Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic[⋆]

Ritam Raha[1,2](✉), Rajarshi Roy[3], Nathanaël Fijalkow[2,4], and Daniel Neider[3]

[1] University of Antwerp, Antwerp, Belgium
ritam.raha@uantwerpen.be
[2] CNRS, LaBRI and Université de Bordeaux, France
nathanael.fijalkow@labri.fr
[3] Max Planck Institute for Software Systems, Kaiserslautern, Germany
{rajarshi,neider}@mpi-sws.org
[4] The Alan Turing Institute of data science, United Kingdom

Abstract. Linear temporal logic (LTL) is a specification language for finite sequences (called traces) widely used in program verification, motion planning in robotics, process mining, and many other areas. We consider the problem of learning formulas in fragments of LTL without the **U**-operator for classifying traces; despite a growing interest of the research community, existing solutions suffer from two limitations: they do not scale beyond small formulas, and they may exhaust computational resources without returning any result. We introduce a new algorithm addressing both issues: our algorithm is able to construct formulas an order of magnitude larger than previous methods, and it is anytime, meaning that it in most cases successfully outputs a formula, albeit possibly not of minimal size. We evaluate the performances of our algorithm using an open source implementation against publicly available benchmarks.

Keywords: Linear Temporal Logic · Artificial Intelligence · Specification Mining

1 Introduction

Linear Temporal Logic (LTL) is a prominent logic for specifying temporal properties [20] over infinite traces, and recently introduced over finite traces [6]. In this paper, we consider finite traces but, in a small abuse of notations, call this logic LTL as well. It has become a de facto standard in many fields such as model checking, program analysis, and motion planning for robotics. Over the past five to ten years learning temporal logics (of which LTL is the core) has become an active research area and identified as an important goal in artificial intelligence:

[⋆] The first two authors contributed equally, the remaining authors are ordered alphabetically. This research was funded by the FWO G030020N project "SAILor" and Deutsche Forschungsgemeinschaft (DFG) grant no. 434592664.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 263–280, 2022.
https://doi.org/10.1007/978-3-030-99524-9_14

it formalises the difficult task of building explainable models from data. Indeed, as we will see in the examples below and as argued in the literature, e.g., by [4] and [24], LTL formulas are typically easy to interpret by human users and therefore useful as explanations. The variable free syntax of LTL and its natural inductive semantics make LTL a natural target for building classifiers separating positive from negative traces.

The fundamental problem we study here, established in [25], is to build an explainable model in the form of an LTL formula from a set of positive and negative traces. More formally (we refer to the next section for formal definitions), given a set $u_1, \ldots, u_n$ of positive traces and a set $v_1, \ldots, v_n$ of negative traces, the goal is to construct a formula φ of LTL which satisfies all u_i's and none of the v_i's. In that case, we say that φ is a separating formula or—using machine learning terminology—a classifier.

To make things concrete let us introduce our running example, a classic motion planning problem in robotics and inspired by [15]. A robot collects wastebin contents in an office-like environment and empties them in a trash container. Let us assume that there is an office o, a hallway h, a container c and a wet area w. The following are possible traces obtained in experimentation with the robot (for instance, through simulation):

$$\begin{aligned} u_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \\ v_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{h} \end{aligned}$$

In LTL learning we start from these labelled data: given u_1 as positive and v_1 as negative, what is a possible classifier including u_1 but not v_1? Informally, v_1 being negative implies that the order is fixed: o must be visited before c. We look for classifiers in the form of separating formulas, for instance

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}),$$

where the **F**-operator stands for “finally” and **X** for “next”. Note that this formula requires to visit the office first and only then visit the container.

Assume now that two more negative traces were added:

$$\begin{aligned} v_2 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \\ v_3 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{w} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{w} \cdot \mathrm{w} \end{aligned}$$

Then the previous separating formula is no longer correct, and a possible separating formula is

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}) \wedge \mathbf{G}(\neg \mathrm{w}),$$

which additionally requires the robot to never visit the wet area. Here the **G**-operator stands for “globally”.

Let us emphasise at this point that for the sake of presentation, we consider only exact classifiers: a separating formula must satisfy all positive traces and none of the negative traces. However, our algorithm naturally extends to the noisy data setting where the goal is to construct an approximate classifier, replacing ‘all’ and ‘none’ by ‘almost all’ and ‘almost none’.

State of the art. A number of different approaches have been proposed, leveraging SAT solvers [19], automata [4], and Bayesian inference [16], and extended to more expressive logics such as Property Specification Language (PSL) [24] and Computational Tree Logic (CTL) [9].

Applications include program specification [17], anomaly and fault detection [3], robotics [5], and many more: we refer to [4], Section 7, for a list of practical applications. An equivalent point of view on LTL learning is as a specification mining question. The ARSENAL [13] and FRET [14] projects construct LTL specifications from natural language, we refer to [18] for an overview.

Existing methods do not scale beyond formulas of small size, making them hard to deploy for industrial cases. A second serious limitation is that they often exhaust computational resources without returning any result. Indeed theoretical studies [11] have shown that constructing the minimal LTL formula is NP-hard already for very small fragments of LTL, explaining the difficulties found in practice.

Our approach. To address both issues, we turn to *approximation* and *anytime* algorithms. Here *approximation* means that the algorithm does not ensure minimality of the constructed formula: it does ensure that the output formula separates positive from negative traces, but it may not be the smallest one. On the other hand, an algorithm solving an optimisation problem is called *anytime* if it finds better and better solutions the longer it keeps running. In other words, anytime algorithms work by refining solutions. As we will see in the experiments, this implies that even if our algorithm timeouts it may yield some good albeit non-optimal formula.

