# $\mathbf{LTL}_f$ Learning meets Boolean Synthesis

**Anonymous submission**


## Abstract

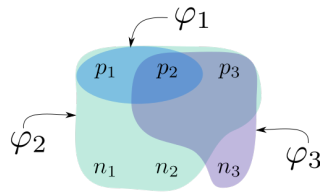
Learning formulas in Linear Temporal Logic (LTL) from traces is a very fundamental research problem which has found applications in artificial intelligence, software engineering, programming languages, formal methods, control of cyber-physical systems, and robotics. We implement a new tool called BOLT improving over the state of the art by solving more than 100x faster over 75% of the benchmarks with smaller or equal formulas in 95% of the cases. Our key insight is to leverage Boolean Synthesis as a subroutine to combine existing formulas using boolean connectives. Importantly, our approach is correct and complete by design, and the Boolean Synthesis component offers a novel trade-off between efficiency and formula size.


## 1 Introduction

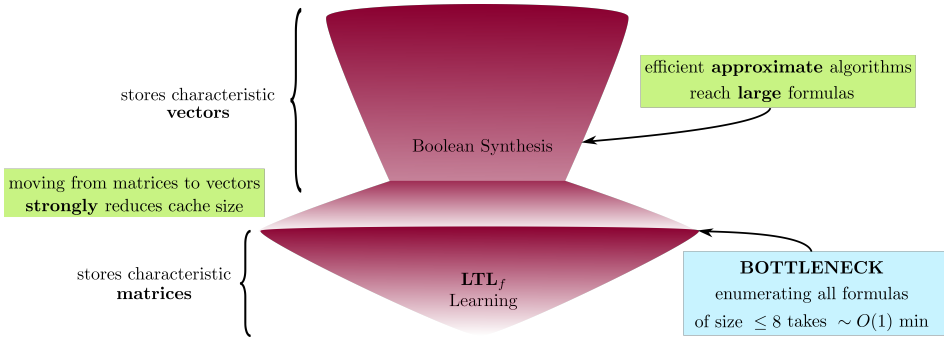
Linear Temporal Logic (**LTL**) (Pnueli 1977) is a prominent logic for specifying temporal properties over traces; in this paper, we consider **LTL** on finite traces (Giacomo and Vardi 2013), abbreviated $\mathbf{LTL}_f$. The fundamental problem we study is to learn $\mathbf{LTL}_f$ formulas from traces: given a set of positive and negative traces, find an $\mathbf{LTL}_f$ formula separating positive from negative ones. $\mathbf{LTL}_f$ learning spans different research communities, each contributing applications, approaches, and viewpoints on this problem. We give below a cursory cross-sectional survey of motivations and applications of $\mathbf{LTL}_f$ learning in these three communities.

**Software Engineering, Programming Languages, and Formal Methods.** $\mathbf{LTL}_f$ learning is an instantiation of specification mining, which is an active area of research devoted to discovering formal specifications of code. At this point it is important to distinguish between the dynamic and the static setting: we are in this paper interested in the *dynamic* setting where we observe program executions, also called traces, to infer properties of the code. The term *specification mining* was coined by Ammons, Bodík, and Larus (2002) in a seminal paper where finite state machines capture both temporal and data dependencies. Zeller (2010) contributed a roadmap to mining specification in 2010, highlighting its potential for software engineering. A few years later Rozier (2016) shaped an entire research programme revolving around ***LTL***$_f$ *Genesis*, motivating and introducing the $\mathbf{LTL}_f$ learning problem as we study it here. The first tool supporting all $\mathbf{LTL}_f$ formulas is called Texada, and was created in 2015 by Lemieux, Park, and Beschastnikh (2015); Lemieux and Beschastnikh (2015). Following Texada, a line of work focuses on scaling $\mathbf{LTL}_f$ learning to industrial sizes, conjuring different approaches. We mention here the state-of-the-art $\mathbf{LTL}_f$ learning tools: Scarlet, based on combinatorial search (Raha et al. 2022), and the GPU-accelerated VFB algorithm (Valizadeh, Fijalkow, and Berger 2024). Applications of mining specifications for software include detecting malicious behaviours (Christodorescu, Jha, and Kruegel 2007) or violations (Li and Zhou 2005) and is already widely adopted in software engineering: for instance, the ARSENAL and ARSENAL2 projects (Ghosh et al. 2016) have been successful in constructing $\mathbf{LTL}_f$ formulas from traces inferred from English requirements, and the FRET project generates $\mathbf{LTL}_f$ from trace descriptions (Giannakopoulou et al. 2020). We refer to the textbook (Lo et al. 2017) and the PhD thesis of Li (Li 2013) for comprehensive presentation of specification mining for software, and its relationship to data mining.

**Control of Cyber-Physical Systems and Robotics.** $\mathbf{LTL}_f$ learning is extensively studied in a second area of research with different goals and very different methods: to capture properties of trajectories in models and systems. Given its quantitative nature, Signal Temporal Logic (**STL**) is preferred over $\mathbf{LTL}_f$ since it involves numerical constants and can thus capture real-valued and time-varying behaviours. In particular, temporal logics allow reasoning about robustness (Bartocci et al. 2015) and anomaly detection (Jones, Kong, and Belta 2014; Kong, Jones, and Belta 2017). There is a vast body of work on learning temporal logics in control and robotics, which can be divided into two: techniques aiming at fitting parameters of a fixed assumed **STL** formula (Asarin et al. 2012; Yang, Hoxha, and Fainekos 2012; Jin et al. 2015), and approaches searching for both formulas and parameters (Jin et al. 2015; Bombara et al. 2016). This led to many case studies: automobile transmission controller and engine airpath control (Yang, Hoxha, and Fainekos 2012), assisted ventilation in intensive care patients (Bufo et al. 2014), dynamics of a biological circadian oscillator and discriminating different types of cardiac malfunction from electro-cardiogram data (Bartocci, Bortolussi, and Sanguinetti 2014), anomaly detection in a mar-




(1) Instance of Boolean Synthesis




(2) Overview of **LTL**$_f$ Learning with Boolean Synthesis

itime environment (Bombara et al. 2016), demonstrations in robotics (Chou, Ozay, and Berenson 2020), and detection of attention loss in pilots (Lyu et al. 2024).

**Artificial Intelligence.** The field of AI has contributed to **LTL**$_f$ learning a number of applications, but also a wealth of techniques. The overarching philosophy of **LTL**$_f$ learning in AI is that **LTL**$_f$ forms a natural and explainable formalism (Camacho and McIlraith 2019) for specifying objectives (Icarte et al. 2018a) in various machine learning contexts. For instance, in the context of reinforcement learning, **LTL**$_f$ formulas have been used to either guide (Camacho et al. 2017; Brafman, Giacomo, and Patrizi 2018) or constrain policies (Icarte et al. 2018b; Camacho et al. 2019). Another recent application aims at speeding up synthesis by separating data and control (Murphy et al. 2024).

**SOTA on LTL$_f$ Learning.** Since learning temporal logics is computationally hard[1], different approaches have been explored. The state-of-the-art tool is Scarlet, which is based on specifically tailored fragments of **LTL**$_f$ (Raha et al. 2022, 2024a). A GPU-accelerated algorithm was recently published (Valizadeh, Fijalkow, and Berger 2024), yielding an improved state of the art in a different category. Scarlet cannot reliably learn formulas of size greater than 15, which is less than ideal: we aim to change this.

**Boolean Synthesis.** Our key insight in this work, and we believe the key to scaling **LTL**$_f$ learning to industrial sizes, is to treat boolean operators differently from temporal operators. To address the former, we introduce a new and natural problem, which we call *Boolean Synthesis*:

Let us consider a finite set $X$ partitioned into positive and negative elements: $X = P \cup N$. We are given a (potentially large) set of atomic formulas $\varphi_1, \ldots, \varphi_k$ over $X$. The goal is to construct a formula separating $P$ from $N$ using conjunctions and disjunctions of the atomic formulas. We refer to Figure (1) for an illustration: here, $P = \{p_1, p_2, p_3\}$ and $N = \{n_1, n_2, n_3\}$. The formulas $\varphi_1, \varphi_2$, and $\varphi_3$ satisfy the points encircled in the corresponding area. In this instance $\varphi_1 \vee (\varphi_2 \wedge \varphi_3)$ is a (minimal) solution.

The **key finding** of this work is that the Boolean Synthesis problem can be solved *in an approximate way* very efficiently, which can then be leveraged to learn **LTL**$_f$ formulas of size way beyond the reach of existing tools. An overview of our general approach is depicted in Figure (2).

**Our contributions.**

- We propose a framework for combining **LTL**$_f$ learning with Boolean Synthesis and construct three algorithms in this framework.
- We implement our algorithms in a new tool called BOLT.
- We show through experiments that BOLT significantly improves over the state of the art both in terms of wall-clock time and size of learnt formulas.

We believe Boolean Synthesis is a fundamental problem of independent interest, which has a lot of potential applications beyond **LTL**$_f$ learning. We leave as future work to explore them; we discuss perspectives in Section 6.

## 2 LTL Learning

### 2.1 Problem Definition

Let us fix a (finite) set of *atomic propositions* $\mathsf{AP}$. We consider *traces*, which are words over alphabet $2^{\mathsf{AP}}$. We index traces from position 1 (not 0) and the letter at position $i$ in the trace $w$ is written $w(i)$, so $w = w(1) \ldots w(\ell)$ where $\ell$ is the length of $w$, written $|w| = \ell$. We write $w[k \ldots] = w(k) \ldots w(\ell)$. To avoid unnecessary technical complications, we only consider non-empty traces.

The syntax of *Linear Temporal Logic* (**LTL**$_f$) includes atomic propositions $c \in \mathsf{AP}$ and their negations, as well as $\top$ and $\bot$, the boolean operators $\wedge$ and $\vee$, and the *temporal operators* **X**, **F**, **G**, and **U**. Note that as usually done, we work with **LTL**$_f$ in *negation normal form*, meaning that negation is only used on atomic propositions. The semantic of **LTL**$_f$ over finite traces is defined inductively, through the notation $w \models \varphi$ where $w$ is a non-empty trace and $\varphi$ is an **LTL**$_f$ formula. The definition is given below for the atomic propositions and temporal operators **U**, **F**, **G**, and **X**, with boolean operators interpreted as usual. For $w$ of length $\ell$:

- $w \models \top$ and $w \not\models \bot$.
- $w \models c$ if $c \in w(1)$ and $w \models \neg c$ if $c \notin w(1)$.
- $w \models \mathbf{X}\,\varphi$ if $\ell > 1$ and $w[2 \ldots] \models \varphi$ (*next* operator).
- $w \models \varphi\,\mathbf{U}\,\psi$ if there is $i \in [1, \ell]$ such that for $j \in [1, i-1]$ we have $w[j \ldots] \models \varphi$, and $w[i \ldots] \models \psi$ (*until* operator).

[1] It is NP-complete, even for restricted fragments of **LTL**$_f$ (Fijalkow and Lagarde 2021; Mascle, Fijalkow, and Lagarde 2023).

We also consider $\mathbf{F}\,\varphi$ ("eventually $\varphi$"), which is syntactic sugar for $\top\,\mathbf{U}\,\varphi$, and $\mathbf{G}\,\varphi$ ("globally $\varphi$"), which is syntactic sugar for $\neg\,\mathbf{F}\,\neg\varphi$. We say that $w$ *satisfies* $\varphi$ if $w \models \varphi$.

