P_2'$ is equivalent to a pattern P'. Thus the pattern $\mathbf{X}^{i_1}(c_1 \wedge P')$ is equivalent to $P_1 \wedge P_2$, hence to ϕ.

  If $i_1 \neq i_2$, without loss of generality $i_1 < i_2$, then $P_1 \wedge P_2$ is equivalent to $\mathbf{X}^{i_1}(c_1 \wedge P_1' \wedge \mathbf{X}^{i_2 - i_1}(c_2 \wedge P_2'))$. Since P_1' is smaller than ϕ_1 and $\mathbf{X}^{i_2-i_1}(c_2 \wedge P_2')$ is smaller than ϕ_2, $P_1' \wedge \mathbf{X}^{i_2-i_1}(c_2 \wedge P_2')$ is smaller than ϕ, so we can apply the induction hypothesis; $P_1' \wedge \mathbf{X}^{i_2-i_1}(c_2 \wedge P_2')$ is equivalent to a pattern P'. Thus the pattern $\mathbf{X}^{i_1}(c_1 \wedge P')$ is equivalent to $P_1 \wedge P_2$, hence to ϕ.

■

The first simple corollary of Lemma 2 is a non-deterministic polynomial time algorithm.

Theorem 3 *The learning problem for $\boldsymbol{LTL}(\boldsymbol{X}, \wedge)$ is in* NP.

An approximation algorithm

Let us start by giving a polynomial time approximation algorithm: it constructs a separating formula which is at most $\log(n)$ times larger than the smallest separating formula.

Theorem 4 *There exists a $O(n \cdot \ell^2)$ time $\log(n)$-approximation algorithm for learning $\boldsymbol{LTL}(\boldsymbol{X}, \wedge)$.*

Algorithm 1: The greedy algorithm returning a $\log(n)$-approximation of a minimal separating $\mathbf{LTL}(\mathbf{X}, \wedge)$-formula with last position equal to the length of the words.

```
Data: Words $u_1, \dots, u_n, v_1, \dots, v_n$ of length $\ell$.
$X \leftarrow \{i \in [1,\ell] : \exists c \in \Sigma, \forall j \in [1,n], u_j(i) = c\}$
for $i \in X$ do
    $Y_i \leftarrow \{j \in [1,n] : v_j(i) \neq u_1(i) = u_2(i) = \cdots = u_n(i)\}$
end
$I_0 \leftarrow \emptyset$
$C_0 \leftarrow \emptyset$
$x \leftarrow 0$
repeat
    $i_x \leftarrow \operatorname{argmax} \{\mathrm{Card}(Y_i \setminus C_x) : i \in X \setminus I_x\}$ ;
    $I_{x+1} \leftarrow I_x \cup \{i_x\}$ ;
    $C_{x+1} \leftarrow C_x \cup Y_{i_x}$ ;
    $x \leftarrow x + 1$ ;
until $C_x = [1,n]$ or $I_x = X$;
if $C_x = [1,n]$ then
    return The pattern corresponding to $I_x$
else
    return No separating pattern with last position $\ell$
end
```

Proof Let $u_1, \dots, u_n, v_1, \dots, v_n$ be a set of $2n$ words of length ℓ. Thanks to Lemma 2 we are looking for a separating pattern:

$$P = \mathbf{X}^{i_1-1}(c_1 \wedge \mathbf{X}^{i_2-i_1}(\cdots \wedge \mathbf{X}^{i_p-i_{p-1}}c_p)\cdots).$$

For a pattern P we define $I(P) = \{i_1, i_2, \dots, i_p\}$. Note that $\mathbf{last}(P) = \max I(P)$ and $\mathbf{width}(P) = \mathrm{Card}(I(P))$.

We define the set $X = \{i \in [1,\ell] : \exists c \in \Sigma, \forall j \in [1,n], u_j(i) = c\}$ holding the indices where the positive words all agree. If P satisfies $u_1, \dots, u_n$, then $I(P) \subseteq X$. Further, given $I \subseteq X$, we can construct a pattern P such that $I(P) = I$ and P satisfies $u_1, \dots, u_n$: we simply choose $c_q = u_1(i_q) = \cdots = u_n(i_q)$ for $q \in [1,p]$. We call P the pattern corresponding to I.