Our algorithm targets a strict fragment of LTL, which does not contain the Until operator (nor its dual Release operator). It combines two ingredients:

- *Searching for directed formulas*: we define a space efficient dynamic programming algorithm for enumerating formulas from a fragment of LTL that we call Directed LTL.
- *Combining directed formulas*: we construct two algorithms for combining formulas using Boolean operators. The first is an off-the-shelf *decision tree algorithm*, and the second is a new greedy algorithm called *Boolean subset cover*.

The two ingredients yield two subprocedures: the first one finds directed formulas of increasing size, which are then fed to the second procedure in charge of combining them into a separating formula. This yields an anytime algorithm as both subprocedures can output separating formulas even with a low computational budget and refine them over time.

Let us illustrate the two subprocedures in our running example. The first procedure enumerates so-called *directed formulas* in increasing size; we refer to the corresponding section for a formal definition. The directed formulas $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg \mathrm{w})$ have small size hence will be generated early on. The second procedure constructs formulas as Boolean combinations of directed formulas. Without getting into the details of the algorithms, let us note that both $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg \mathrm{w})$ satisfy u_1. The first does not satisfy v_1 and the second does not satisfy v_2 and v_3. Hence their conjunction $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}) \wedge \mathbf{G}(\neg \mathrm{w})$ is separating, meaning it satisfies u_1 but none of v_1, v_2, v_3.

Outline. The mandatory definitions and the problem statement we deal with are described in Section 2. Section 3 shows a high-level overview of our main idea in the algorithm. The next two sections, Section 4 and Section 5 describe the two phases of our algorithm in details, in one section each. We discuss the theoretical guarantees of our algorithm in Section 6. We conclude with an empirical evaluation in Section 7.

2 Preliminaries

Traces. Let $\mathcal{P}$ be a finite set of atomic propositions. An *alphabet* is a finite non-empty set $\Sigma = 2^{\mathcal{P}}$, whose elements are called *symbols*. A finite *trace* over Σ is a finite sequence $t = a_1 a_2 \dots a_n$ such that for every $1 \leq i \leq n$, $a_i \in \Sigma$. We say that t has length n and write $|t| = n$. For example, let $\mathcal{P} = \{p, q\}$, in the trace $t = \{p, q\} \cdot \{p\} \cdot \{q\}$ both p and q hold at the first position, only p holds in the second position, and q in the third position. Note that, throughout the paper, we only consider finite traces.

A trace is a *word* if exactly one atomic proposition holds at each position: we used words in the introduction example for simplicity, writing $h \cdot o \cdot c$ instead of $\{h\} \cdot \{o\} \cdot \{c\}$.

Given a trace $t = a_1 a_2 \dots a_n$ and $1 \leq i \leq j \leq n$, let $t[i, j] = a_i \dots a_j$ be the *infix* of t from position i up to and including position j. Moreover, $t[i] = a_i$ is the symbol at the i^{th} position.

Linear Temporal Logic. The syntax of Linear Temporal Logic (LTL, in short) is defined by the following grammar

$$\varphi := p \in \mathcal{P} \mid \neg p \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathbf{X}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{G}\,\varphi \mid \varphi\,\mathbf{U}\,\psi$$

We use the standard formulas: *true* $= p \vee \neg p$, *false* $= p \wedge \neg p$ and **last** $= \neg\,\mathbf{X}\,$*true*, which denotes the last position of the trace. As a shorthand, we use $\mathbf{X}^n\,\varphi$ for $\underbrace{\mathbf{X} \dots \mathbf{X}}_{n \text{ times}}\varphi$.

The *size of a formula* is the size of its underlying syntax tree.

Formulas in LTL are evaluated over finite traces. To define the semantics of LTL we introduce the notation $t, i \models \varphi$, which reads 'the LTL formula φ holds over trace t from position i'. We say that t satisfies φ and we write $t \models \varphi$ when $t, 1 \models \varphi$. The definition of $\models$ is inductive on the formula φ:

- $t, i \models p \in \mathcal{P}$ if $p \in t[i]$.

- $t, i \models \mathbf{X}\,\varphi$ if $i < |t|$ and $t, i+1 \models \varphi$. It is called the ne**X**t operator.
- $t, i \models \mathbf{F}\,\varphi$ if $t, i' \models \varphi$ for some $i' \in [i, |t|]$. It is called the eventually operator (F comes from **F**inally).
- $t, i \models \mathbf{G}\,\varphi$ if $t, i' \models \varphi$ for all $i' \in [i, |t|]$. It is called the **G**lobally operator.
- $t, i \models \varphi\,\mathbf{U}\,\psi$ if $t, j \models \psi$ for some $i \leq j \leq |t|$ and $t, i' \models \varphi$ for all $i \leq i' < j$. It is called the **U**ntil operator.

The LTL Learning Problem. The LTL exact learning problem studied in this paper is the following: given a set P of positive traces and a set N of negative traces, construct a minimal LTL separating formula φ, meaning such that $t \models \varphi$ for all $t \in P$ and $t \not\models \varphi$ for all $t \in N$.

There are two relevant parameters for a sample: its *size*, which is the number of traces, and its *length*, which is the maximum length of all traces.

The problem is naturally extended to the LTL noisy learning problem where the goal is to construct an ε-separating formula, meaning such that φ satisfies all but an ε proportion of the traces in P and none but an ε proportion of the traces in N. For the sake of simplicity we present an algorithm for solving the LTL exact learning problem, and later sketch how to extend it to the noisy setting.

3 High-level view of the algorithm

Let us start with a naive algorithm for the LTL Learning Problem. We can search through all LTL formulas in some order and check whether they are separating for our sample or not. Checking whether an LTL formula is separating can be done using standard methods (for e.g. using bit vector operations [2]). However, the major drawback of this idea is that we have to search through all LTL formulas, which is hard as the number of LTL formulas grows very quickly[5].