**The $\mathbf{LTL}_f$ Learning Problem.** A *sample* is a pair $(P, N)$ where $P$ is a finite set of *positive* traces and $N$ a finite set of *negative* traces. The $\mathbf{LTL}_f$ learning problem is:

> **INPUT**: a sample $(P, N)$
> **OUTPUT**: a small $\mathbf{LTL}_f$ formula $\varphi$ such that all of $P$ satisfies $\varphi$ and none of $N$ satisfies $\varphi$

## 2.2 The VFB algorithm

Our first baseline for $\mathbf{LTL}_f$ learning is an optimised enumerative algorithm from (Valizadeh, Fijalkow, and Berger 2024); we call it the *VFB algorithm*. Simply put:

> VFB algorithm = combinatorial search + fast evaluation + observational equivalence

**Combinatorial Search.** The core algorithm is a bottom-up enumeration of $\mathbf{LTL}_f$ formulas by size. First, the set of formulas of size 1 contains all letters and their negations. At step $k$, we generate all formulas of size $k+1$. In order to do this, for each operator, we take arguments such that the sum of sizes of arguments is equal to $k$; this way, the generated formula will have size $k+1$ by combining it with the selected operator. The detailed pseudocode is provided in Appendix A, Algorithm 2. A property of this algorithm is that it is agnostic to the operators: it can be adapted to any fragment or be augmented with other operators, as long as we can easily compute inductively whether a trace satisfies a formula.

**Fast Evaluation.** The key idea is to represent the semantics of $\mathbf{LTL}_f$ formulas on input traces using so-called "characteristic matrices":

- The *characteristic sequence (CS)* of a formula $\varphi$ on a trace $w$ is the bit vector $v \in \{0,1\}^{|w|}$ such that $w[i\ldots] \models \varphi$ if and only if $v(i) = 1$;
- The *characteristic matrix (CM)* of a formula $\varphi$ over the positive traces $P$ and the negative traces $N$ is a sequence $m$ such that $m(i)$ is the CS of $\varphi$ on the $i^{\text{th}}$ trace of $P \cup N$.

The characteristic matrix is not necessarily a standard rectangular matrix if traces in $P \cup N$ do not have the same length. With these objects, one can compute inductively the characteristic sequences and matrices of formulas with only a few bitwise operations. For instance, the semantic of the next operator $\mathbf{X}$ is simply a "left shift" on characteristic sequences. This originates from (Valizadeh, Fijalkow, and Berger 2024) and is recalled in Appendix A.

**Observational Equivalence.** A second idea to tackle the combinatorial explosion of the number of formulas is to consider only "semantically unique" formulas. *Observational equivalence* is an idea from program synthesis (Gabbrielli, Levi, and Meo 1992); in the context of $\mathbf{LTL}_f$ learning, two formulas are observationally equivalent if they generate the same characteristic matrices. When enumerating formulas, if we keep a single representative for each equivalence class, we do not lose completeness.

**Scaling Issues.** Despite these improvements to a simple exhaustive search, the scaling of the problem is enormous. We identified two problems limiting the depth of algorithms.

The first is of course the exponential blow-up due to the number of increasingly large formulas. This phenomenon is exacerbated by the number of operators, each additional operator adding to this exponential blow-up.

The second is the memory used to store the evaluation of $\mathbf{LTL}_f$ formulas in the form of characteristic matrices, which is necessary to check for observational equivalence. The more input traces there are, the larger the characteristic matrices get; the longer the traces are, the wider these matrices get. When matrices have more elements, observational equivalence is less likely to reduce the blow-up.

# 3 Boolean Synthesis

Our idea to tackle these scaling issues is to use the VFB algorithm on the full $\mathbf{LTL}_f$ only up to a fixed formula size, chosen based on a memory or time threshold. Since the exponential blow-up does not allow to go much further beyond that size, we then restrict our focus to the fragment of $\mathbf{LTL}_f$ with only $\wedge$ and $\vee$ to move forward. This restricted problem is an instance of what we call *Boolean Synthesis*, a problem of independent interest that is solved as a subroutine in our $\mathbf{LTL}_f$ learning algorithms.

A high-level view of the procedure is in Algorithm 1. In this algorithm, VFB-BOUNDED is a variant of VFB which only computes $\mathbf{LTL}_f$ formulas up to some size $k$, then returns two objects: a solution or $\perp$ if it has not found a solution, and the generated formulas up to size $k$. If no solution is found, the generated formula are fed to a Boolean Synthesis solver called BS-SOLVER. From the point of view of the Boolean Synthesis solver, the $\mathbf{LTL}_f$ formulas are letters of a new alphabet with a specific weight.

**Algorithm 1** VFB algorithm with Boolean Synthesis solver

**Hyperparameter**: LTL2BS-switch (below, $k$), the maximal size of $\mathbf{LTL}_f$ formulas before switching to Boolean Synthesis.

```
1: procedure SEARCH(AP, O1, O2, P, N, CM, k)
2:     S, F ← VFB-BOUNDED(AP, O1, O2, CM, k)
3:     if S ≠ ⊥ then          ▷ If VFB algorithm found a solution
4:         return S
5:     else
6:         F' ← COLLAPSE(F)
7:         return BS-SOLVER(F', P, N)
```

When we send the generated formulas to a Boolean Synthesis solver, two kinds of collapse occur (which we represent in Algorithm 1 with function COLLAPSE).

First, once we restrict our focus to operators $\wedge$ and $\vee$, two formulas can be assumed to be observationally equivalent if and only if they have the same *first bit* for each row of their characteristic matrix. Indeed, as there is no more "temporal phenomenon", this is a sufficient information to determine whether a given trace satisfies a boolean combination of (already evaluated) $\mathbf{LTL}_f$ formulas. In other words,

we move from the computation of matrices to vectors (the vector of first bits), which produces many collisions and reduces the number of formulas to consider. We quantified this collapse in practice over the benchmarks used in Section 5 (only benchmarks that did not time out within 60 s have been used). Let $\mathcal{F}$ denote the set of observationally non-equivalent $\mathbf{LTL}_f$ formulas of size $\leq 8$, and $\mathcal{F}' = \text{COLLAPSE}(\mathcal{F})$ denote the set of formulas with a distinct first column for the characteristic matrix. On average over 780 instances, the ratio $|\mathcal{F}|/|\mathcal{F}'|$ is 3.54, with a minimum of 1.09, a maximum of 48.22, and a standard deviation of 5.46.

Second, storing vectors instead of matrices is a massive gain: space-wise, as storing the evaluation of a formula is a bit vector of size $|P| + |N|$ instead of $\ell \cdot (|P| + |N|)$, where $\ell$ is the total number of bits in an input trace, and time-wise, as new formulas can be evaluated faster. This enables to explore much larger formulas (in an incomplete way, as operators are missing) at a reduced space and time cost.

### 3.1 Definition of Boolean Synthesis

Let $P$ and $N$ be two disjoint finite sets. Let $\mathcal{F}$ be a family of sets with $F \subseteq P \cup N$ for all $F \in \mathcal{F}$. We assume that a *weight* $\mathsf{weight}(F) \geq 1$ is associated with each set $F \in \mathcal{F}$ (which is unrelated to the cardinality of $F$).

We define $\{\cup, \cap\}$-*boolean formulas* (or *boolean formulas* for brevity) and their weights inductively: $\emptyset$ is a formula of weight 0; $F \in \mathcal{F}$ is a formula of weight $\mathsf{weight}(F)$; if $\theta_1$ and $\theta_2$ are boolean formulas, then so are $\theta_1 \cup \theta_2$ and $\theta_1 \cap \theta_2$, both of weight $1+\mathsf{weight}(\theta_1)+\mathsf{weight}(\theta_2)$. The interpretation $[\![\theta]\!]$ of a boolean formula $\theta$ is simply the set it describes. We adjust observational equivalence: $\theta \sim \theta'$ if $[\![\theta]\!] = [\![\theta']\!]$.

The Boolean Synthesis problem is as follows:

> **INPUT**: two disjoint finite sets $P$, $N$ and a family $\mathcal{F}$ of weighted subsets of $P \cup N$
> **OUTPUT**: a $\{\cup, \cap\}$-boolean formula $\theta$ of small weight such that $[\![\theta]\!] = P$

An example was shown in Figure (1). The Boolean Synthesis problem was briefly considered in (Raha et al. 2022) (as *Boolean subset cover*), but our algorithms offer a more complete trade-off between completeness and efficiency.

Boolean Synthesis roughly corresponds to the fragment of the $\mathbf{LTL}_f$ learning problem restricted to the atomic propositions, their negations, and operators $\wedge$ and $\vee$ (i.e., without temporal operators); it is a slight generalisation of it as the atoms can have arbitrary weights for Boolean Synthesis.

We discuss the NP-completeness of Boolean Synthesis and the existence of a polynomial-time criterion for the existence of a solution of arbitrary size in Appendix B.

### 3.2 Domination for Boolean Formulas

In this section, we describe *domination*, a generalisation of observational equivalence that identifies more redundant objects when working with monotone operators.

Intuitively, we say that a formula $\theta_1$ dominates a formula $\theta_2$ if $\theta_2$ can be replaced with $\theta_1$ in a formula without increasing its weight or decreasing its quality. More formally, let $\mathsf{sat}(\theta)$ be the subset of elements of $P \cup N$ that $\theta$ classifies correctly, i.e., $\mathsf{sat}(\theta) = ([\![\theta]\!] \cap P) \cup (N \setminus [\![\theta]\!])$.

**Definition 1.** *A formula $\theta_2$ is* dominated *by $\theta_1$, denoted $\theta_2 \preceq \theta_1$, if $\mathsf{weight}(\theta_1) \leq \mathsf{weight}(\theta_2)$ and $\mathsf{sat}(\theta_2) \subseteq \mathsf{sat}(\theta_1)$.*

We claim that this notion corresponds to the above intuition (complete details in Appendix B). For a boolean formula $\theta$, let $\theta[\theta_2 \leftarrow \theta_1]$ be the formula obtained by replacing any occurrence of $\theta_2$ in $\theta$ by $\theta_1$. Any occurrence of a dominated formula can be replaced with its dominating formula:

**Lemma 1.** *Let $\theta, \theta_1$ and $\theta_2$ be boolean formulas. If $\theta_2 \preceq \theta_1$, then $\theta \preceq \theta[\theta_2 \leftarrow \theta_1]$.*

Domination can be thought of as a variation on *Property dependence* (see Theorem 6 in (Dureja and Rozier 2018)), used to prune $\mathbf{LTL}_f$ formulas for efficient model-checking.

**Further Reducing the Input Set.** Lemma 1 implies that we can preprocess $\mathcal{F}$ to only keep maximal elements for $\preceq$. By encoding the characteristic vectors of formulas in bits of integers, testing whether $\theta_1$ dominates $\theta_2$ can be implemented efficiently using bitwise operations. All our algorithms for Boolean Synthesis start by reducing $\mathcal{F}$ in this way.

**Fast Approximate Domination.** There is little hope of achieving a subquadratic algorithm to fully reduce $\mathcal{F}$ as much as possible with domination (see Appendix B). Yet, a quadratic running time is prohibitive in our application where we need to test domination for millions of boolean formulas: we therefore propose a heuristic that drastically speeds up the search of a dominating formula while allowing for some false negatives.

Instead of searching through all of $\mathcal{F}$ for dominating formulas, we restrict the search to a small subset of candidates with a high likelihood of dominating other formulas. For a given weight value $w$ and an integer $k$, let $\mathcal{T}(w, k)$ denote the $k$ formulas $\theta$ that maximise $|\mathsf{sat}(\theta)|$ among formulas of weight $w$ in $\mathcal{F}$. Our heuristic searches for a formula $\theta$ that dominates $\theta'$ in the sets $\mathcal{T}(w, k)$ for $w \leq \mathsf{weight}(\theta')$. The value chosen for $k$ controls the trade-off between speed and accuracy of the technique. We illustrate the gain in practice, which is already substantial for small values of $k$. Let $\mathcal{F}'$ be as in Algorithm 1, and $\mathcal{F}'' = \text{FASTNONDOMINATED}(\mathcal{F}', k)$ be the result of the shrinkage of $\mathcal{F}'$ by the above approximate algorithm. On average, over 780 instances, the ratio $|\mathcal{F}'|/|\mathcal{F}''|$ is 3.45 for $k = 3$, 4.07 for $k = 5$, 4.97 for $k = 10$, 5.94 for $k = 25$, and 6.60 for $k = 50$.