Recall that the size of the pattern P is $\mathbf{last}(P) + 2(\mathbf{width}(P) - 1)$. This makes the task of minimising it difficult: there is a trade-off between minimising the last position $\mathbf{last}(P)$ and the width $\mathbf{width}(P)$.

Let us consider the following easier problem: construct a $\log(n)$-approximation of a minimal separating pattern with last position equal to the length of the words. Assuming we have such an algorithm, we obtain a $\log(n)$-approximation of a minimal separating pattern by running the previous algorithm on prefixes of length ℓ' for each $\ell' \in [1,\ell]$.

We now focus on the question of constructing a $\log(n)$-approximation of a minimal separating pattern with last position equal to the length of the words. We refer to Algorithm 1 for the pseudocode. For a set I, we write $C_I = \bigcup \{Y_i : i \in I\}$: the pattern corresponding to I does not satisfy v_j if and only if $j \in C_I$. In particular, the pattern corresponding to I is separating if and only if $C_I = [1,n]$.

The algorithm constructs a set I incrementally through the sequence $(I_x)_{x \geq 0}$, with the following easy invariant: for $x \geq 0$, we have $C_x = C_{I_x}$. The algorithm is greedy: I_x is augmented with $i \in X \setminus I_x$ maximising the number of word indices added to C_x by adding i, which is the cardinality of $Y_i \setminus C_x$.

We now prove that this yields a $\log(n)$-approximation algorithm. Let P_{opt} be a minimal separating pattern with last position ℓ, inducing $I_{\text{opt}} = I(P_{\text{opt}}) \subseteq [1, \ell]$ of cardinal m. Note that $C_{I_{\text{opt}}} = [1, n]$.

We let $n_x = n - |C_x|$ and show the following by induction on $x \geq 0$:

$$n_{x+1} \leq n_x \cdot \left(1 - \frac{1}{m}\right) = n_x \cdot \frac{m-1}{m}.$$

We claim that there exists $i \in X \setminus I_x$ such that $\text{Card}(Y_i \setminus C_x) \geq \frac{n_x}{m}$. Indeed, assume towards contradiction that for all $i \in X \setminus I_x$ we have $\text{Card}(Y_i \setminus C_x) < \frac{n_x}{m}$. In this case, for any case subset I of cardinality m, $I' = I \cup I_x$ is such that $|C_{I'}| < |C_x| + |I| \cdot \frac{n_x}{m} \leq (n - n_x) + m\frac{n_x}{m} \leq n$, contradicting the existence of I_{opt}. Thus there exists $i \in X \setminus I_x$ such that $\text{Card}(Y_i \setminus C_x) \geq \frac{n_x}{m}$, implying that the algorithm chooses such an i and $n_{x+1} \leq n_x - \frac{n_x}{m} = n_x \cdot \left(1 - \frac{1}{m}\right)$.

The proved inequality implies $n_x \leq n \cdot \left(1 - \frac{1}{m}\right)^x$. This quantity is less than 1 for $x \geq \log(n) \cdot m$, implying that the algorithm stops after at most $\log(n) \cdot m$ steps. Consequently, the pattern corresponding to I has size at most $\log(n) \cdot |P_{\text{opt}}|$, completing the claim on approximation.

A naive complexity analysis yields an implementation of Algorithm 1 running in time $O(n \cdot \ell)$, leading to an overall complexity of $O(n \cdot \ell^2)$ by running Algorithm 1 on the prefixes of length ℓ' of $u_1, \ldots, u_n, v_1, \ldots, v_n$ for each $\ell' \in [1, \ell]$. ■

Hardness results

The algorithm we just constructed implied a $\log(n)$-approximation, begging the question whether this can be improved. We prove here, conditionally to very standard complexity assumptions, that the approximation ratio is optimal.

Theorem 5 *The* $\textbf{LTL}(\mathbf{X}, \wedge)$ *learning problem is* NP*-hard, and there are no* $(1 - o(1)) \cdot \log(n)$ *polynomial time approximation algorithms unless* P = NP*, even for a single positive word.*

Note that Theorem 4 and Theorem 5 yield matching upper and lower bounds on approximation algorithms for learning $\textbf{LTL}(\mathbf{X}, \wedge)$.