To tackle this issue, instead of the entire LTL fragment, our algorithm (as outlined in Algorithm 1) performs an iterative search through a fragment of LTL, which we call Directed LTL (Line 4). We expand upon this in Section 4. In that section, we also describe how we can iteratively generate these Directed LTL formulas in a particular "size order" (not the usual size of an LTL formula) and evaluate these formulas over the traces in the sample efficiently using dynamic programming techniques.

To include more formulas in our search space, we generate and search through Boolean combinations of the most promising formulas of Directed LTL formulas (Line 11), which we describe in detail in Section 5. Note that, the fragment of LTL that our algorithm searches through ultimately does not include formulas with **U** operator. Thus, for readability, we use LTL to refer to the fragment LTL $\setminus$ **U** in the rest of the paper.

During the search of formulas, our algorithm searches for smaller separating formulas at each iteration than the previously found ones, if any. In fact, as a

[5] The number of LTL formulas of size k is asymptotically equivalent to $\frac{\sqrt{14}\cdot 7^k}{2\sqrt{\pi k^3}}$ [12]

Algorithm 1 Overview of our algorithm

```
 1: $B \leftarrow \emptyset$
 2: $\psi \leftarrow \emptyset$: best formula found
 3: for all $s$ in "size order" do
 4:     $D \leftarrow$ all Directed LTL formulas of parameter $s$
 5:     for all $\varphi \in D$ do
 6:         if $\varphi$ is separating and smaller than $\psi$ then
 7:             $\psi \leftarrow \varphi$
 8:         end if
 9:     end for
10:     $B \leftarrow B \cup D$
11:     $B \leftarrow$ Boolean combinations of the promising formulas in $B$
12:     for all $\varphi \in B$ do
13:         if $\varphi$ is separating and smaller than $\psi$ then
14:             $\psi \leftarrow \varphi$
15:         end if
16:     end for
17: end for
18: Return $\psi$
```

heuristic, once a separating formula is found, we only search through formulas that are smaller than the found separating formula. Such a heuristic, along with aiding the search for minimal formulas, also reduces the search space significantly.

Anytime property. The anytime property of our algorithm is also consequence of storing the smallest formula seen so far ((Line 7 and 14)). Once we find a separating formula, we can output it and continue the search for smaller separating formulas.

Extension to the noisy setting. The algorithm is seamlessly extended to the noisy setting by rewriting lines 6 and 13: instead of outputting only separating formulas, we output ε-separating formulas.

4 Searching for directed formulas

The first insight of our approach is the definition of a fragment of LTL that we call *directed LTL.*

A *partial symbol* is a conjunction of positive or negative atomic propositions. We write $s = p_0 \wedge p_2 \wedge \neg p_1$ for the partial symbol specifying that p_0 and p_2 hold and p_1 does not. The definition of a symbol satisfying a partial symbol is natural: for instance the symbol $\{p_0, p_2, p_4\}$ satisfies s. The *width* of a partial symbol is the number of atomic propositions it uses.

Directed LTL is defined by the following grammar:

$$\varphi = \mathbf{X}^n\, s \quad | \quad \mathbf{F}\,\mathbf{X}^n\, s \quad | \quad \mathbf{X}^n(s \wedge \varphi) \quad | \quad \mathbf{F}\,\mathbf{X}^n(s \wedge \varphi),$$

where s is a partial symbol and $n \in \{0, 1, \cdots\}$. As an example, the directed formula

$$\mathbf{F}((p \wedge q) \wedge \mathbf{F}\,\mathbf{X}^2\, \neg p)$$

reads: there exists a position satisfying $p \wedge q$, and at least two positions later, there exists a position satisfying $\neg p$. The intuition behind the term "directed" is that a directed formula fixes the order in which the partial symbols occur. A non-directed formula is $\mathbf{F}\, p \wedge \mathbf{F}\, q$: there is no order between p and q. Note that Directed LTL only uses the **X** and **F** operators as well as conjunctions and atomic propositions.

Generating directed formulas. Let us consider the following problem: given the sample $S = P \cup N$, we want to generate all directed formulas together with a list of traces in S, they satisfy. Our first technical contribution and key to the scalability of our approach is an efficient solution to this problem based on dynamic programming.

Let us define a natural order in which we want to generate directed formulas. They have two parameters: *length*, which is the number of partial symbols in the directed formula, and *width*, which is the maximum of the widths of the partial symbols in the directed formula. We consider the order based on summing these two parameters:

$$(1,1), (2,1), (1,2), (3,1), (2,2), (1,3), \ldots$$

(We note that in practice, slightly more complicated orders on pairs are useful since we want to increase the length more often than the width.) Our enumeration algorithm works by generating all directed formulas of a given pair of parameters in a recursive fashion. Assuming that we already generated all directed formulas for the pair of parameters (ℓ, w), we define two procedures, one for generating the directed formulas for the parameters $(\ell + 1, w)$, and the other one for $(\ell, w + 1)$.

When we generate the directed formulas, we also keep track of which traces in the sample they satisfy by exploiting a dynamic programming table called LastPos. We define it is as follows, where φ is a directed formula and t a trace in S:

$$\textsc{LastPos}(\varphi, t) = \{i \in [1, |t|] : t[1, i] \models \varphi\}\,.$$

The main benefit of LastPos is that it meshes well with directed formulas: it is algorithmically easy to compute them recursively on the structure of directed formulas.

A useful idea is to change the representation of the set of traces S, by precomputing the lookup table Index defined as follows, where t is a trace in S, s a partial symbol, and i in $[1, |t|]$:

$$\textsc{Index}(t, s, i) = \{j \in [i + 1, |t|] : t[j] \models s\}\,.$$

The table Index can be precomputed in linear time from S, and makes the dynamic programming algorithm easier to formulate.