### 3.3 Divide and Conquer

Before describing our algorithms, we show a general procedure to extend any solver for Boolean Synthesis to mitigate the inability of the solver to find a valid solution (due to time, space, or incompleteness of the algorithm). The general idea is part of the $\mathbf{LTL}_f$ learning algorithm from (Valizadeh, Fijalkow, and Berger 2024). It is an application of *divide and conquer*: when a Boolean Synthesis solver fails to find a solution, divide $P$ or $N$ in two smaller subsets, solve the two subproblems, and combine them into a solution (see the full pseudocode in Appendix B, Algorithm 3).

The meta-algorithm first calls a Boolean Synthesis solver. In case no solution is found, the problem is broken into two subproblems. If $|P| \geq |N|$, then $P$ is randomly split into two subsets $P_1$, $P_2$ of roughly equal cardinality. From this, we consider two smaller instances of Boolean Synthesis: $(P_1, N, \mathcal{F}_{\restriction P_1 \cup N})$ and $(P_2, N, \mathcal{F}_{\restriction P_2 \cup N})$, where $\mathcal{F}_{\restriction U} = \{F \cap U \mid F \in \mathcal{F}\}$. The two subproblems are solved recursively, yielding respectively solutions $\theta_1$ and $\theta_2$; observe that $\theta = \theta_1 \cup \theta_2$ is a solution to the original problem. If $|N| > |P|$, we split $N$ and the algorithm is symmetrical. This algorithm always finds a solution if there exists one, no matter how the Boolean Synthesis solver is implemented; however, there is no guarantee of minimality.

# 4 Algorithms for Boolean Synthesis

This section presents our main three algorithms to solve Boolean Synthesis. We only describe them at a high-level; complete pseudocodes are available in Appendix C.

## 4.1 Set Cover Algorithm

Our first algorithm is called the *set cover* algorithm. It is inspired from a greedy algorithm of Johnson for the classical *set cover problem* (Johnson 1974).

We use this greedy algorithm as a subroutine, by first making multiple small set covers only focusing on elements of $P$ using unions, then combining these set covers with intersections to "cover $N$ negatively" (i.e., ensure that there remains no element in $N$). Symmetrically, we also cover $N$ negatively with intersections and then take unions of these solutions. To bound the running time, we limit the number of coverings used with a hyperparameter number-coverings. Additional details are in Appendix C, Algorithm 4.

## 4.2 Boolean Enum with Domination Pruning

Our second algorithm, *boolean enumeration with domination pruning*, enumerates all boolean formulas in increasing order of weight, with optimisations based on *domination* (Section 3.2). It can be implemented in a similar way to the VFB algorithm (Algorithm 2), with a few changes: the set of operators is restricted to $\{\cup, \cap\}$, and only a vector of size $|P| + |N|$ must be stored to evaluate the formulas and check for observational equivalence (instead of a matrix). As formulas are enumerated in increasing weight, a minimal formula is guaranteed to be returned.

In addition to the VFB algorithm, the idea of domination allows to speed up the enumeration: instead of storing formulas that are unique up to observational equivalence, we only store an antichain of all maximal elements w.r.t. $\preceq$. The full algorithm is given in Appendix C, Algorithm 5.

As described in Section 3.2, to efficiently check for domination, we use *fast approximate domination* by only using a bounded number of formulas $\theta$ with the highest $|\mathsf{sat}(\theta)|$. This bound is given by a hyperparameter size-dom. As the number of formulas increases exponentially with the weight, we bound the weight of the formulas to enumerate with a parameter D&C-switch; when that bound is reached, we split the input using divide and conquer and restart the search.

## 4.3 Beam Search Algorithm

Our third algorithm is a greedy algorithm based on *beam search*. The gist is to enumerate formulas but only keep a fixed number of "best" formulas of each weight. The notion of "best" is again based on the cardinality of $\mathsf{sat}(\theta)$. The number of formulas to keep for each weight is a hyperparameter beam-width, and the parameter D&C-switch is as for Boolean Enumeration. The procedure is described in Appendix C, Algorithm 6.

# 5 Experiments

We perform experiments to address the following questions:

1. How does BOLT perform compared to the state of the art?
2. How do our three Boolean Synthesis algorithms compare in the context of $\mathbf{LTL}_f$ learning?
3. What do the highlighted ideas of our algorithms (switch from $\mathbf{LTL}_f$ learning to Boolean Synthesis, domination) bring in terms of performance?

For each question, there are two (independent) metrics: wall-clock time and formula size.

## 5.1 Benchmarks and State of the Art

As discussed in Section 1, many tools have been constructed recently for $\mathbf{LTL}_f$ learning. A consolidated benchmark suite is still missing, but we identified three families of benchmarks used by existing tools. We contribute a fourth one. Additional details on the four families are in Appendix D.

**Flie Benchmarks.** Flie (Neider and Gavran 2018) is a tool based on a reduction of $\mathbf{LTL}_f$ learning to SAT. To evaluate Flie, the authors created a synthetic dataset starting from 9 common $\mathbf{LTL}_f$ patterns used in practice (following (Dwyer, Avrunin, and Corbett 1999)). This family includes 98 tasks[2], which are typically easy.

**Scarlet Benchmarks.** Scarlet (Raha et al. 2022, 2024a) is the state-of-the-art tool for $\mathbf{LTL}_f$ learning. It learns a specifically designed fragment of $\mathbf{LTL}_f$ using a tailored combinatorial search. The authors created a family of 445 tasks[3], some yet to be solved!

**VFB Benchmarks.** VFB (Valizadeh, Fijalkow, and Berger 2024) targets a new category: GPU-based tools, meaning that the implementation specifically takes advantage of the specificities of GPU (most importantly, massively parallel computations). The results are orders of magnitude better than CPU-based tools such as Scarlet, but this is not a fair comparison. Since our tool runs in the CPU-based track, we do not compare it against the GPU-powered VFB implementation. To evaluate how far the GPU-powered VFB tool goes, the authors created a family of very hard instances called *Hamming* and consisting of a single positive trace and many negative traces obtained by changing one or two bits from the positive one. This results in a family of 32 tasks[4], arguably far from typical industrial inputs.

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

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

[4] https://github.com/MojtabaValizadeh/ltl-learning-on-gpus

| | solved tasks (TO 60 s) | | avg. time (s) | | avg. size ratio | |
|---|---|---|---|---|---|---|
| | Scarlet | BOLT | Scarlet | BOLT | Scarlet | BOLT |
| All bench. | 424 / 676 | **637 / 676** | 26.69 | **5.96** | 2.15 | **1.56** |
| Flie bench. | **81 / 81** | **81 / 81** | 1.77 | **0.33** | **1.01** | **1.01** |
| Scarlet bench. | 271 / 354 | **328 / 354** | 18.63 | **6.81** | 1.71 | **1.25** |
| VFB bench. | 1 / 25 | **12 / 25** | 57.62 | **31.36** | 3.91 | **2.75** |
| BOLT bench. | 71 / 216 | **216 / 216** | 45.67 | **3.75** | 3.07 | **2.14** |

Table 1: Main results per benchmark families (testing set)

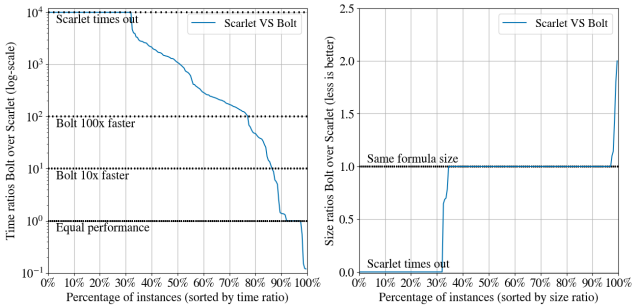



Figure 2: Running time and ratio size (testing set)

**BOLT Benchmarks.** As the experiments will show, to appreciate the progress of BOLT, we needed to generate more challenging yet realistic benchmarks. Our benchmark family is inspired by the decompositions of $\mathbf{LTL}_f$ formulas into patterns introduced in (Raha et al. 2022; Fijalkow and Lagarde 2021; Mascle, Fijalkow, and Lagarde 2023). We describe our generation procedure, which follows two steps: first, generating $\mathbf{LTL}_f$ formulas of a certain shape, and then, generating positive and negative traces from them. A *pattern* is an $\mathbf{LTL}_f$ formula in some normal form using $\mathbf{F}$ and $\mathbf{X}$. Here is an example: $\mathbf{F}(p \wedge \mathbf{X}(q \wedge \mathbf{X}\, r))$, where $p$, $q$, and $r$ are atomic propositions. The formulas we consider are boolean combinations of patterns. To generate such formulas, we sample both the boolean formula combining the patterns, and each of the patterns independently. The second step is, given a formula, to generate positive and negative traces. The exact algorithm is explained in Appendix D; it ensures that the generated traces cover the patterns as accurately as possible. We include a family of 270 tasks.

Putting the four families together, we obtain a set of 845 tasks. Towards creating a consolidated benchmark suite for evaluating further progress on $\mathbf{LTL}_f$ learning, we will release (once anonymisation is lifted) scripts for generating benchmarks for the four generation methods described above, together with an improved JSON-based format.

As our algorithms use hyperparameters, we distinguish a (randomly chosen) *training set* consisting of 20% of each of the four families from a *testing set* containing the remaining 80%. The choice of hyperparameters was conducted on the training set and other experiments on the testing set.

## 5.2 Experiments

Since Scarlet improved over its competitors by a large margin, we take it as a reference point for the state of the art. Scarlet is an "anytime" algorithm, meaning that it outputs a stream of formulas, each of them solutions for the task but of decreasing size, and it only guarantees that the latest output formula is minimal (in the targeted "directed fragment of $\mathbf{LTL}_f$") when the algorithm terminates. When reporting on time for Scarlet, we report how long it took to output the last formula, and *not* termination time, which is typically a lot longer (and often beyond the timeout of 60 s).

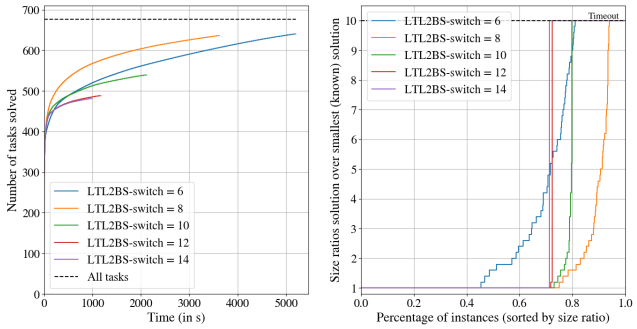



Figure 3: Influence of parameter LTL2BS-switch using Beam Search (training set)

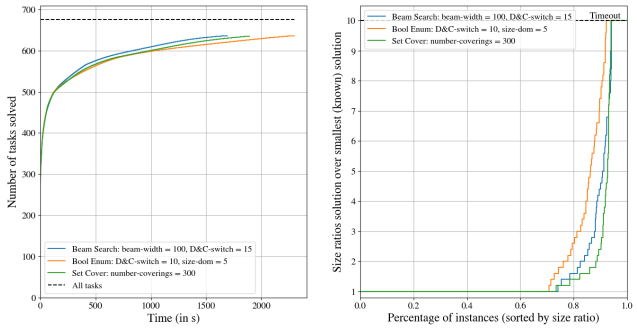
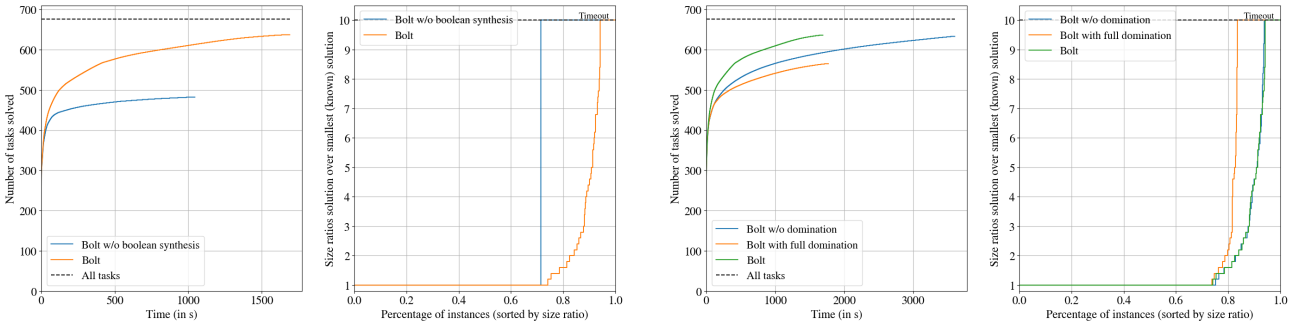



Figure 4: Comparison of the three algorithms

Below, "BOLT" refers to our VFB algorithm implementation with beam search as a Boolean Synthesis solver with LTL2BS-switch $= 8$, beam-width $= 100$, and D&C-switch $= 15$. We will argue in experiments that it is our most competitive algorithm. When comparing formula size, we use *size ratio*, which is the ratio between the size of the returned formula and the shortest formula found by all our runs using Scarlet and our algorithms with various hyperparameters. No algorithm uses randomisation and we observe negligible performance shifts on runs over the same instances.