The hardness result stated in Theorem 5 follows from a reduction to the set cover problem, which we define now. The set cover decision problem is: given $S_1, \ldots, S_\ell$ subsets of $[1, n]$ and $k \in \mathbb{N}$, does there exists $I \subseteq [1, \ell]$ of size at most k such that $\bigcup_{i \in I} S_i = [1, n]$? In that case we say that I is a cover. An α-approximation algorithm returns a cover of size at most $\alpha \cdot k$ where k is the size of a minimal cover. The following results form the state of the art for solving exact and approximate variants of the set cover problem.

Theorem 6 (Dinur and Steurer (2014)) *The set cover problem is* NP*-complete, and there are no* $(1 - o(1)) \cdot \log(n)$ *polynomial time approximation algorithms unless* $\mathsf{P} = \mathsf{NP}$.

We proceed with proving Theorem 5.

Proof We construct a reduction from set cover. Let $S_1, \dots, S_\ell$ be subsets of $[1, n]$ and $k \in \mathbb{N}$.

Let us consider the word $u = a^{\ell+1}$, and for each $j \in [1, n]$ and $i \in [1, \ell]$, writing $v_j(i)$ for the i^{th} letter of v_j:

$$v_j(i) = \begin{cases} b \text{ if } j \in S_i, \\ a \text{ if } j \notin S_i, \end{cases}$$

and we set $v_j(\ell + 1) = a$ for any $j \in [1, n]$. We also add $v_{n+1} = a^\ell b$.

We claim that there is a cover of size k if and only if there is a formula of size $\ell + 2k - 1$ separating u from $v_1, \dots, v_{n+1}$.

Thanks to Lemma 2 we can restrict our attention to patterns, *i.e* formulas of the form (we adjust the indexing for technical convenience)

$$\phi = \mathbf{X}^{i_1 - 1}(c_1 \wedge \mathbf{X}^{i_2 - i_1}(\cdots \wedge \mathbf{X}^{i_{p+1} - i_p} c_{p+1}) \cdots),$$

for some positions $i_1 \leq \cdots \leq i_{p+1}$ and letters $c_1, \dots, c_{p+1} \in \Sigma$. If ϕ satisfies u, then necessarily $c_1 = \cdots = c_{p+1} = a$. This implies that if ϕ does not satisfy v_{n+1}, then necessarily $i_{p+1} = \ell + 1$.

We associate to ϕ the set $I = \{i_1 \leq \cdots \leq i_p\}$. Note that ϕ is equivalent to the larger formula $\bigwedge_{q \in [1,p]} \mathbf{X}^{i_q - 1} a \wedge \mathbf{X}^\ell a$, and its size is $\ell + 1 + 2(|I| - 1)$.

By construction, ϕ separates u from $v_1, \dots, v_{n+1}$ if and only if I is a cover. Indeed, I is a cover if and only if for every $j \in [1, n]$ there exists $i \in I$ such that $j \in S_i$. This is equivalent to the fact that for every $j \in [1, n]$ we have $v_j \not\models \phi$. ■

4. LTL(F, ∧)

As we will see, **LTL**$(\mathbf{F}, \wedge)$ over an alphabet of size 2 is very weak. This degeneracy vanishes when considering alphabets of size at least 3. Let us fix a (finite) alphabet Σ.

Minimal formulas

Instead of defining a normal form as we did for **LTL**$(\mathbf{X}, \wedge)$ we characterise the expressive power of **LTL**$(\mathbf{F}, \wedge)$ and construct for each property expressible in this logic a minimal formula.

Let us consider two words $u = u(1) \dots u(\ell')$ and $v = v(1) \dots v(\ell)$. We say that u is a subsequence of v if there exists $\phi : [1, \ell'] \to [1, \ell]$ increasing such that $v(\phi(i)) = u(i)$. For example *abba* is a subsequence of *b***ab***aaaa***ba**. We say that a word is non-repeating if every two consecutive letters are different.