Having defined the important ingredients, we now present the pseudocode 2 for both increasing the length and width of a formula. For the length increase algorithm, we define two extension operators $\wedge_{=k}$ and $\wedge_{\geq k}$ that "extend" the length of a directed formula φ by including a partial symbol s in the formula. Precisely, the operator $s \wedge_{=k} \varphi$ replaces the rightmost partial symbol s' in φ with $(s' \wedge \mathbf{X}^k\, s)$, while $s \wedge_{\geq k} \varphi$ replaces s' with $(s' \wedge \mathbf{F}\,\mathbf{X}^k\, s)$. For instance, $c \wedge_{=2} \mathbf{X}(a \wedge \mathbf{X}\, b) = \mathbf{X}(a \wedge \mathbf{X}(b \wedge \mathbf{X}^2\, c))$. For the width increase algorithm, we say that two directed formulas are *compatible* if they are equal except for partial symbols. For two compatible formulas, we define a *pointwise-and* ($\dot{\wedge}$) operator that takes the conjunction of the corresponding partial symbols at the same positions. For instance, $\mathbf{X}(a \wedge \mathbf{X}\, b) \mathbin{\dot{\wedge}} \mathbf{X}(b \wedge \mathbf{X}\, c) = \mathbf{X}((a \wedge b) \wedge \mathbf{X}(b \wedge c))$. The actual implementation of the algorithm refines the algorithms in certain places. For instance:

- Line 3: instead of considering all partial symbols, we restrict to those appearing in at least one positive trace.
- Line 13: some computations for $\varphi_{\geq j}$ can be made redundant; a finer data structure factorises the computations.
- Line 25: using a refined data structure, we only enumerate compatible directed formulas.

Lemma 1. *Algorithm 2 generates all directed formulas and correctly computes the tables* LastPos.

The dual point of view. We use the same algorithm to produce formulas in a dual fragment to directed LTL, which uses the $\mathbf{X}$ and $\mathbf{G}$ operators, the **last** predicate, as well as disjunctions and atomic propositions. The only difference is that we swap positive and negative traces in the sample. We obtain a directed formula from such a sample and apply its negation as shown below:

$$\neg\,\mathbf{X}\,\varphi = \mathbf{last} \vee \mathbf{X}\,\neg\varphi \qquad ; \qquad \neg\,\mathbf{F}\,\varphi = \mathbf{G}\,\neg\varphi \qquad ; \qquad \neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2.$$

5 Boolean combinations of formulas

As explained in the previous section, we can efficiently generate directed formulas and dual directed formulas. Now we explain how to form a Boolean combination of these formulas in order to construct separating formulas, as illustrated in the introduction.

Boolean combination of formulas. Let us consider the following subproblem: given a set of formulas, does there exist a Boolean combination of some of the formulas that is a separating formula? We call this problem the Boolean subset

Algorithm 2 Generation of directed formulas for the set of traces S

```
procedure Search directed formulas – length increase($\ell, w$)
    for all directed formulas $\varphi$ of length $\ell$ and width $w$ do
        for all partial symbols $s$ of width at most $w$ do
            for all $t \in S$ do
                $I = \textsc{LastPos}(\varphi, t)$
                for all $i \in I$ do
                    $J = \textsc{Index}(t, s, i)$
                    for all $j \in J$ do
                        $\varphi_{=j} \leftarrow s \wedge_{=(j-i)} \varphi$
                        add $j$ to $\textsc{LastPos}(\varphi_{=j}, t)$
                    end for
                    for all $j' \leq \max(J)$ do
                        $\varphi_{\geq j'} \leftarrow s \wedge_{\geq(j-i)} \varphi$
                        add $J \cap [j', |t|]$ to $\textsc{LastPos}(\varphi_{\geq j'}, t)$
                    end for
                end for
            end for
        end for
    end for
end procedure

procedure Search directed formulas – width increase($\ell, w$)
    for all directed formulas $\varphi$ of length $\ell$ and width $w$ do
        for all directed formulas $\varphi'$ of length $\ell$ and width 1 do
            if $\varphi$ and $\varphi'$ are compatible then
                $\varphi'' \leftarrow \varphi \mathbin{\dot\wedge} \varphi'$
                for all $t \in S$ do
                    $\textsc{LastPos}(\varphi'', t) \leftarrow \textsc{LastPos}(\varphi, t) \cap \textsc{LastPos}(\varphi', t)$
                end for
            end if
        end for
    end for
end procedure
```

cover, which is illustrated in Figure 1. In this example we have three formulas φ_1, φ_2, and φ_3, each satisfying subsets of $u_1, u_2, u_3, v_1, v_2, v_3$ as represented in the drawing. Inspecting the three subsets reveals that $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula.

The Boolean subset cover problem is a generalization of the well known and extensively studied subset cover problem, where we are given $S_1, \ldots, S_m$ subsets of $[1, n]$, and the goal is to find a subset I of $[1, m]$ such that $\bigcup_{i \in I} S_i$ covers all of $[1, n]$ – such a set I is called a cover. Indeed, it corresponds to the case where all formulas satisfy none of the negative traces: in that case, conjunctions are not useful, and we can ignore the negative traces. The subset cover problem is known to be NP-complete. However, there exists a polynomial-time $\log(n)$-approximation algorithm called the greedy algorithm: it is guaranteed to

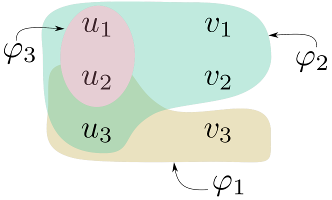


Fig. 1: The Boolean subset cover problem: the formulas φ_1, φ_2, and φ_3 satisfy the words encircled in the corresponding area; in this instance $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula.

construct a cover that is at most $\log(n)$ times larger than the minimum cover. This approximation ratio is optimal in the following sense [7]: there is no polynomial time $(1 - o(1)) \log(n)$-approximation algorithm for subset cover unless P = NP. Informally, the greedy algorithm for the subset cover problem does the following: it iteratively constructs a cover I by sequentially adding the most 'promising subset' to I, which is the subset S_i maximising how many more elements of $[1, n]$ are covered by adding i to I.

