Learning Interpretable Behaviour Classifiers for PDDL Planning

Arnaud Lequen

IRIT, Université Toulouse III - Paul Sabatier
arnaud.lequen@irit.fr

Résumé

On cherche à synthétiser des modèles interprétables reconnaissant le comportement d'un agent parmi d'autres, et ce sur tout un domaine de planification, exprimé en PDDL. Nous proposons d'apprendre des formules logiques, à partir d'un ensemble d'exemples de taille réduite, qui montrent la solution apportée par l'agent à un ensemble de petites instances. Ces formules sont exprimées dans une variante de la Logique Temporelle du Premier Ordre (FTL) adaptée au formalisme de la planification automatique. De telles formules sont lisibles par un humain, et peuvent être vues comme des explications (partielles) de la politique mise en œuvre par un agent. Notre méthode consiste à apprendre de tels classifieurs de comportements au travers d'une compilation vers MaxSAT topologiquement guidée qui nous permet d'apprendre une grande variété de formules. Une étude expérimentale montre que notre implémentation peut apprendre des formules intéressantes en temps raisonnable.

Abstract

We consider the problem of synthesizing interpretable models that recognize the behaviour of an agent out of many others, on a whole set of planning problems expressed in PDDL. Our approach consists in learning logical formulas, from a set of small examples that show how an agent solved small planning instances. These formulas are expressed in a version of First-Order Temporal Logic (FTL) tailored to our planning formalism. Such formulas are human-readable, and serve as (partial) explanations of an agent's policy. We propose to learn such behaviour classifiers through a topology-guided compilation to MaxSAT, which allows us to generate a wide range of different formulas. Experiments show that interesting formulas can be learned in reasonable time.

1 Introduction

One of the main strengths of PDDL planning models is that they are succinct and human-readable, but can nonetheless express general, complex problems, whose state search spaces are exponential in the size of the encoding – as can be the solutions. As a consequence, given a set of examples of the behaviour of an agent (called traces), understanding and recognizing this behaviour can be tedious.

In order to summarize the behaviour of a planning agent in a concise, interpretable way, we propose to learn properties that are specific to the solutions proposed by this agent. Such properties, expressed in a temporal logic tailored to fit PDDL planning models, are not only human-readable, but are also general, and can be evaluated against different instances of the same planning problem. This allows them to recognize the behaviour of an agent on instances that are substantially different from the ones used in the set of examples.

More specifically, the problem we tackle is the one where, given a set of positive example traces (the ones of the agent we seek to recognize) and negative examples traces (the ones of other agents), we wish to learn a model that can discriminate as well as possible between positive and negative traces. A wide variety of techniques and models of different natures have been proposed in the literature. Among these, the learning of finite-state automata (DFA) is a well-studied problem [1, 20, 21], but DFAs can grow quickly (thus becoming harder to interpret) and do not generalize to instances not in the example set. More recently, neural network-based architectures such as LSTMs [22] have shown very promising results, but lack interpretability, and the rationale for their decision is rarely clear.

In the past decade, significant efforts have been made towards learning logical formulas expressed in (a form of) temporal logic. Such works [18, 19, 8, 15, 3, 4] often leverage symbolic methods to learn Linear Temporal Logic (LTL) formulas [17] that fit the example traces, and thus share some similarities with our work. Some other authors propose other techniques, such as Latent Dirichlet Allocation [12], which stems from the field of natural language processing.

However, in all of these cases, the knowledge extracted from the sets of examples has the major drawback of not generalizing well to unknown instances. This is due to the

choice of the language used to express these properties. For instance, since LTL formulas are built over a set of propositional variables, they do not generalize to models that do not share the same variables.

To address this issue, we propose to learn properties in a version of First-Order Temporal Logic (FTL). When tailored to the PDDL planning formalism, FTL can express a wide range of properties that generalize from one planning instance to the other, given that they model similar problems. This was shown in [2], who proposed to express *search control knowledge* in a language similar to ours, albeit with the aim of guiding the search of a planner designed to use such knowledge. In [4], the authors proposed to synthesize such control knowledge automatically, and thus address the problem of learning properties expressed in a fragment of FTL.

In this paper, we show that it is possible to learn richer and more expressive properties, using the whole range of FTL operators and modalities. The properties we wish to learn should describe the behaviour of a given planning agent, without being true for the behaviour of other agents. We show that learning such formulas is computationally intractable, as the associated decision problem is NP-hard. This is why the core of our approach consists in encoding the learning problem into a MaxSAT instance, which has the added benefit of showing resilience to any potential noise in the set of training examples. To make the search more efficient, we fix the general topology of the target formula before the encoding. In addition to alleviating the load on the MaxSAT solver and rendering the algorithm more parallelizable, this also increases the diversity in the formulas learned by our algorithm, thus providing varied descriptions of the behaviour of the agent of interest.

Our article is organised as follows: Section 2 introduces the planning formalism as well as the FTL language. Section 3 formally introduces the learning problem we tackle in this paper, and shows that the associated decision problem is intractable. Sections 4 and 5 present some technical choices that we made to solve our problem in reasonable time in practice. In Section 6, we describe our reduction of the problem to MaxSAT, and in Section 7, we present our experimental results, as well as a few examples of formulas that are within reach of our implementation.