The experiments were conducted on identical nodes of the Grid'5000 cluster, running Debian GNU/Linux 5.10.0-28-amd64. The hardware configuration includes an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz and 126GB of RAM. All our algorithms, including BOLT, are implemented in the Rust programming language, version 1.79.0 (`129f3b996`, 2024-06-10). Our code was compiled in `release` mode.

**Measuring wallclock time and formula size.** We compare BOLT against the state-of-the-art tool Scarlet (Raha et al. 2022) on the testing set in Table 1 and Figure 2. Table 1 shows that BOLT performs better on all benchmarks




(a) Ablation study: removing Boolean Synthesis

(b) Ablation study: removing domination

Figure 5: Ablation studies (testing set)

families: it returns a solution for more benchmarks, with a better average running time and smaller average size ratio. More precisely, Figure 2 shows that BOLT returns solutions more than 100x faster over 75% of the benchmarks, with formulas of smaller or equal size in over 95% of the cases.

**Comparing different algorithms for Boolean Synthesis.** We compare our new algorithms and show experiments to select hyperparameter values on the training set.

The first hyperparameter, LTL2BS-switch, appears in Algorithm 1 and is common to all our algorithms. We see in Figure 3 that among tested values, 8 appears the best both for running time and for size ratio. Higher values of LTL2BS-switch lead to more timeouts: more time is spent enumerating small $\mathbf{LTL}_f$ formulas, but less time is spent exploring large formulas with Boolean Synthesis solvers. With smaller values, about the same number of benchmarks are solved, but Boolean Synthesis solvers have access to fewer $\mathbf{LTL}_f$ formulas, leading to longer solves and formulas of much larger sizes. All further experiments therefore use LTL2BS-switch $= 8$.

We describe the effect of the five other algorithm-specific hyperparameters on time and formula sizes; this analysis is backed up by graphs provided in Appendix E.

For the Set Cover algorithm (Section 4.1), as expected, allowing for more coverings (i.e., larger values for number-coverings) leads to a longer running time, but also smaller formulas (Figure 6).

For Boolean Enumeration (Section 4.2): first, larger values of D&C-switch (Figure 7) lead to fewer instances returning a solution, but with shorter solutions on the solved instances; second, tweaking the values of size-dom (Figure 8) has only a small influence on time, and has, as expected, no effect on the formula sizes.

For Beam Search (Section 4.3), out of five values tested for beam-width (Figure 9), it appears that 100 solves more benchmarks faster, with little influence on the formula size. Parameter D&C-switch (Figure 10) has little influence on the time, but higher values tend to produce smaller formulas.

Finally, we choose hyperparameter values for each algorithm and compare them in Figure 4. Boolean Enumeration is the worst over both metrics, but Beam Search is the fastest while Set Cover finds slightly smaller formulas on average. To highlight BOLT in other experiments, we select Beam Search with beam-width $= 100$ and D&C-switch $= 15$.

**Ablation Studies.** We perform two ablation studies on the testing set to appreciate the impact of our two main ideas: Boolean Synthesis and domination. In the first experiment (Figure 5a) we compare the raw VFB algorithm against BOLT to evaluate the impact of Boolean Synthesis. The raw VFB algorithm solves far fewer tasks (461 vs 637), even given the same task budget, hence clearly Boolean Synthesis improves over the existing algorithms. Observe that the raw VFB algorithm either finds the smallest formula or times out, as expected. In the second experiment (Figure 5b), we consider three scenarios: without using domination at all, using full domination, and using fast approximate domination (as done in BOLT). Again, the interpretation is clear: domination makes solving faster, but does not allow solving more tasks, and full domination being too time-consuming implies solving fewer tasks.

## 6 Perspectives

The first contribution of this paper is a framework for combining $\mathbf{LTL}_f$ learning with Boolean Synthesis. We proposed three algorithms for Boolean Synthesis, and through extensive experimental analysis showed that our new tool BOLT greatly improves over the state of the art.

These results yield evidence towards our main belief: Boolean Synthesis is a fundamental problem of independent interest, which has a lot of potential applications beyond $\mathbf{LTL}_f$ learning. Our framework is generic; for instance, it can be adapted to any kind of temporal operators, and beyond that to any logic or specification language with disjunctions and conjunctions. Natural candidates include regular expressions (Valizadeh and Berger 2023), Boolean circuits (Chowdhury et al. 2024), and Bit-vector programs (Ding and Qiu 2024).

Our paper provides a first step towards efficiently and approximately solving Boolean Synthesis. We believe there are algorithms that achieve better trade-offs between compute time and size of learnt formulas. The directions outlined above contribute hard benchmarks, much needed to push further the theory and practice of Boolean Synthesis!

## References


Ammons, G.; Bodík, R.; and Larus, J. R. 2002. Mining Specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'02, 4–16. New York, NY, USA: Association for Computing Machinery. ISBN 1581134509.

Arif, M. F.; Larraz, D.; Echeverria, M.; Reynolds, A.; Chowdhury, O.; and Tinelli, C. 2020. SYSLITE: Syntax-Guided Synthesis of PLTL Formulas from Finite Traces. In *Formal Methods in Computer Aided Design*, 93–103. IEEE.

Asarin, E.; Donzé, A.; Maler, O.; and Nickovic, D. 2012. Parametric Identification of Temporal Properties. In Khurshid, S.; and Sen, K., eds., *Runtime Verification*, 147–160. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-29860-8.

Bartocci, E.; Bortolussi, L.; Nenzi, L.; and Sanguinetti, G. 2015. System design of stochastic models using robustness of temporal properties. *Theoretical Computer Science*, 587: 3–25.

Bartocci, E.; Bortolussi, L.; and Sanguinetti, G. 2014. Data-Driven Statistical Learning of Temporal Logic Properties. In Legay, A.; and Bozga, M., eds., *International Conference on Formal Modeling and Analysis of Timed Systems*, volume 8711 of *Lecture Notes in Computer Science*, 23–37. Springer.

Bombara, G.; Vasile, C. I.; Penedo Alvarez, F.; Yasuoka, H.; and Belta, C. 2016. A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Hybrid Systems: Computation and Control*, HSCC'16.

Borassi, M.; Crescenzi, P.; and Habib, M. 2015. Into the Square: On the Complexity of Some Quadratic-time Solvable Problems. In Crescenzi, P.; and Loreti, M., eds., *Proceedings of the 16th Italian Conference on Theoretical Computer Science, ICTCS 2015, Firenze, Italy, September 9–11, 2015*, volume 322 of *Electronic Notes in Theoretical Computer Science*, 51–67. Elsevier.

Brafman, R. I.; Giacomo, G. D.; and Patrizi, F. 2018. LTLf/LDLf Non-Markovian Rewards. In McIlraith, S. A.; and Weinberger, K. Q., eds., *AAAI Conference on Artificial Intelligence*, 1771–1778. AAAI Press.

Bufo, S.; Bartocci, E.; Sanguinetti, G.; Borelli, M.; Lucangelo, U.; and Bortolussi, L. 2014. Temporal Logic Based Monitoring of Assisted Ventilation in Intensive Care Patients. In Margaria, T.; and Steffen, B., eds., *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, 391–403. Springer.

Camacho, A.; Chen, O.; Sanner, S.; and McIlraith, S. A. 2017. Non-Markovian Rewards Expressed in LTL: Guiding Search Via Reward Shaping. In Fukunaga, A.; and Kishimoto, A., eds., *International Symposium on Combinatorial Search*, 159–160. AAAI Press.

Camacho, A.; Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2019. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In Kraus, S., ed., *International Joint Conference on Artificial Intelligence*, 6065–6073. ijcai.org.

Camacho, A.; and McIlraith, S. A. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. *International Conference on Automated Planning and Scheduling*, 29.

Caplain, M. 1975. Finding Invariant assertions for proving programs. In Shooman, M. L.; and Yeh, R. T., eds., *International Conference on Reliable Software 1975*, 165–171. ACM.

Chou, G.; Ozay, N.; and Berenson, D. 2020. Explaining Multi-stage Tasks by Learning Temporal Logic Formulas from Suboptimal Demonstrations. In Toussaint, M.; Bicchi, A.; and Hermans, T., eds., *Robotics: Science and Systems XVI*.

Chowdhury, A. B.; Romanelli, M.; Tan, B.; Karri, R.; and Garg, S. 2024. Retrieval-Guided Reinforcement Learning for Boolean Circuit Minimization. In *International Conference on Learning Representations, ICLR*. OpenReview.net.

Christodorescu, M.; Jha, S.; and Kruegel, C. 2007. Mining specifications of malicious behavior. In Crnkovic, I.; and Bertolino, A., eds., *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 5–14. ACM.

Ding, Y.; and Qiu, X. 2024. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. *Proceedings of the ACM on Programming Languages*, 8(POPL): 2129–2159.

Dureja, R.; and Rozier, K. Y. 2018. More Scalable LTL Model Checking via Discovering Design-Space Dependencies ( D^3 D 3 ). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 10805 of *Lecture Notes in Computer Science*, 309–327. Springer.

Dwyer, M. B.; Avrunin, G. S.; and Corbett, J. C. 1999. Patterns in Property Specifications for Finite-State Verification. In *International Conference on Software Engineering, ICSE*, 411–420. ACM.

Engler, D. R.; Chen, D. Y.; and Chou, A. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In Marzullo, K.; and Satyanarayanan, M., eds., *ACM Symposium on Operating System Principles*, 57–72. ACM.

Fijalkow, N.; and Lagarde, G. 2021. The complexity of learning linear temporal formulas from examples. In *International Conference on Grammatical Inference*, volume 153 of *ICGI'21*, 237–250. Proceedings of Machine Learning Research.

Gabbrielli, M.; Levi, G.; and Meo, M. C. 1992. Observational Equivalences for Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, 131–145.

Gabel, M.; and Su, Z. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'08, 339–349. New York,

NY, USA: Association for Computing Machinery. ISBN 9781595939951.

Ghosh, S.; Elenius, D.; Li, W.; Lincoln, P.; Shankar, N.; and Steiner, W. 2016. ARSENAL: Automatic Requirements Specification Extraction from Natural Language. In Rayadurgam, S.; and Tkachuk, O., eds., *International Symposium on NASA Formal Methods*, volume 9690 of *Lecture Notes in Computer Science*, 41–46. Springer.

Giacomo, G. D.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In Rossi, F., ed., *International Joint Conference on Artificial Intelligence*, IJCAI'13, 854–860. IJCAI/AAAI.

Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; Rhein, J.; Schumann, J.; and Shi, N. 2020. Formal Requirements Elicitation with FRET. In *International Conference on Requirements Engineering: Foundation for Software Quality*, volume 2584 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Gupta, A.; Komp, J.; Rajput, A. S.; Krishna, S.; Trivedi, A.; and Varshney, N. 2024. Integrating Explanations in Learning LTL Specifications from Demonstrations. *CoRR*, abs/2404.02872.

Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2018a. Teaching Multiple Tasks to an RL Agent using LTL. In André, E.; Koenig, S.; Dastani, M.; and Sukthankar, G., eds., *International Conference on Autonomous Agents and MultiAgent Systems*, 452–461. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM.

Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2018b. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In Dy, J. G.; and Krause, A., eds., *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 2112–2121. PMLR.

Ielo, A.; Law, M.; Fionda, V.; Ricca, F.; De Giacomo, G.; and Russo, A. 2023. Towards ILP-Based LTLf Passive Learning. In *Inductive Logic Programming*, 30–45. Cham: Springer Nature Switzerland. ISBN 978-3-031-49299-0.

Isik, I.; Gol, E. A.; and Cinbis, R. G. 2024. Learning to Estimate System Specifications in Linear Temporal Logic using Transformers and Mamba. *CoRR*, abs/2405.20917.

Jeppu, N.; Melham, T.; and Kroening, D. 2023. Enhancing active model learning with equivalence checking using simulation relations. *Formal Methods in System Design*.

Jeppu, N.; Melham, T.; Kroening, D.; and O'Leary, J. 2020. Learning Concise Models from Long Execution Traces. In *ACM/IEEE Design Automation Conference*, DAC'20, 1–6.

Jeppu, N. Y.; Melham, T.; and Kroening, D. 2022. Active Learning of Abstract System Models from Traces Using Model Checking. In *Conference & Exhibition on Design, Automation & Test in Europe*, DATE'22, 100–103. Leuven, BEL: European Design and Automation Association. ISBN 9783981926361.

Jin, X.; Donzé, A.; Deshmukh, J. V.; and Seshia, S. A. 2015. Mining Requirements From Closed-Loop Control Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11): 1704–1717.

Johnson, D. S. 1974. Approximation Algorithms for Combinatorial Problems. *J. Comput. Syst. Sci.*, 9(3): 256–278.

Jones, A.; Kong, Z.; and Belta, C. 2014. Anomaly detection in cyber-physical systems: A formal methods approach. In *IEEE Conference on Decision and Control*, 848–853. IEEE.

Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In Miller, R. E.; and Thatcher, J. W., eds., *Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, 85–103. Plenum Press, New York.

Kim, J.; Muise, C.; Shah, A.; Agarwal, S.; and Shah, J. 2019. Bayesian Inference of Linear Temporal Logic Specifications for Contrastive Explanations. In Kraus, S., ed., *International Joint Conference on Artificial Intelligence*, 5591–5598. ijcai.org.

Kong, Z.; Jones, A.; and Belta, C. 2017. Temporal Logics for Learning and Detection of Anomalous Behavior. volume 62, 1210–1222.

Lemieux, C.; and Beschastnikh, I. 2015. Investigating Program Behavior Using the Texada LTL Specifications Miner. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE'15, 870–875. Los Alamitos, CA, USA: IEEE Computer Society.

Lemieux, C.; Park, D.; and Beschastnikh, I. 2015. General LTL Specification Mining. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE'15, 81–92. Los Alamitos, CA, USA: IEEE Computer Society.

Li, W. 2013. *Specification Mining: New Formalisms, Algorithms and Applications*. Ph.D. thesis, University of California, Berkeley, USA.

Li, Z.; and Zhou, Y. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In Wermelinger, M.; and Gall, H. C., eds., *European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 306–315. ACM.

Lo, D.; Khoo, S.-C.; Han, J.; and Liu, C. 2017. *Mining Software Specifications: Methodologies and Applications*. USA: CRC Press, Inc., 1st edition. ISBN 1138114901.

Luo, W.; Liang, P.; Du, J.; Wan, H.; Peng, B.; and Zhang, D. 2022. Bridging LTLf Inference to GNN Inference for Learning LTLf Formulae. *AAAI Conference on Artificial Intelligence*, 36(9): 9849–9857.

Lyu, M.; Li, F.; Lee, C.-H.; and Chen, C.-H. 2024. VALIO: Visual attention-based linear temporal logic method for explainable out-of-the-loop identification. *Knowledge-Based Systems*, 299: 112086.

Mascle, C.; Fijalkow, N.; and Lagarde, G. 2023. Learning temporal formulas from examples is hard. arXiv:2312.16336.

Murphy, W.; Holzer, N.; Koenig, N.; Cui, L.; Rothkopf, R.; Qiao, F.; and Santolucito, M. 2024. Guiding LLM Temporal Logic Generation with Explicit Separation of Data and Control. *CoRR*, abs/2406.07400.

Neider, D.; and Gavran, I. 2018. Learning Linear Temporal Properties. In *Formal Methods in Computer Aided Design*, FMCAD'18, 1–10.

Peng, B.; Liang, P.; Han, T.; Luo, W.; Du, J.; Wan, H.; Ye, R.; and Zheng, Y. 2023. PURLTL: Mining LTL Specification from Imperfect Traces in Testing. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE'23, 1766–1770. Los Alamitos, CA, USA: IEEE Computer Society.

Pnueli, A. 1977. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, SFCS'77.

Pommellet, A.; Stan, D.; and Scatton, S. 2024. SAT-based Learning of Computation Tree Logic. In *International Joint Conference on Automated Reasoning*, 366–385.

Raha, R.; Roy, R.; Fijalkow, N.; and Neider, D. 2022. Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *TACAS'22*, 263–280. Springer.

Raha, R.; Roy, R.; Fijalkow, N.; and Neider, D. 2024a. Scarlet: Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic. *Journal of Open Source Software*.

Raha, R.; Roy, R.; Fijalkow, N.; Neider, D.; and Pérez, G. A. 2024b. Synthesizing Efficiently Monitorable Formulas in Metric Temporal Logic. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'24.

Rozier, K. Y. 2016. Specification: The Biggest Bottleneck in Formal Methods and Autonomy. In Blazy, S.; and Chechik, M., eds., *International Conference on Verified Software. Theories, Tools, and Experiments*, volume 9971 of *Lecture Notes in Computer Science*, 8–26.

Shah, A.; Kamath, P.; Shah, J. A.; and Li, S. 2018. Bayesian Inference of Temporal Task Specifications from Demonstrations. In Bengio, S.; Wallach, H. M.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Annual Conference on Neural Information Processing Systems 2018*, 3808–3817.

Valizadeh, M.; and Berger, M. 2023. Search-Based Regular Expression Inference on a GPU. *Proceedings of the ACM on Programming Languages*, 7(PLDI): 1317–1339.

Valizadeh, M.; Fijalkow, N.; and Berger, M. 2024. LTL Learning on GPUs. In *International Conference on Computer Aided Verification, CAV*, volume 14683 of *Lecture Notes in Computer Science*, 209–231. Springer.

Wegbreit, B. 1974. The Synthesis of Loop Predicates. *Communications of the ACM*, 17(2): 102–112.

Williams, R. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3): 357–365.

Yang, H.; Hoxha, B.; and Fainekos, G. E. 2012. Querying Parametric Temporal Logic Properties on Embedded Systems. In Nielsen, B.; and Weise, C., eds., *IFIP International Conference on Testing Software and Systems*, volume 7641 of *Lecture Notes in Computer Science*, 136–151. Springer.

Yang, J.; Evans, D.; Bhardwaj, D.; Bhat, T.; and Das, M. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *International Conference on Software Engineering*, ICSE'06, 282–291. New York, NY, USA: Association for Computing Machinery. ISBN 1595933751.

Zeller, A. 2010. Mining Specifications: A Roadmap. In Nanz, S., ed., *The Future of Software Engineering*, 173–182. Springer.

# A Additional Pseudocodes for Section 2

For completeness, we show in Algorithm 2 the complete pseudocode for the VFB algorithm (Section 2.2). The way we enumerate all formulas of size $k+1$ is as follows: for a unary operator, we take all formulas of size $k$ in order to produce formulas of size $k+1$ and, for a binary operator, we iterate over all pairs of formulas whose sizes sum to $k$: $(1, k-1), (2, k-2), \ldots$

**Algorithm 2** VFB algorithm

**procedure** VFB($\mathsf{AP}, O_1, O_2, \mathsf{CM}$) ▷ $O_n$ is the set of $n$-ary operators, $\mathsf{CM}$ takes a formula and computes its characteristic matrix
  $\mathcal{F}_1 \leftarrow \mathsf{AP} \cup \{\neg a \mid a \in \mathsf{AP}\}$
  $M \leftarrow \{\mathsf{CM}(a) \mid a \in \mathsf{AP}\} \cup \{\mathsf{CM}(\neg a) \mid a \in \mathsf{AP}\}$
  $k \leftarrow 1$
  **while** True **do**
    $\mathcal{F}_{k+1} \leftarrow \emptyset$
    **for** $\mathsf{op} \in O_1$ **do**
      **for** $\varphi \in \mathcal{F}_k$ **do**
        $\varphi' \leftarrow \mathsf{op}(\varphi)$ ▷ Formula of size $k+1$
        **if** $\varphi'$ is a solution **then**
          **return** $\varphi'$
        **else if** there is no $\psi$ such that $\psi \sim \varphi'$ **then**
          $\mathcal{F}_{k+1} \leftarrow \mathcal{F}_{k+1} \cup \{\varphi'\}$
          $M \leftarrow M \cup \{\mathsf{CM}(\varphi')\}$
    **for** $\mathsf{op} \in O_2$ **do**
      **for** $i, j \geq 1, i+j=k$ **do**
        **for** $\varphi_1 \in \mathcal{F}_i, \varphi_2 \in \mathcal{F}_j$ **do**
          $\varphi' \leftarrow \mathsf{op}(\varphi_1, \varphi_2)$ ▷ Formula of size $k+1$
          **if** $\varphi'$ is a solution **then**
            **return** $\varphi'$
          **else if** there is no $\psi$ such that $\psi \sim \varphi'$ **then**
            $\mathcal{F}_{k+1} \leftarrow \mathcal{F}_{k+1} \cup \{\varphi'\}$
            $M \leftarrow M \cup \{\mathsf{CM}(\varphi')\}$
    $k \leftarrow k+1$ ▷ No solution of size $k+1$

As discussed in Section 2.2, we show how to compute inductively the characteristic sequences of $\mathbf{LTL}_f$ formulas on traces with few bitwise operations:

1. $\mathbf{X}\,\varphi$ is just a (left) shift by 1 of the characteristic sequences of $\varphi$, padding it with a 0.
2. $\neg\varphi$ is the bitwise negation of the characteristic sequences.
3. $\varphi \wedge \psi$ is the bitwise "and" of the characteristic sequences of the two formulas; the same can be done for "or".
4. $\mathbf{F}\,\varphi$ can be seen as $\varphi \vee \mathbf{X}\,\varphi \vee \mathbf{X}^2\,\varphi \vee \ldots$ Since we work with finite traces, this converges to a fixpoint in finite time. We can actually compute this efficiently with a *logarithmic* number of shifts in the trace length: simply do shifts of powers of 2 up to the length of the trace. For example, with the sequence 0000000100000001, first shift by 1 and do an "or": it gives us 0000001100000011. Shift by 2 and do an "or", we get: 0000111100001111. Now we shift by 4 and do an "or", giving us only 1's. If instead we have 0000000000000001, we also need to shift by 8.
5. $\mathbf{G}\,\varphi = \neg\mathbf{F}\,\neg\varphi$.
6. $\varphi\,\mathbf{U}\,\psi$ can be done similarly to $\mathbf{F}\,\varphi$ since it is $\psi \vee (\varphi \wedge \mathbf{X}\,\psi) \vee (\varphi \wedge \mathbf{X}(\varphi \wedge \mathbf{X}\,\psi)) \vee \ldots$ Intuitively, first compute $\varphi \wedge \mathbf{X}\,\psi$ for all positions; it suffices to do an "and" on a shifted $\psi$. Now, following the same protocol as $\mathbf{F}\,\varphi$, it suffices to do the same shifts and "or", but with the addition that before the "or", as $\varphi$ needs to be true at all positions before, we must do an "and" with the shifted $\varphi$.