We introduce an extension of the greedy algorithm to the Boolean subset cover problem. The first ingredient is a scoring function, which takes into account both how close the formula is to being separating, and how large it is. We use the following score:

$$\mathit{Score}(\varphi) = \frac{\mathrm{Card}(\{t \in P : t \models \varphi\}) + \mathrm{Card}(\{t \in N : t \not\models \varphi\})}{\sqrt{|\varphi|} + 1},$$

where $|\varphi|$ is the size of φ. The use of $\sqrt{\cdot}$ is empirical, it is used to mitigate the importance of size over being separating.

The algorithm maintains a set of formulas B, which is initially the set of formulas given as input, and add new formulas to B until finding a separating formula. Let us fix a constant K, which in the implementation is set to 5. At each point in time, the algorithm chooses the K formulas $\varphi_1, \dots, \varphi_K$ with the highest score in B and constructs all disjunctions and conjunctions of φ_i with formulas in B. For each i, we keep the disjunction or conjunction with a maximal score, and add this formula to B if it has higher score than φ_i. We repeat this procedure until we find a separating formula or no formula is added to B.

Another natural approach to the Boolean subset cover problem is to use decision trees: we use one variable for each trace and one atomic proposition for each formula to denote whether the trace satisfies the formula. We then construct a decision tree classifying all traces. We experimented with both approaches and found that the greedy algorithm is both faster and yields smaller formulas. We do not report on these experiments because the formulas output using the decision tree approach are prohibitively larger and therefore not useful for explanations. Let us, however, remark that using decision trees we get a theoretical guarantee that if there exists a separating formula as a Boolean combination of the formulas, then the algorithm will find it.

6 Theoretical guarantees

The following result shows the relevance of our approach using directed LTL and Boolean combinations.

Theorem 1. *Every formula of LTL*$(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ *is equivalent to a Boolean combination of directed formulas. Equivalently, every formula of LTL*$(\mathbf{G}, \mathbf{X}, \wedge, \vee)$ *is equivalent to a Boolean combination of dual directed formulas.*

The proof of Theorem 1 can be found in the extended version of the paper [21]. To get an intuition, let us consider the formula $\mathbf{F}\,p \wedge \mathbf{F}\,q$, which is not directed, but equivalent to $\mathbf{F}(p \wedge \mathbf{F}\,q) \vee \mathbf{F}(q \wedge \mathbf{F}\,p)$. In the second formulation, there is a disjunction over the possible orderings of p and q. The formal proof generalises this rewriting idea.

This implies the following properties for our algorithm:

- *terminating*: given a bound on the size of formulas, the algorithm eventually generates all formulas of bounded size,
- *correctness*: if the algorithm outputs a formula, then it is separating,
- *completeness*: if there exists a separating formula in LTL$(\mathbf{F}, \mathbf{G}, \mathbf{X}, \wedge, \vee)$ with no nesting of $\mathbf{F}$ and $\mathbf{G}$, then the algorithm finds a separating formula.

7 Experimental evaluation

In this section, we answer the following research questions to assess the performance of our LTL learning algorithm.

RQ1: How effective are we in learning concise LTL formulas from samples?
RQ2: How much scalability do we achieve through our algorithm?
RQ3: What do we gain from the anytime property of our algorithm?

Experimental Setup. To answer the questions above, we have implemented a prototype of our algorithm in Python 3 in a tool named `SCARLET`[6] (SCalable Anytime algoRithm for LEarning lTl). We run `SCARLET` on several benchmarks generated synthetically from LTL formulas used in practice. To answer each research question precisely, we choose different sets of LTL formulas. We discuss them in detail in the corresponding sections. Note that, however, we did not consider any formulas with $\mathbf{U}$-operator, since `SCARLET` is not designed to find such formulas.

To assess the performance of `SCARLET`, we compare it against two state-of-the-art tools for learning logic formulas from examples:

[6] https://github.com/rajarshi008/Scarlet

1. FLIE[7], developed by [19], infers minimal LTL formulas using a learning algorithm that is based on constraint solving (SAT solving).
2. SYSLITE[8], developed by [1], originally infers minimal past-time LTL formulas using an enumerative algorithm implemented in a tool called CVC4SY [23]. For our comparisons, we use a version of SYSLITE that we modified (which we refer to as SYSLITE$_\mathrm{L}$) to infer LTL formulas rather than past-time LTL formulas. Our modifications include changes to the syntactic constraints generated by SYSLITE$_\mathrm{L}$ as well as changing the semantics from past-time LTL to ordinary LTL.

To obtain a fair comparison against SCARLET, in both the tools, we disabled the **U**-operator. This is because if we allow **U**-operator this will only make the tools slower since they will have to search through all formulas containing **U**.

All the experiments are conducted on a single core of a Debian machine with Intel Xeon E7-8857 CPU (at 3 GHz) using up to 6 GB of RAM. We set the timeout to be 900 s for all experiments. We include scripts to reproduce all experimental results in a publicly available artifact [22].