Lemma 7 *For every formula* $\phi \in$ ***LTL***$(\boldsymbol{F}, \wedge)$, *either it is equivalent to false or there exists a finite set of non-repeating words* $w_1, \dots, w_p$ *and* $c \in \Sigma \cup \{\varepsilon\}$ *such that for every word* z,

$$z \models \phi \textit{ if and only if } \begin{cases} \textit{for all } q \in [1, p], w_q \textit{ is a subsequence of } z, \\ \textit{and } z \textit{ starts with } c. \end{cases}$$

Lemma 7 gives a characterisation of the properties expressible in $\mathbf{LTL}(\mathbf{F}, \wedge)$. It implies that over an alphabet of size 2 the fragment $\mathbf{LTL}(\mathbf{F}, \wedge)$ is very weak. Indeed, there are very few non-repeating words over the alphabet $\Sigma = \{a, b\}$: only prefixes of $abab\ldots$ and $baba\ldots$. This implies that formulas in $\mathbf{LTL}(\mathbf{F}, \wedge)$ over $\Sigma = \{a, b\}$ can only place lower bounds on the number of alternations between a and b (starting from a or from b) and check whether the word starts with a or b. In particular, the $\mathbf{LTL}(\mathbf{F}, \wedge)$ learning problem over this alphabet is (almost) trivial and thus not interesting. Hence we now assume that Σ has size at least 3.

We move back from semantics to syntax, and show how to construct minimal formulas. Let $w_1, \ldots, w_p$ be a finite set of non-repeating words and $c \in \Sigma \cup \{\varepsilon\}$. We define a formula ϕ as follows.

Let us first associate to $w_1, \ldots, w_p$ an acyclic directed graph: the set of vertices is the set of words w which are prefixes of some $w_1, \ldots, w_p$, and there is an edge from w to wc for any word w and letter c. Note that the graph is acyclic (since following an edge we add a letter), so it is a forest, meaning a set of trees. We interpret each tree t as a formula ϕ_t in $\mathbf{LTL}(\mathbf{F}, \wedge)$ as follows, in an inductive fashion: if the tree consists of only a leaf wa, then we let $\phi_t = a$, and otherwise let wa the root of t and $t_1, \ldots, t_q$ its subtrees, then

$$\phi_t = \mathbf{F}(a \wedge \bigwedge_i \phi_{t_i}).$$

As an example, consider the set of words ab, ac, bab. The forest corresponding to ab, ac, bab contains two trees: one contains the nodes b, ba, bab, and the other one the nodes a, ab, ac. The two corresponding formulas are

$$\mathbf{F}(b \wedge \mathbf{F}(a \wedge \mathbf{F}b)) \qquad ; \qquad \mathbf{F}(a \wedge \mathbf{F}b \wedge \mathbf{F}c).$$

To obtain the minimal formula equivalent to ϕ, we make the following case distinction: if $c = \varepsilon$, then the formula associated to $w_1, \ldots, w_p$ and c is the conjunction of the formulas for each tree of the forest, and if $c \in \Sigma$, then the formula additionally has a conjunct c. For instance, the formula corresponding to the set of words ab, ac, bab, and the letter a is

$$a \ \wedge \ \mathbf{F}(b \wedge \mathbf{F}(a \wedge \mathbf{F}b)) \ \wedge \ \mathbf{F}(a \wedge \mathbf{F}b \wedge \mathbf{F}c).$$

Lemma 8 *For every set of non-repeating words $w_1, \ldots, w_p$ and $c \in \Sigma \cup \{\varepsilon\}$, the formula ϕ constructed above is minimal, meaning there are no smaller equivalent formulas.*

Applying the construction above to a single non-repeating word $w = c_1 \ldots c_p$ we obtain what we call a "**f**attern" (pattern with an F):

$$F = \mathbf{F}(c_1 \wedge \mathbf{F}(\cdots \wedge \mathbf{F}c_p) \cdots),$$

We say that the non-repeating word w induces the fattern F above, and conversely that the fattern F induces the word w. The size of a fattern F is $3|w| - 1$. Adding the initial letter we obtain a grounded fattern $c \wedge F$, in that case the letter c is added at the beginning of w and the size is $3|w| - 2$.

A first corollary of Lemmas 7 and 8 is a non-deterministic polynomial time algorithm.

Theorem 9 *The learning problem for $\boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ is in* NP.

The following two normalisation lemmas show the role of fatterns for separating formulas in $\mathbf{LTL}(\mathbf{F}, \wedge)$.