We include Python code for the fast inductive evaluation of $\mathbf{LTL}_f$ formulas, based on bitwise operations. These algorithms describe the general idea of how to compute iteratively the characteristic sequences of formulas from existing ones.

```
def X(phi):
  return phi << 1
```

```
def logic_not(phi):
  return ~phi
def logic_and(phi, psi):
  return phi & psi
def F(phi):
  out = phi
  # Assume trace lengths <= 64
  for shift in [1, 2, 4, 8, 16, 32]:
    out |= (out << shift)
  return out
def U(phi, psi):
  out = psi
  # Assume trace lengths <= 64
  for shift in [1, 2, 4, 8, 16, 32]:
    out |= ((out << shift) & phi)
    phi = (phi << shift)
  return out
```

## B Additional Details for Section 3

We give additional theoretical properties of Boolean Synthesis (Section 3.1).

**Polynomial Criterion for a Solution.** We show that we can check whether there exists a formula $\theta$ (*of arbitrary weight*) such that $[\![\theta]\!] = P$ in polynomial time, just by evaluating a specific formula of length $\mathcal{O}(|P| \cdot |\mathcal{F}|)$. This allows to quickly decide the nonexistence of a solution, and when a solution exists, it gives a polynomial upper bound on the weight of formulas to consider.

**Lemma 2** (Existence of a solution)**.** *Let $P$, $N$, $\mathcal{F}$ be an instance of Boolean Synthesis. There exists a boolean formula $\theta$ such that $[\![\theta]\!] = P$ if and only if for all $p \in P$, $n \in N$, there is $F \in \mathcal{F}$ such that $p \in F$ and $n \notin F$. When there is a solution, there is a solution of size $\mathcal{O}(|\mathcal{F}| \cdot |P|)$.*

*Proof.* We show the equivalence. For the left-to-right implication, we consider the contrapositive. Assume that there is $p \in P$ and $n \in N$ such that, for all $F \in \mathcal{F}$, $p \in F$ implies $n \in F$. One can then show by induction that any formula $\theta'$ such that $p \in [\![\theta']\!]$ is also such that $n \in [\![\theta']\!]$.

For the right-to-left implication, we build a boolean formula $\theta$ such that $[\![\theta]\!] = P$. This formula happens to be of size $\mathcal{O}(|\mathcal{F}| \cdot |P|)$, thereby settling at the same time the claim about the size. Assume that for all $p \in P$, $n \in N$, there is $F \in \mathcal{F}$ such that $p \in F$ and $n \notin F$. For $p \in P$, let $\theta_p = \bigcap_{F \in \mathcal{F}, p \in F} F$ and $\theta = \bigcup_{p \in P} \theta_p$. Due to the hypothesis, for all $p \in P$, we have $p \in [\![\theta_p]\!]$ and $N \cap [\![\theta_p]\!] = \emptyset$. Hence, $[\![\theta]\!] = P$. □

**Complexity.** Boolean Synthesis is a generalisation of the NP-complete *set cover* problem (Karp 1972): an instance of the set cover problem corresponds to an instance of Boolean Synthesis with $N = \emptyset$ (in which case using $\cap$ is futile, and minimal formulas can be obtained using only $\cup$). This proves NP-hardness of the decision problem for Boolean Synthesis. Since there are short solutions by Lemma 2, and since computing the set described by a boolean formula is polynomial in its length, it is NP-complete.

**Domination.** We now restate and prove the claim on domination (Section 3.2).

We define formally the notation $\theta[\theta_2 \leftarrow \theta_1]$. For a boolean formula $\theta$, let $\theta[\theta_2 \leftarrow \theta_1]$ denote the formula obtained by replacing any occurrence of $\theta_2$ in $\theta$ by $\theta_1$; this operation is defined inductively as follows:

$$F_i[\theta_2 \leftarrow \theta_1] = \begin{cases} \theta_1 & \text{if } F_i = \theta_2 \\ F_i & \text{otherwise,} \end{cases}$$

$$\theta_2[\theta_2 \leftarrow \theta_1] = \theta_1$$

$$(\theta \cap \theta')[\theta_2 \leftarrow \theta_1] = \theta[\theta_2 \leftarrow \theta_1] \cap \theta'[\theta_2 \leftarrow \theta_1]$$

$$\text{and } (\theta \cup \theta')[\theta_2 \leftarrow \theta_1] = \theta[\theta_2 \leftarrow \theta_1] \cup \theta'[\theta_2 \leftarrow \theta_1].$$

**Lemma 1.** *Let $\theta, \theta_1$ and $\theta_2$ be boolean formulas. If $\theta_2 \preceq \theta_1$, then $\theta \preceq \theta[\theta_2 \leftarrow \theta_1]$.*

*Proof.* First, note that, as $P$ and $N$ are disjoint, then if $\mathsf{sat}(\theta_2) \subseteq \mathsf{sat}(\theta_1)$ we have:

$$[\![\theta_2]\!] \cap P \subseteq [\![\theta_1]\!] \cap P \text{ and } N \setminus [\![\theta_2]\!] \subseteq N \setminus [\![\theta_1]\!]. \tag{1}$$

We now show that the $\mathsf{sat}(\cdot)$ function is *monotone* with respect to the $\cup$ and $\cap$ operators: if $\mathsf{sat}(\theta_2) \subseteq \mathsf{sat}(\theta_1)$, then for any formula $\theta$, we have $\mathsf{sat}(\theta_2 \cap \theta) \subseteq \mathsf{sat}(\theta_1 \cap \theta)$ and $\mathsf{sat}(\theta_2 \cup \theta) \subseteq \mathsf{sat}(\theta_1 \cup \theta)$. This can be shown through direct calculation using Eq. (1):

$$\begin{aligned}
\mathsf{sat}(\theta_2 \cap \theta) &= ([\![\theta_2 \cap \theta]\!] \cap P) \cup (N \setminus [\![\theta_2 \cap \theta]\!]) \\
&= ([\![\theta_2]\!] \cap [\![\theta]\!] \cap P) \cup (N \setminus ([\![\theta_2]\!] \cap [\![\theta]\!])) \\
&= (([\![\theta_2]\!] \cap P) \cap [\![\theta]\!]) \cup ((N \setminus [\![\theta_2]\!]) \cup (N \setminus [\![\theta]\!])) \\
&\subseteq (([\![\theta_1]\!] \cap P) \cap [\![\theta]\!]) \cup ((N \setminus [\![\theta_1]\!]) \cup (N \setminus [\![\theta]\!])) \qquad (2) \\
&= ([\![\theta_1]\!] \cap [\![\theta]\!] \cap P) \cup (N \setminus ([\![\theta_1]\!] \cap [\![\theta]\!])) \\
&= \mathsf{sat}(\theta_1 \cap \theta).
\end{aligned}$$

The inclusion in Eq. (2) follows from Eq. (1). A similar calculation shows the inclusion for the $\cup$ operator.

Similarly, the $\mathsf{weight}(\cdot)$ function is also monotone, in the sense that if $\mathsf{weight}(\theta_1) \leq \mathsf{weight}(\theta_2)$, then $\mathsf{weight}(\theta_1 \cap \theta) \leq \mathsf{weight}(\theta_2 \cap \theta)$ (and similarly for $\cup$) for any formula $\theta$.

We can now prove that if $\theta_2 \preceq \theta_1$, then $\theta \preceq \theta[\theta_2 \leftarrow \theta_1]$ by structural induction over $\theta$. The base cases follow from the definition of domination, and the cases $\theta = \theta' \cap \theta''$ or $\theta = \theta' \cup \theta''$ are handled using the monotonicity of the $\mathsf{sat}(\cdot)$ and $\mathsf{weight}(\cdot)$ functions. □

**The Need for a Heuristic.** We argue that the existence of a subquadratic algorithm to completely reduce a set $\mathcal{F}$ with domination is unlikely. The obvious algorithm to find whether a given formula $\theta \in \mathcal{F}$ is dominated by a formula $\theta' \in \mathcal{F}$ is to go through every formula in $\mathcal{F}$ and check whether it dominates $\theta$. This takes linear time, and when there are $O(n)$ formulas $\theta$ and $\theta'$, this approach takes quadratic time. We do not expect a faster algorithm: the *Orthogonal Vectors problem* (Williams 2005) can be reduced to finding whether there exists a formula $\theta_1 \in A$ that dominates a formula $\theta_2 \in B$ where $A, B$ are sets of $n$ formulas, and a strongly subquadratic algorithm for Orthogonal Vectors would imply that the Strong Exponential Time Hypothesis is false. The reductions between our problem and the Orthogonal Vectors problem is presented in (Borassi, Crescenzi, and Habib 2015).

**Divide and Conquer.** We show the complete pseudocode for the divide-and-conquer algorithm in Algorithm 3 (Section 3.3).

A useful optimisation is to use the solution from the first subproblems to simplify the second subproblems further; if $\theta_1$ is the returned solution to the instance $(P_1, N, \mathcal{F}_{\restriction P_1 \cup N})$, it may already contain elements of $P_2$. It therefore suffices to solve the instance $(P_2 \setminus [\![\theta_1]\!], N, \mathcal{F}_{\restriction (P_2 \setminus [\![\theta_1]\!]) \cup N})$ for the second subproblem. This optimisation means that the solution may differ depending on the order in which we solve the two subproblems.

We add a remark about the base case (line 17) to argue for the completeness of the algorithm: as per Lemma 2, such an $F$ necessarily exists if there is a solution, and failure to exist immediately indicates that there is no solution to the Boolean Synthesis problem. This shows that the algorithm always returns a valid solution if there exists one, no matter how BS-SOLVER is implemented.

## C Pseudocodes for the Boolean Synthesis Algorithms

We show the full pseudocodes of our three algorithms for Boolean Synthesis: Algorithm 4 for the Set Cover algorithm (Section 4.1), Algorithm 5 for Boolean Enumeration (Section 4.2), and Algorithm 6 for Beam Search (Section 4.3). We also discuss additional implementation details for each algorithm.

**Additional Details for Set Cover.** The simple algorithm of Johnson for the classical set cover problem (Johnson 1974) is as follows: it iteratively adds a set maximizing the number of newly covered elements. This algorithm achieves an approximation factor of $1 + \log(n)$, where $n$ is the number of elements to cover.

We use this greedy algorithm as a subroutine, by first making multiple small set covers only focusing on elements of $P$ using unions, then combining these set covers with intersections to "cover $N$ negatively" (i.e., ensure that there remains no element in $N$). Symmetrically, we also first cover $N$ negatively with intersections and then take unions of these solutions. This algorithm only generates boolean formulas that are intersections of unions or unions of intersections (i.e., $\mathbf{\Pi}_2^0$ and $\mathbf{\Sigma}_2^0$ formulas). From an expressiveness point of view, this is not a restriction: the proof of Lemma 2 shows that when a solution exists, there is always a solution of this kind.

To ensure a good variety of coverings, the multiple coverings use every set in $\mathcal{F}$ at most once; as soon as a set is used for one covering, we remove the set from $\mathcal{F}$.