Table 1: Common LTL formulas used in practice

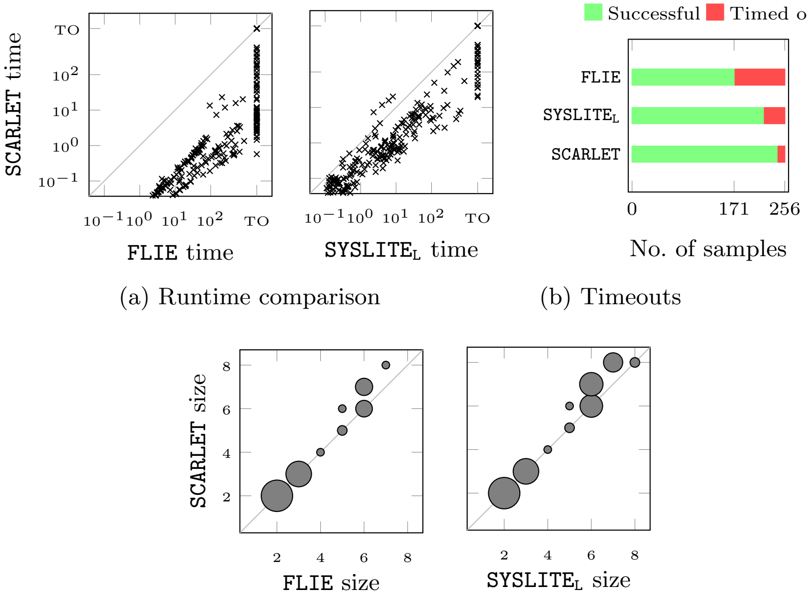
Absence:	$\mathbf{G}(\neg p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(\neg p))$
Existence:	$\mathbf{F}(p)$, $\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q))$
Universality:	$\mathbf{G}(p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(p))$
Disjunction of patterns:	$\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q)$ $\vee\, \mathbf{G}(\neg s) \vee \mathbf{F}(r \wedge \mathbf{F}(s))$, $\mathbf{F}(r) \vee \mathbf{F}(p) \vee \mathbf{F}(q)$

Sample generation. To provide a comparison among the learning tools, we follow the literature [19,24] and use synthetic benchmarks generated from real-world LTL formulas. For benchmark generation, earlier works rely on a fairly naive generation method. In this method, starting from a formula φ, a sample is generated by randomly drawing traces and categorizing them into positive and negative examples depending on the satisfaction with respect to φ. This method, however, often results in samples that can be separated by formulas much smaller than φ. Moreover, it often requires a prohibitively large amount of time to generate samples (for instance, for $\mathbf{G}\, p$, where almost all traces satisfy a formula) and, hence, often does not terminate in a reasonable time.

To alleviate the issues in the existing method, we have designed a novel generation method for the quick generation of large samples. In our method, we first convert the starting formula into an equivalent DFA and then extract accepted and rejected words to obtain a sample of the desired size. We provide more details on this new generation method used in the extended version [21].

[7] https://github.com/ivan-gavran/samples2LTL

[8] https://github.com/CLC-UIowa/SySLite



(a) Runtime comparison

(b) Timeouts

(c) Size comparison

Fig. 2: Comparison of SCARLET, FLIE and SYSLITE$_\text{L}$ on synthetic benchmarks. In Figure 2a, all times are in seconds and 'TO' indicates timeouts. The size of bubbles in the figure indicate the number of samples for each datapoint.

7.1 RQ1: Performance Comparison

To address our first research question, we have compared all three tools on a synthetic benchmark suite generated from eight LTL formulas. These formulas originate from a study by Dwyer et al. [8], who have collected a comprehensive set of LTL formulas arising in real-world applications (see Table 1 for an excerpt). The selected LTL formulas have, in fact, also been used by FLIE for generating its benchmarks. While FLIE also considered formulas with **U**-operator, we did not consider them for generating our benchmarks due to reasons mentioned in the experimental setup.

Our benchmark suite consists of a total of 256 samples (32 for each of the eight LTL formulas) generated using our generation method. The number of traces in the samples ranges from 50 to 2 000, while the length of traces ranges from 8 to 15.

Figure 2a presents the runtime comparison of FLIE, SYSLITE$_\text{L}$ and SCARLET on all 256 samples. From the scatter plots, we observe that SCARLET ran faster than FLIE on all samples. Likewise, SCARLET was faster than SYSLITE$_\text{L}$ on all but eight (out of 256) samples. SCARLET timed out on only 13 samples, while FLIE and SYSLITE$_\text{L}$ timed out on 85 and 36, respectively (see Figure 2b).

The good performance of SCARLET can be attributed to its efficient formula search technique. In particular, SCARLET only considers formulas that have a high potential of being a separating formula since it extracts Directed LTL formulas from the sample itself. `FLIE` and `SYSLITE`$_\mathrm{L}$, on the other hand, search through arbitrary formulas (in order of increasing size), each time checking if the current one separates the sample.

Figure 2c presents the comparison of the size of the formulas inferred by each tool. On 170 out of the 256 samples, all tools terminated and returned an LTL formula with size at most 7. In 150 out of this 170 samples, `SCARLET`, `FLIE`, and `SYSLITE`$_\mathrm{L}$ inferred formulas of equal size, while on the remaining 20 samples `SCARLET` inferred formulas that were larger. The latter observation indicates that `SCARLET` misses certain small, separating formulas, in particular, the ones which are not a Boolean combination of directed formulas.

However, it is important to highlight that the formulas learned by `SCARLET` are in most cases not significantly larger than those learned by `FLIE` and `SYSLITE`$_\mathrm{L}$. This can be seen from the fact that the average size of formulas inferred by `SCARLET` (on benchmarks in which none of the tools timed out) is 3.21, while the average size of formulas inferred by `FLIE` and `SYSLITE`$_\mathrm{L}$ is 3.07.

Overall, `SCARLET` displayed significant speed-up over both `FLIE` and `SYSLITE`$_\mathrm{L}$ while learning a formula similar in size, answering question RQ1 in the positive.