Lemma 10 *Let $u_1, \ldots, u_n, v_1, \ldots, v_n$ be a set of $2n$ words. If there exists $\phi \in \boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ separating $u_1, \ldots, u_n$ from $v_1, \ldots, v_n$, then there exists a conjunction of at most n fatterns separating $u_1, \ldots, u_n$ from $v_1, \ldots, v_n$.*

Lemma 11 *Let $u, v_1, \ldots, v_n$ be a set of $n + 1$ words. If there exists $\phi \in \boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ separating u from $v_1, \ldots, v_n$, then there exists a fattern of size smaller than or equal to ϕ separating u from $v_1, \ldots, v_n$.*

A dynamic programming algorithm

Let us define the following intermediate problem called shortest subsequence: given a set of $2n$ words $u_1, \ldots, u_n, v_1, \ldots, v_n$, the goal is to find the shortest word w such that for all $j \in [1, n]$, w is a subsequence of u_j and not a subsequence of v_j.

Lemma 10 and Lemma 11 imply that learning $\mathbf{LTL}(\mathbf{F}, \wedge)$ in both cases of a single positive word and a single negative word is equivalent to the shortest subsequence problem, since minimising the size of a fattern is equivalent to minimising the size of the word it induces. In particular, this implies that the shortest subsequence problem is in NP. Let us construct an algorithm for solving the shortest subsequence problem and then discuss its consequences for learning $\mathbf{LTL}(\mathbf{F}, \wedge)$.

Lemma 12 *There exists an algorithm solving the shortest subsequence problem running in time $O(n \cdot (2^\ell + |\Sigma| \cdot \ell))$.*

Proof We use Python-inspired notation for suffixes: we let $w(k :)$ denote the word obtained from w starting at position k.

Let us write $\bar{i} = (i_1, i_2, \ldots, i_n, i'_1, i'_2, \ldots, i'_n)$ for a tuple of positions in each of the $2n$ words. We include the special position ω and for w a word, a a letter and i a position we define $\mathbf{ind}(w, a, i) = \min\{i' : w(i') = a, i' \geq i\}$ with the convention that $\mathbf{ind}(w, a, i) = \omega$ if there is no such i'. Let $\mathbf{R}(\bar{i})$ be the length of a shortest word w such that for all $j \in [1, n]$, w is a subsequence of $u_j(i_j :)$ and not a subsequence of $v_j(i'_j :)$. We construct a dynamic programming algorithm populating the table $\mathbf{R}$; the goal is to compute $\mathbf{R}(1, 1, \ldots, 1)$.

The key equality on which the algorithm relies is

$$\mathbf{R}(\bar{i}) = \min \begin{cases} 1 + \mathbf{R}(i_1 + 1, ni_2, \ldots, ni_n, ni'_1, \ldots, ni'_n) \\ \mathbf{R}(i_1 + 1, i_2, \ldots, i_n, i'_1, \ldots, i'_n) \end{cases},$$

where $ni_j = \mathbf{ind}(u_j, u_1(i_1), i_j)$ and $ni'_j = \mathbf{ind}(v_j, u_1(i_1), i'_j)$. It corresponds to the following case distinction: we consider the shortest subsequence w from $\bar{i}$ together with the functions ϕ_i mapping w to each $u_1, \ldots, u_n, v_1, \ldots, v_n$. Then

- either $\phi_1(1) = i_1$, and then necessarily $\phi_i(1) \geq ni_i$ for $i \in [2, n]$ and $\phi_{i'}(1) \geq ni'_i$, so $w(2 :)$ is the shortest subsequence starting from $(i_1 + 1, ni_2, \ldots, ni_n, ni'_1, \ldots, ni'_n)$,

- or $\phi_1(1) > i_1$, and then w is the shortest subsequence starting from the tuple $(i_1 + 1, i_2, \ldots, i_n, i'_1, \ldots, i'_n)$.

Complexity analysis. There are at most 2^ℓ subsequences and processing each is done in time $O(n)$ since we need to query the values $\mathbf{ind}(u_j, u_1(i_1), i_j)$ and $\mathbf{ind}(v_j, u_1(i_1), i'_j)$ for $j \in [1, n]$. The naive algorithm to compute all the values $\mathbf{ind}(w, c, i)$ runs in time $O(n \cdot |\Sigma| \cdot \ell^2)$ but this can be easily reduced to a running time of $O(n \cdot |\Sigma| \cdot \ell)$. ■

We now show how to instantiate the previous algorithm for learning $\mathbf{LTL}(\mathbf{F}, \wedge)$.