**Algorithm 3** Divide and conquer (meta-algorithm for Boolean Synthesis)

**procedure** DIVCONQ($\mathcal{F}, P, N$)
  **while** $|P| > 1$ or $|N| > 1$ **do**
    $\theta \leftarrow$ BS-SOLVER($\mathcal{F}, P, N$)
    **if** $[\![\theta]\!] = P$ **then** ▷ Valid solution
      **return** $\theta$
    **if** $|P| \geq |N|$ **then**
      $P_1, P_2 \leftarrow$ split $P$ randomly into two subsets
      $\theta_1 \leftarrow$ DIVCONQ($\mathcal{F}_{\restriction P_1 \cup N}, P_1, N$)
      $\theta_2 \leftarrow$ DIVCONQ($\mathcal{F}_{\restriction P_2 \cup N}, P_2, N$)
      **return** $\theta_1 \cup \theta_2$
    **else**
      $N_1, N_2 \leftarrow$ split $N$ randomly into two subsets
      $\theta_1 \leftarrow$ DIVCONQ($\mathcal{F}_{\restriction P \cup N_1}, P, N_1$)
      $\theta_2 \leftarrow$ DIVCONQ($\mathcal{F}_{\restriction P \cup N_2}, P, N_2$)
      **return** $\theta_1 \cap \theta_2$
  take $p \in P$, $n \in N$ ▷ Here, $P = \{p\}$ and $N = \{n\}$
  **return** $F \in \mathcal{F}$ of minimal weight s.t. $p \in F, n \notin F$

**Algorithm 4** Set cover algorithm

**Hyperparameter**: number-coverings (below, $k$), the maximal number of set coverings.

**procedure** GREEDYSETCOVERSPOSITIVE($\mathcal{F}, P, k$)
  $\mathcal{F}' \leftarrow$ copy of $\mathcal{F}$
  $\mathcal{C} \leftarrow$ empty set ▷ $\mathcal{C}$ will contain coverings of $P$
  $\theta \leftarrow \emptyset$ ▷ We assume $[\![\emptyset]\!] = \emptyset$
  **while** $\mathcal{F}'$ is not empty and $|\mathcal{C}| \leq k$ **do**
    **if** $P \subseteq [\![\theta]\!]$ **then**
      add $\theta$ to $\mathcal{C}$
      $\theta \leftarrow \emptyset$
    **else** ▷ $\theta$ does not cover $P$ yet
      $F \leftarrow \operatorname{argmax}_{F \in \mathcal{F}'} |[\![\theta \cup F]\!] \cap P|$ ▷ Maximise the covering by an extra set, exit if no progress is made
      $\theta \leftarrow \theta \cup F$
      Remove $F$ from $\mathcal{F}'$
  **return** $\mathcal{C}$
**procedure** GREEDYSETCOVERSNEGATIVE($\mathcal{F}, N, k$)
  $\mathcal{F}' \leftarrow$ copy of $\mathcal{F}$
  $\mathcal{C} \leftarrow$ empty set ▷ $\mathcal{C}$ will contain "negative coverings" of $N$
  $\theta \leftarrow \emptyset$ ▷ We assume $[\![\emptyset]\!] = N$
  **while** $\mathcal{F}'$ is not empty and $|\mathcal{C}| \leq k$ **do**
    **if** $N \cap [\![\theta]\!] = \emptyset$ **then**
      add $\theta$ to $\mathcal{C}$
      $\theta \leftarrow \emptyset$
    **else** ▷ $\theta$ does not cover $N$ yet
      $F \leftarrow \operatorname{argmin}_{F \in \mathcal{F}'} |[\![\theta \cap F]\!] \cap N|$ ▷ Minimise the covering by an extra set, exit if no progress is made
      $\theta \leftarrow \theta \cap F$
      Remove $F$ from $\mathcal{F}'$
  **return** $\mathcal{C}$
**procedure** SETCOVER($\mathcal{F}, P, N, k$)
  $\mathcal{C}_P \leftarrow$ GREEDYSETCOVERSPOSITIVE($\mathcal{F}, P$)
  $\mathcal{C}_N \leftarrow$ GREEDYSETCOVERSNEGATIVE($\mathcal{F}, N$)
  $\mathcal{C}_{PN} \leftarrow$ GREEDYSETCOVERSNEGATIVE($\mathcal{C}_P, N$)
  $\mathcal{C}_{NP} \leftarrow$ GREEDYSETCOVERSPOSITIVE($\mathcal{C}_N, P$)
  **return** formula of minimal weight in $\mathcal{C}_{PN} \cup \mathcal{C}_{NP}$

**Additional Details for Boolean Enumeration.** We provide the full pseudo of Boolean Enumeration in Algorithm 5. Note that the idea of domination pruning while enumerating could also be used for $\mathbf{LTL}_f$ formulas in the VFB algorithm to go beyond observational equivalence. However, it turns out that, in practice, dominations are much rarer for characteristic matrices generated with all $\mathbf{LTL}_f$ operators than for smaller characteristic vectors generated with $\cup$ and $\cap$. Due to the large number of formulas, the size of the characteristic matrices, and the rarity of dominations, we believe that it is unlikely that an implementation of the VFB algorithm can gain time by checking for dominations.

**Algorithm 5** Boolean enumeration algorithm with domination pruning

**Hyperparameters**:

- D&C-switch (not shown in the pseudocode), the weight up to which to enumerate before splitting with DIVCONQ;
- size-dom (not shown in the pseudocode), the maximal number of formulas to use to check for domination.

**procedure** BOOLEANENUMERATION($P, N, \mathcal{F}$)
  **for** $k = 1$ to $\max_{F \in \mathcal{F}} \mathsf{weight}(F)$ **do** ▷ Store sets in $\mathcal{F}$ by weight
    $\mathcal{F}_k \leftarrow \{F \in \mathcal{F} \mid \mathsf{weight}(F) = k\}$
  $k \leftarrow 2$
  $\mathcal{M} \leftarrow \mathcal{F}_1 \cup \mathcal{F}_2$
  **while** True **do**
    **for** $i, j \geq 1, i + j = k$ **do**
      **for** $\theta_1 \in \mathcal{F}_i, \theta_2 \in \mathcal{F}_j$ **do**
        **for** $\mathsf{op} \in \{\cup, \cap\}$ **do**
          $\theta \leftarrow \theta_1 \,\mathsf{op}\, \theta_2$ ▷ $\theta$ has weight $k + 1$
          **if** there exists $\theta' \in \mathcal{M}$ such that $\theta' \sim \theta$ **then**
            **continue**
          **if** there exists $\theta' \in \mathcal{M}$ such that $\theta \preceq \theta'$ **then**
            **continue**
          **if** $[\![\theta]\!] = P$ **then**
            **return** $\theta$
          $\mathcal{F}_{k+1} \leftarrow \mathcal{F}_{k+1} \cup \{\theta\}$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{F}_{k+1}$
    $k \leftarrow k + 1$ ▷ No solution of weight $k + 1$

**Additional Details for Beam Search.** We provide the full pseudo of Beam Search in Algorithm 6. For each weight, a "min" priority queue pq stores the at most $b$ formulas of this weight with the largest score, where the priority queue allows for an efficient query of the stored formula with the lowest score. The enumeration then proceeds as for Boolean Enumeration.

## D Description of the benchmark families for Section 5

We give more thorough comments on all four benchmark families (Section 5.1).

**Flie Benchmarks.** Flie (Neider and Gavran 2018) is a tool based on a reduction of $\mathbf{LTL}_f$ learning to SAT. To evaluate Flie, the authors created a synthetic dataset as follows: starting from 9 common $\mathbf{LTL}_f$ patterns used in practice (following (Dwyer, Avrunin, and Corbett 1999)), they generated random traces and categorised them as positive and negative. This family includes 98 tasks[5], which are typically easy.

**Scarlet Benchmarks.** Scarlet (Raha et al. 2022, 2024a) is the state-of-the-art tool for $\mathbf{LTL}_f$ learning. It learns a specifically designed fragment of $\mathbf{LTL}_f$ using a tailored combinatorial search. The authors created harder tasks in a similar fashion: starting from handwritten formulas (or parameterised families of formulas), they generated traces in a more efficient and fair way, by compiling formulas into deterministic automata and sampling accepting or rejecting paths. This results in a family of 445 tasks[6], some yet to be solved!

**VFB Benchmarks.** VFB (Valizadeh, Fijalkow, and Berger 2024) targets a new category: GPU-based tools, meaning that the implementation specifically takes advantage of the specificities of GPU (most importantly, massively parallel

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

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

**Algorithm 6** Beam search algorithm

**Hyperparameters:**

- D&C-switch (not shown in the pseudocode), the weight up to which to enumerate before splitting with DIVCONQ;
- beam-width (below, $b$), the maximal number of promising formulas to store for each weight.

**procedure** ADDBOUNDED($\theta, P, N, \mathsf{pq}, b$)
    **if** $\mathsf{size}(\mathsf{pq}) < b$ or $\mathsf{score}(\theta) > \mathsf{score}(\min(\mathsf{pq}))$ **then**
        Insert $\theta$ into $\mathsf{pq}$
    **if** $\mathsf{size}(\mathsf{pq}) > b$ **then**
        Pop minimal element from $\mathsf{pq}$
**procedure** BEAMSEARCH($P, N, \mathcal{F}, b$)
    $\mathsf{pq}_1, \ldots, \mathsf{pq}_{\max_{F \in \mathcal{F}} \mathsf{weight}(F)} \leftarrow$ empty priority queues
    **for** $F \in \mathcal{F}$ **do**
        ADDBOUNDED($F, P, N, \mathsf{pq}_{\mathsf{weight}(F)}, b$)
    $k \leftarrow 2$
    $\mathcal{M} \leftarrow \mathsf{pq}_1 \cup \mathsf{pq}_2$
    **while** True **do**
        **for** $i, j \geq 1, i + j = k$ **do**
            **for** $\theta_1 \in \mathsf{pq}_i, \theta_2 \in \mathsf{pq}_j$ **do**
                **for** $\mathsf{op} \in \{\cup, \cap\}$ **do**
                    $\theta \leftarrow \theta_1 \,\mathsf{op}\, \theta_2$ ▷ $\theta$ has weight $k + 1$
                    **if** there exists $\theta' \in \mathcal{M}$ such that $\theta' \sim \theta$ **then**
                        **continue**
                    **if** $[\![\theta]\!] = P$ **then**
                        **return** $\theta$
                    ADDBOUNDED($\theta, P, N, \mathsf{pq}_{k+1}, b$)
        $\mathcal{M} \leftarrow \mathcal{M} \cup \mathsf{pq}_{k+1}$
        $k \leftarrow k + 1$

computations). The results are orders of magnitude better than CPU-based tools such as Scarlet, but this is not a fair comparison. Since our tool runs in the CPU-based track, we do not compare it against the GPU-powered VFB implementation. To evaluate how far the GPU-powered VFB tool goes, the authors created a family of very hard instances called *Hamming* and consisting of a single positive trace and many negative traces obtained by changing one or two bits from the positive one. This results in a family of 32 tasks[7], arguably far from typical industrial inputs.

**BOLT Benchmarks.** As the experiments will show, to appreciate the progress of BOLT, we needed to generate more challenging yet realistic benchmarks. Our benchmark family is inspired by the decompositions of $\mathbf{LTL}_f$ formulas into patterns introduced in (Raha et al. 2022; Fijalkow and Lagarde 2021; Mascle, Fijalkow, and Lagarde 2023). We describe our generation procedure, which follows two steps: first, generating $\mathbf{LTL}_f$ formulas of a certain shape, and then, generating positive and negative traces from them. A *pattern* is an $\mathbf{LTL}_f$ formula in some normal form using $\mathbf{F}$ and $\mathbf{X}$. Here is an example: $\mathbf{F}(p \wedge \mathbf{F}(q \wedge \mathbf{X}\, r))$, where $p$, $q$, and $r$ are atomic propositions. The formulas we consider are boolean combinations of patterns. To generate such formulas, we sample both the boolean formula combining the patterns, and each of the patterns independently. The second step is, given a formula, to generate positive and negative traces.

- To generate positive traces, we first go down the boolean formula. For conjunctions, we need to satisfy both subformulas, and for disjunctions, we flip a coin to choose which subformula the trace will satisfy. Once we reach a pattern, we randomly choose a subset of non-overlapping positions of the corresponding size, and fix the propositions in this subset. The rest of the trace is chosen uniformly at random.

  For instance for the pattern $\mathbf{F}(p \wedge \mathbf{F}(q \wedge \mathbf{X}\, r))$ we may choose the subset of positions $\{4, 9, 10\}$, fixing that $p$ holds in position $4$, $q$ in position $9$, and $r$ in position $10$.
- To generate negative traces, we simply generate random traces until we find ones that do not satisfy the formula, which happens typically quickly.

We include a family of 270 tasks.

## E Additional graphs for Section 5

We include here additional graphs to justify the choice of hyperparameters for our three Boolean Synthesis solvers. This complements the text in Section 5.2.