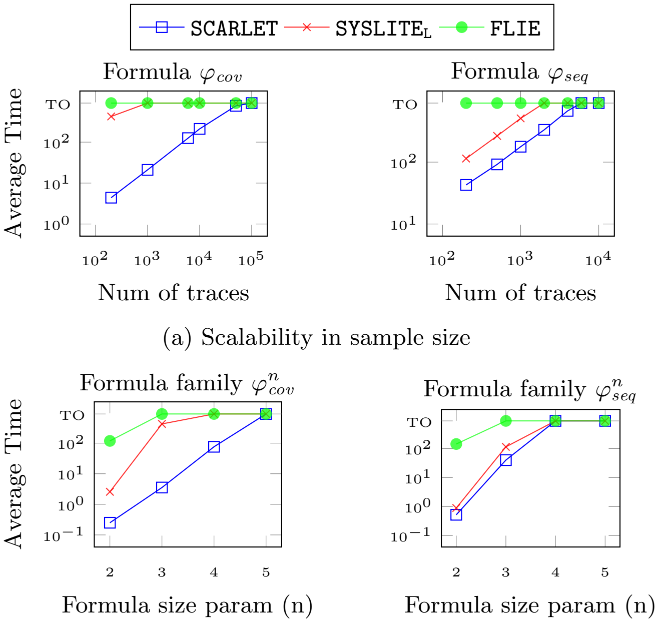
7.2 RQ2: Scalability

To address the second research question, we investigate the scalability of `SCARLET` in two dimensions: the size of the sample and the size of the formula from which the samples are generated.

Scalability with respect to the size of the samples. For demonstrating the scalability with respect to the size of the samples, we consider two formulas $\varphi_{cov} = \mathbf{F}(a_1) \wedge \mathbf{F}(a_2) \wedge \mathbf{F}(a_3)$ and $\varphi_{seq} = \mathbf{F}(a_1 \wedge \mathbf{F}(a_2 \wedge \mathbf{F}\, a_3))$, both of which appear commonly in robotic motion planning [10]. While the formula φ_{cov} describes the property that a robot eventually visits (or covers) three regions a_1, a_2, and a_3 in arbitrary order, the formula φ_{seq} describes that the robot has to visit the regions in the specific order $a_1 a_2 a_3$.

We have generated two sets of benchmarks for both formulas for which we varied the number of traces and their length, respectively. More precisely, the first benchmark set contains 90 samples of an increasing number of traces (5 samples for each number), ranging from 200 to 100 000, each consisting of traces of fixed length 10. On the other hand, the second benchmark set contains 90 samples of 200 traces, containing traces from length 10 to length 50. As the results on both benchmark sets are similar, we here discuss the results on the first set and refer the readers to the extended version [21] for the second set.

Figure 3a shows the average runtime results of `SCARLET`, `FLIE`, and `SYSLITE`$_\mathrm{L}$ on the first benchmark set. We observe that `SCARLET` substantially outperformed the other two tools on all samples. This is because both φ_{cov} and φ_{seq} are of size eight and inferring formulas of such size is computationally challenging for `FLIE` and `SYSLITE`$_\mathrm{L}$. In particular, `FLIE` and `SYSLITE`$_\mathrm{L}$ need to search through



(a) Scalability in sample size

(b) Scalability in formula size

Fig. 3: Comparison of `SCARLET`, `FLIE` and `SYSLITE`$_L$ on synthetic benchmarks. In Figure 3a, all times are in seconds and 'TO' indicates timeouts.

all formulas of size upto eight to infer the formulas, while, `SCARLET`, due to its efficient search order (using length and width of a formula), infers them faster.

From Figure 3a, we further observe a significant difference between the run times of `SCARLET` on samples generated from formula φ_{cov} and from formula φ_{seq}. This is evident from the fact that `SCARLET` failed to infer formulas for samples of φ_{seq} starting at a size of 6 000, while it could infer formulas for samples of φ_{cov} up to a size of 50 000. Such a result is again due to the search order used by `SCARLET`: while φ_{cov} is a Boolean combination of directed formulas of length 1 and width 1, φ_{seq} is a directed formula of length 3 and width 1.

Scalability with respect to the size of the formula. To demonstrate the scalability with respect to the size of the formula used to generate samples, we have extended φ_{cov} and φ_{seq} to families of formulas $(\varphi^n_{cov})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{cov} = \mathbf{F}(a_1)\wedge\mathbf{F}(a_2)\wedge\ldots\wedge\mathbf{F}(a_n)$ and $(\varphi^n_{seq})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{seq} = \mathbf{F}(a_1\wedge\mathbf{F}(a_2\wedge\mathbf{F}(\ldots\wedge\mathbf{F}\,a_n)))$, respectively. These family of formulas describe properties similar to that of φ_{cov} and φ_{seq}, but the number of regions is parameterized by $n\in\mathbb{N}\setminus\{0\}$. We consider formulas from the two families by varying n from 2 to 5 to generate a benchmark suite consisting of samples (5 samples for each formula) having 200 traces of length 10.

Figure 3b shows the average run time comparison of the tools for samples from increasing formula sizes. We observe a trend similar to Figure 3a: `SCARLET` performs better than the other two tools and infers formulas of family φ^n_{cov} faster than that of φ^n_{seq}. However, contrary to the near linear increase of the runtime with the number of traces, we notice an almost exponential increase of the runtime with the formula size.

Overall, our experiments show better scalability with respect to sample and formula size compared against the other tools, answering RQ2 in the positive.

7.3 RQ3: Anytime Property

To answer RQ3, we list two advantages of the anytime property of our algorithm. We demonstrate these advantages by showing evidence from the runs of SCARLET on benchmarks used in RQ1 and RQ2.