Theorem 13

- *There exists a $O(n \cdot (2^\ell + |\Sigma| \cdot \ell))$ time algorithm for learning $\boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ with a single negative word.*
- *There exists a $O(n \cdot (2^\ell + |\Sigma| \cdot \ell))$ time algorithm for learning $\boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ with a single positive word.*
- *There exists a $O(n^2 \cdot (2^\ell + |\Sigma| \cdot \ell)))$ time n-approximation algorithm for learning $\boldsymbol{LTL}(\boldsymbol{F}, \wedge)$.*

Hardness results

Theorem 14 *The $\boldsymbol{LTL}(\boldsymbol{F}, \wedge)$ learning problem is NP-hard, and there are no $(1 - o(1)) \cdot \log(n)$ polynomial time approximation algorithms unless $\mathsf{P} = \mathsf{NP}$, even with a single positive word.*

The result follows from a reduction from the hitting set problem. The hitting set decision problem is: given $C_1, \ldots, C_n$ subsets of $[1, \ell]$ and $k \in \mathbb{N}$, does there exist H subset of $[1, \ell]$ of size at most k such that for every $j \in [1, n]$ we have $H \cap C_j \neq \emptyset$. In that case we say that H is a hitting set.

The hitting set problem is an equivalent formulation of the set cover problem, but it is here technically more convenient to construct a reduction from the hitting set problem. The hardness results stated in Theorem 6 apply to the hitting set problem.

We can now prove Theorem 14.

Proof We construct a reduction from the hitting set problem. Let $C_1, \ldots, C_n$ be subsets of $[1, \ell]$ and $k \in \mathbb{N}$. Let us consider the alphabet $[0, \ell]$. We define the word $u = 012 \ldots \ell$. For each $j \in [1, n]$ we let $[1, \ell] \setminus C_j = \left\{a_{j,1} < \cdots < a_{j,m_j}\right\}$, and define $v_j = 0 a_{j,1} \ldots a_{j,m_j}$.

We claim that there exists a hitting set of size at most k if and only if there exists a formula in $\mathbf{LTL}(\mathbf{F}, \wedge)$ of size at most $3k - 1$ separating u from $v_1, \ldots, v_n$.

Let $H = \{c_1, \ldots, c_k\}$ be a hitting set of size k with $c_1 < c_2 < \cdots < c_k$. We construct the (non-grounded) fattern induced by $w = c_1 \ldots c_k$, it separates u from $v_1, \ldots, v_n$ and has size $3k - 1$.

Conversely, let ϕ be a formula in $\mathbf{LTL}(\mathbf{F}, \wedge)$ of size $3k - 1$ separating u from $v_1, \ldots, v_n$. Thanks to Lemma 11 we can assume that ϕ is a fattern. Let $w = c_1 \ldots c_k$ be the non-repeating word it induces. Necessarily $c_1 < c_2 < \cdots < c_k$. If ϕ is grounded then $c_1 = 0$, but then the (non-grounded) fattern induced by $c_2 \ldots c_k$ is also separating, so we can assume that ϕ is not grounded. We let $H = \{c_1, \dots, c_k\}$, and argue that H is a hitting set. Indeed, H is a hitting set if and only if for every $j \in [1, n]$ we have $H \cap C_j \neq \emptyset$, which is equivalent to for every $j \in [1, n]$ we have $v_j \not\models \phi$; indeed for $c_i \in H \cap C_j$ by definition c_i does not appear in v_j so $v_j \not\models \mathbf{F} c_i$. ■

5. $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$

Theorem 15 *The learning problem for $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ is in* NP*.*

A closer inspection at this proof shows that the argument applies to any fragment containing $\mathbf{X}, \wedge$, and $\vee$; in particular this shows that the learning problem for $\mathbf{LTL} = \mathbf{LTL}(\mathbf{G}, \mathbf{F}, \mathbf{X}, \wedge, \vee)$ is in NP.

Hardness result

We show that the reduction constructed in Section 3 extends to $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$.