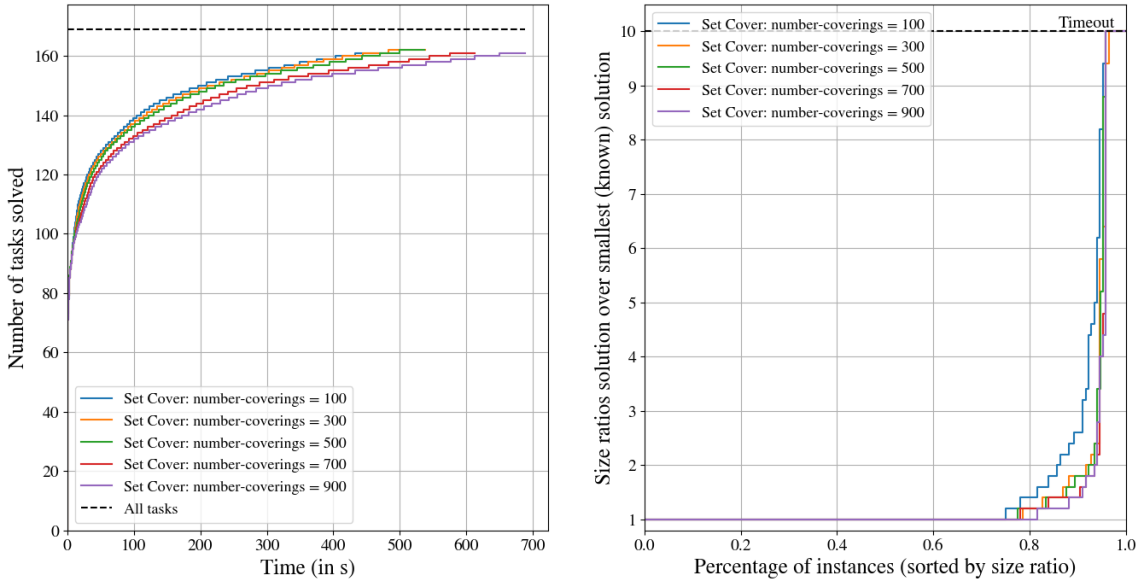



Figure 6: Influence of number-coverings for Set Cover

[7]https://github.com/MojtabaValizadeh/ltl-learning-on-gpus

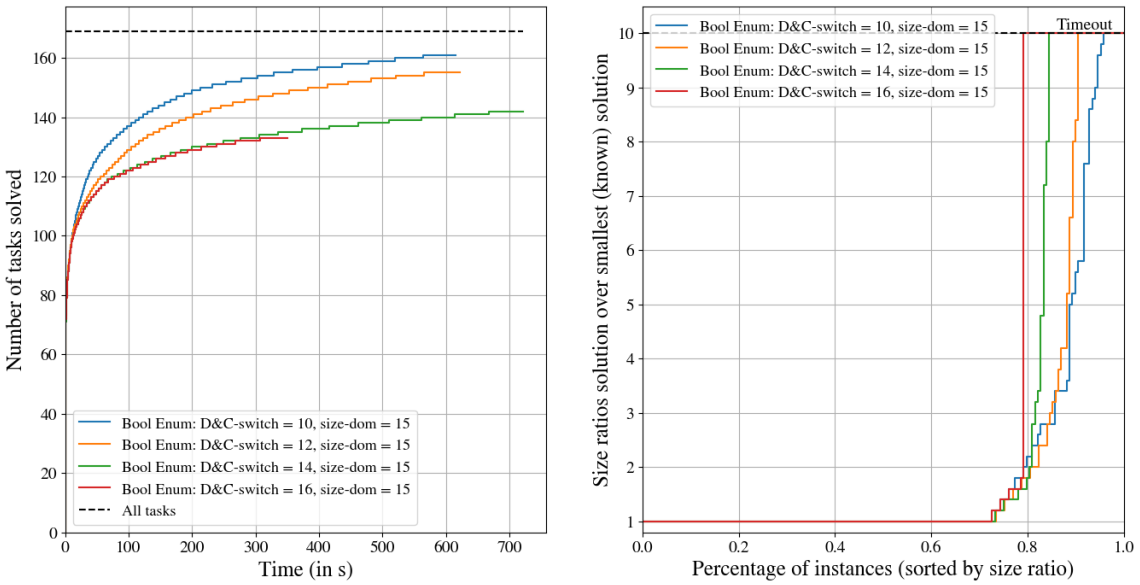



Figure 7: Influence of D&C-switch for Boolean Enumeration

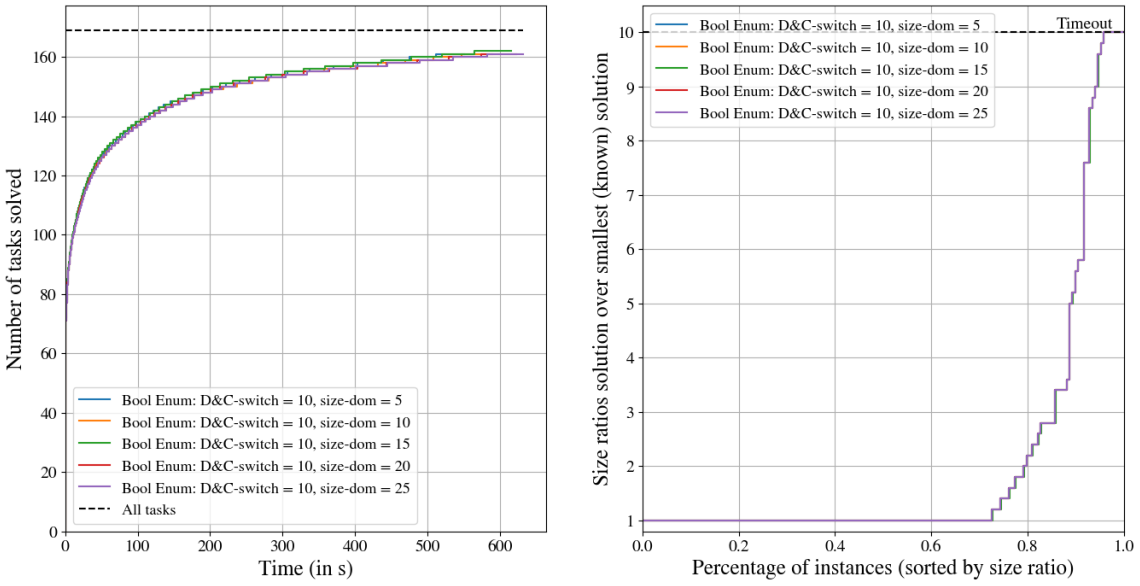



Figure 8: Influence of size-dom for Boolean Enumeration

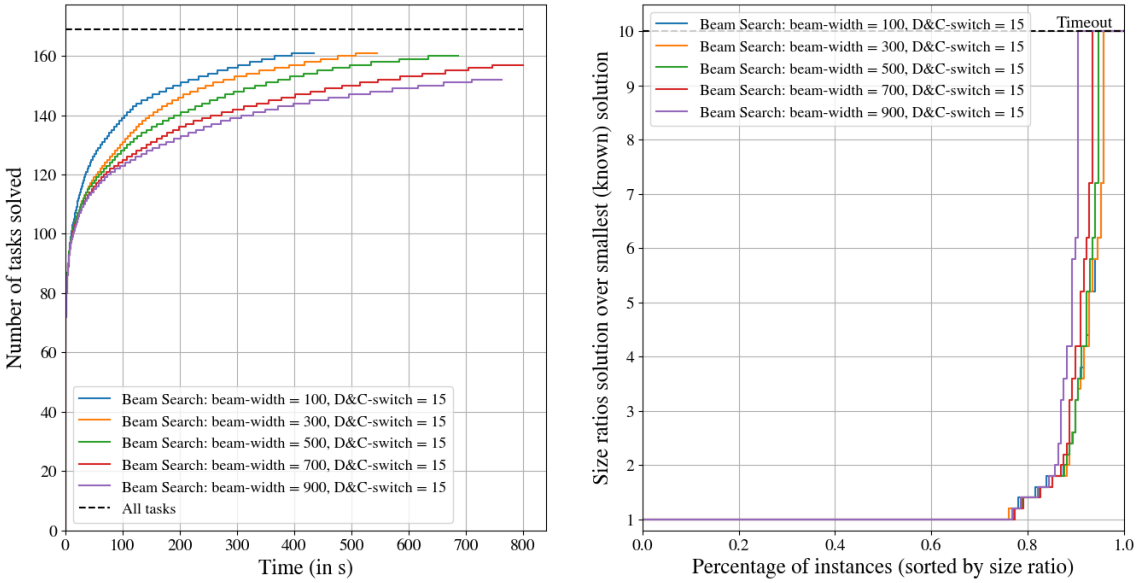



Figure 9: Influence of beam-width for Beam Search

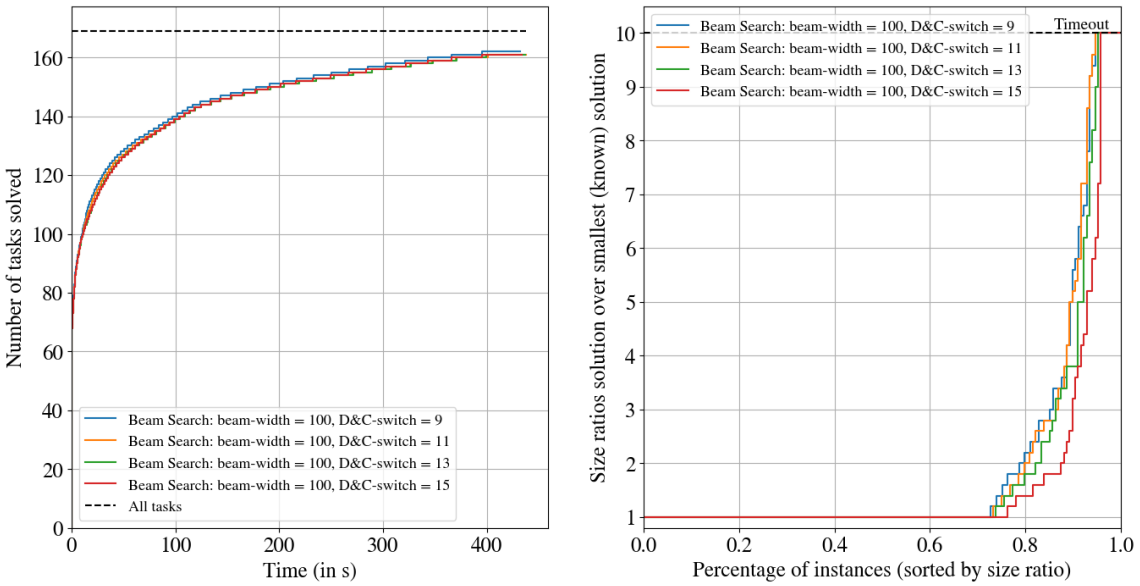



Figure 10: Influence of D&C-switch for Beam Search

## F Reproducibility Checklist

This paper:

- Includes a conceptual outline and/or pseudocode description of AI methods introduced. **Yes**
- Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results. **Yes**
- Provides well marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper. **Yes**

Does this paper make theoretical contributions? **No**
Does this paper rely on one or more datasets? **Yes**
If yes, please complete the list below.

- A motivation is given for why the experiments are conducted on the selected datasets. **Yes**
- All novel datasets introduced in this paper are included in a data appendix. **Yes**
- All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. **Yes**
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. **Yes**
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available. **Yes**
- All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing. **NA**

Does this paper include computational experiments? **Yes**
If yes, please complete the list below.

- Any code required for pre-processing data is included in the appendix. **Yes**.
- All source code required for conducting and analyzing the experiments is included in a code appendix. **Yes, though since we used a laboratory cluster for experiments, it may not adapt to your structure out of the box; however, it is trivial to adapt it.**
- All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. **Yes**
- All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from **No, it is not present in the source code since it we changed some steps however it is fully described in the paper.**
- If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results. **Yes**
- This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks. **Yes**
- This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics. **Yes**
- This paper states the number of algorithm runs used to compute each reported result. **Yes**
- Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information. **No, as our algorithms are all deterministic.**
- The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank). **NA: since our algorithm are deterministic, there is no reason to use a statistical test.**
- This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments. **Yes**
- This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting. **Yes**