First, in the instance of a time out, our algorithm may find a "concise" separating formula while the other tools will not. In our experiments, we observed that for all benchmarks used in RQ1 and RQ2, SCARLET obtained a formula even when it timed out. In fact, in the samples from φ^5_{cov} used in RQ2, SCARLET (see Figure 3b) obtained the exact original formula, that too within one second (0.7 seconds in average), although timed out later. The time out was because SCARLET continued to search for smaller formulas even after finding the original formula.

Second, our algorithm can actually output the final formula earlier than its termination. This is evident from the fact that, for the 243 samples in RQ1 where SCARLET does not time out, the average time required to find the final formula is 10.8 seconds, while the average termination time is 25.17 seconds. Thus, there is a chance that even if one stops the algorithm earlier than its termination, one can still obtain the final formula.

Our observations from the experiments clearly indicate the advantages of anytime property to obtain a concise separating formula and thus, answering RQ3 in the positive.

8 Conclusion

We have proposed a new approach for learning temporal properties from examples, fleshing it out in an approximation anytime algorithm. We have shown in experiments that our algorithm outperforms existing tools in two ways: it scales to larger formulas and input samples, and even when it timeouts it often outputs a separating formula.

Our algorithm targets a strict fragment of LTL, restricting its expressivity in two aspects: it does not include the $\mathbf{U}$ ("until") operator, and we cannot nest the eventually and globally operators. We leave for future work to extend our algorithm to full LTL.

An important open question concerns the theoretical guarantees offered by the greedy algorithm for the Boolean subset cover problem. It extends a well known algorithm for the classic subset cover problem and this restriction has been proved to yield an optimal $\log(n)$-approximation. Do we have similar guarantees in our more general setting?

References

1. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: Formal Methods in Computer Aided Design, FMCAD (2020)
2. Baresi, L., Kallehbasti, M.M.P., Rossi, M.: Efficient scalable verification of LTL specifications. In: ICSE (1). pp. 711–721. IEEE Computer Society (2015)
3. Bombara, G., Vasile, C.I., Penedo Alvarez, F., Yasuoka, H., Belta, C.: A Decision Tree Approach to Data Classification using Signal Temporal Logic. In: Hybrid Systems: Computation and Control, HSCC (2016). https://doi.org/10.1145/2883817.2883843
4. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. International Conference on Automated Planning and Scheduling, ICAPS (2019), https://ojs.aaai.org/index.php/ICAPS/article/view/3529
5. Chou, G., Ozay, N., Berenson, D.: Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations. In: Robotics: Science and Systems (2020). https://doi.org/10.15607/RSS.2020.XVI.097
6. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. p. 854–860. IJCAI '13, AAAI Press (2013)
7. Dinur, I., Steurer, D.: Analytical approach to parallel repetition. In: Symposium on Theory of Computing, STOC. pp. 624–633 (2014). https://doi.org/10.1145/2591796.2591884
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering, ICSE (1999). https://doi.org/10.1145/302405.302672
9. Ehlers, R., Gavran, I., Neider, D.: Learning properties in LTL $\cap$ ACTL from positive examples only. In: Formal Methods in Computer Aided Design, FMCAD. pp. 104–112 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_17
10. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for mobile robots. In: International Conference on Robotics and Automation, ICRA (2005). https://doi.org/10.1109/ROBOT.2005.1570410
11. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: International Conference on Grammatical Inference, ICGI (2021), https://proceedings.mlr.press/v153/fijalkow21a.html
12. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
13. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: automatic requirements specification extraction from natural language. In: NASA Formal Methods, NFM (2016). https://doi.org/10.1007/978-3-319-40648-0_4
14. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ (2020), http://ceur-ws.org/Vol-2584/PT-paper4.pdf
15. Grover, K., Barbosa, F.S., Tumova, J., Kretínský, J.: Semantic abstraction-guided motion planning for scltl missions in unknown environments. In: Robotics: Science and Systems XVII (2021). https://doi.org/10.15607/RSS.2021.XVII.090
16. Kim, J., Muise, C., Shah, A., Agarwal, S., Shah, J.: Bayesian inference of linear temporal logic specifications for contrastive explanations. In: International Joint Conference on Artificial Intelligence, IJCAI (2019). https://doi.org/10.24963/ijcai.2019/776

17. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: International Conference on Automated Software Engineering, ASE (2015). https://doi.org/10.1109/ASE.2015.71
18. Li, W.: Specification Mining: New Formalisms, Algorithms and Applications. Ph.D. thesis, University of California, Berkeley, USA (2013), http://www.escholarship.org/uc/item/4027r49r
19. Neider, D., Gavran, I.: Learning linear temporal properties. In: Formal Methods in Computer Aided Design, FMCAD (2018). https://doi.org/10.23919/FMCAD.2018.8603016
20. Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science, SFCS (1977). https://doi.org/10.1109/SFCS.1977.32
21. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning formulas in linear temporal logic. CoRR **abs/2110.06726** (2021), https://arxiv.org/abs/2110.06726
22. Raha, R., Roy, R., Fijalkow, N., Neider, D.: SCARLET: Scalable Anytime Algorithm for Learning LTL (Jan 2022). https://doi.org/10.5281/zenodo.5890149, https://doi.org/10.5281/zenodo.5890149
23. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Computer-Aided Verification, CAV (2019). https://doi.org/10.1007/978-3-030-25543-5_5
24. Roy, R., Fisman, D., Neider, D.: Learning interpretable models in the property specification language. In: International Joint Conference on Artificial Intelligence, IJCAI. pp. 2213–2219 (2020). https://doi.org/10.24963/ijcai.2020/306
25. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: Verified Software. Theories, Tools, and Experiments, VSTTE (2016). https://doi.org/10.1007/978-3-319-48869-1_2