Theorem 16 *The $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ learning problem is* NP*-hard, and there are no $(1 - o(1)) \cdot \log(n)$ polynomial time approximation algorithms unless $\mathsf{P} = \mathsf{NP}$, even for a single positive word.*

To prove Theorem 16, we show that the reduction constructed in Theorem 5 is also a reduction from set cover to the $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ learning problem.

To prove this result we need a reduction lemma for disjunctions that we state now. Let $\phi \in \mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$. We define $D(\phi) \subseteq \mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge)$ by induction:

- If $\phi = c$ then $D(\phi) = \{c\}$.
- If $\phi = \phi_1 \wedge \phi_2$ then $D(\phi) = \{\psi_1 \wedge \psi_2 : \psi_1 \in D(\phi_1), \psi_2 \in D(\phi_2)\}$.
- If $\phi = \phi_1 \vee \phi_2$ then $D(\phi) = D(\phi_1) \cup D(\phi_2)$.
- If $\phi = \mathbf{X}\phi'$ then $D(\phi) = \{\mathbf{X}\psi : \psi \in D(\phi')\}$.
- If $\phi = \mathbf{F}\phi'$ then $D(\phi) = \{\mathbf{F}\psi : \psi \in D(\phi')\}$.

Lemma 17 *For any $u, v_1, \dots, v_n$, if ϕ separates u from $v_1, \dots, v_n$, then there exists $\psi \in D(\phi)$ which separates u from $v_1, \dots, v_n$.*

6. Open problems

Open problem 1:
Does there exist a polynomial time $o(n)$-approximation algorithm for learning $\mathbf{LTL}(\mathbf{F}, \wedge)$?

We have proved that the learning problem is NP-complete for $\mathbf{LTL}(\mathbf{X}, \wedge), \mathbf{LTL}(\mathbf{F}, \wedge)$, and $\mathbf{LTL}(\mathbf{F}, \mathbf{X}, \wedge, \vee)$. These complexity results do not extend to larger fragments: the complexity might be either lower or higher, since the logic is a priori more expressive. In this paper we restricted ourselves to the temporal operators $\mathbf{X}$ and $\mathbf{F}$; let us call 'full

LTL' the logic obtained by adding the globally operator **G** and the until operator **U**. The reduction used for proving the last result does not extend to full **LTL** (indeed **G**a separates u from $v_1, \ldots, v_{n+1}$).

Open problem 2:
Is the learning problem NP-complete for full **LTL**?

Acknowledgments

We thank Daniel Neider for introducing us to this fascinating question, and the reviewers for their very useful comments.

References

Giuseppe Bombara, Cristian Ioan Vasile, Francisco Penedo Alvarez, Hirotoshi Yasuoka, and Calin Belta. A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Hybrid Systems: Computation and Control, HSCC*, 2016. URL https://doi.org/10.1145/2883817.2883843.

Alberto Camacho and Sheila A. McIlraith. Learning interpretable models expressed in linear temporal logic. *International Conference on Automated Planning and Scheduling, ICAPS*, 29, 2019. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3529.

Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing, STOC*, pages 624–633, 2014. URL https://doi.org/10.1145/2591796.2591884.

Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. Learning properties in LTL $\cap$ ACTL from positive examples only. In *Formal Methods in Computer Aided Design, FMCAD*, 2020. URL https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_17.

Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2008. URL http://algo.inria.fr/flajolet/Publications/AnaCombi/anacombi.html.

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

Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *International Joint Conference on Artificial Intelligence, IJCAI*, 2019. URL https://doi.org/10.24963/ijcai.2019/776.

Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL specification mining. In *International Conference on Automated Software Engineering, ASE*, 2015. URL https://doi.org/10.1109/ASE.2015.71.

Daniel Neider and Ivan Gavran. Learning linear temporal properties. In *Formal Methods in Computer Aided Design, FMCAD*, 2018. URL https://doi.org/10.23919/FMCAD.2018.8603016.

Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM*, 40(1):95–142, 1993. URL https://doi.org/10.1145/138027.138042.

Amir Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science, SFCS*, 1977. URL https://doi.org/10.1109/SFCS.1977.32.

Rajarshi Roy, Dana Fisman, and Daniel Neider. Learning interpretable models in the property specification language. In *International Joint Conference on Artificial Intelligence, IJCAI*, 2020. URL https://doi.org/10.24963/ijcai.2020/306.