2 Background

2.1 Planning with PDDL

This section introduces the model that we use to describe planning tasks. Our definition of a PDDL planning task differs from [9], for instance, as we require the organization of the objects of our instances into types. The model we use resembles the one defined in [11]

Definition 1 (Type tree) *A* type tree $\mathcal{T}$ *is a non-empty tree where each node is labeled by a symbol, called a* type*. For any type $\tau \in \mathcal{T}$, we call* strict subtype *any descendant τ' of τ. τ' is a subtype of τ (denoted $\tau' \preceq \tau$) when τ' is a strict subtype of τ or when $\tau' = \tau$.*

Definition 2 (Object class) *Let O be a set of elements called* objects*. We call* object class *any subset of O. A class c_i is said to be a* subclass *of type c_j if $c_i \subseteq c_j$.*

Definition 3 (Type hierarchy) *A* type hierarchy $\mathcal{H}$ *over type tree $\mathcal{T}$ is a set of object classes such that $O \in \mathcal{H}$, and such that each object class of $\mathcal{H}$ is mapped to a unique type of $\mathcal{T}$. This mapping $\tau : \mathcal{H} \to \mathcal{T}$ is such that for any pair c_i, c_j of object classes:*

- *c_i is a subclass of c_j iff $\tau(c_i)$ is a subtype of $\tau(c_j)$ (and conversely);*
- *$c_i \cap c_j = \emptyset$ iff $\tau(c_i)$ is not a subtype of $\tau(c_j)$ (and conversely).*

We say that object $o \in O$ is of type $\tau(o) := \tau(c)$ where c is the smallest (for inclusion $\subseteq$) class of $\mathcal{H}$ to which o belongs.

Definition 4 (Predicate, atoms and fluents) *A* predicate *p is a symbol, which is associated with:*

- *An arity $ar(p) \in \mathbb{N}$*
- *A type for each of its arguments. For $i \in \{1, \dots, ar(p)\}$, the type of its argument at position i is denoted $\tau_p(i) \in \mathcal{T}$*

An atom *is a predicate for which each argument is associated with a symbol, which can be a variable symbol, or an object of O. When the i-th argument of the atom is an object o (associated to type hierarchy $\mathcal{H}$), then we require that $\tau(o) = \tau_p(i)$. The atom consisting of predicate p and symbols $x_1, \dots, x_{ar(p)}$ is denoted $p(x_1, \dots, x_{ar(p)})$.*

A fluent *is an atom where each argument is associated an object of O. A* state *is a set of fluents.*

Definition 5 (Action schema and operators) *An* action schema *is a tuple $a = \langle \mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a) \rangle$, such that $\mathsf{pre}(a)$, $\mathsf{add}(a)$ and $\mathsf{del}(a)$ are sets of* atoms *instantiated with variables only.*

An operator *o is akin to an action schema, except that the sets $\mathsf{pre}(\mathsf{o})$, $\mathsf{add}(\mathsf{o})$ and $\mathsf{del}(\mathsf{o})$ are sets of fluents.*

This leads us to the main definition of this section, which crystallizes the elements above.

Definition 6 (PDDL planning problem) *A PDDL planning problem is a pair $\Pi = \langle \mathcal{D}, \mathcal{I} \rangle$ where $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{T} \rangle$ is the domain and $\mathcal{I} = \langle O, \mathcal{H}, I, G \rangle$ is the instance.*

The domain $\mathcal{D}$ consists of a set $\mathcal{P}$ of predicates, a set of actions schemas $\mathcal{A}$, and a type hierarchy $\mathcal{T}$.

The instance $\mathcal{I}$ consists of a set of objects O and an associated type hierarchy $\mathcal{H}$, as well as two states, I and G, which are respectively the initial state and goal.

An operator o is applicable in a state s if $\mathsf{pre}(\mathsf{o}) \subseteq s$. The state that results of the application of o on s is $s[\mathsf{o}] = (s \setminus \mathsf{del}(\mathsf{o})) \cup \mathsf{add}(\mathsf{o})$.

A sequence of operators $\mathsf{o}_1, \ldots, \mathsf{o}_n$ is called a *plan* for Π if there exists a sequence of states $s_0, \ldots, s_n$ where $s_0 = I$, and which is such that, for all $i \in \{1, \ldots, n\}$, $s_i = s_{i-1}[\mathsf{o}_i]$ and o_i is applicable in s_{i-1}. Such a sequence of states (which is unique for each plan) is called a *trace*. A plan is called a *solution-plan* if, in addition to this, $G \subseteq s_n$.

We will say that a fluent $p(o_1, \ldots, o_{ar(p)})$ is true in state s iff $p(o_1, \ldots, o_{ar(p)}) \in s$.

2.2 First-Order Temporal Logic

This section introduces the language in which are expressed the formulas that we attempt to learn throughout this paper.

Syntax Let $\mathcal{X}$ be a set of variable symbols, $\mathcal{P}$ a set of predicates, and $\mathcal{T}$ a type tree. We define our language $\mathcal{L}_{\text{FTL}}$ such that:

$$\psi := \exists x \in \tau.\psi \mid \forall x \in \tau.\psi \mid \varphi$$

where $\varphi \in \mathcal{L}_{\text{TL}}$, and $\mathcal{L}_{\text{TL}}$ is such that:

$$\varphi := \top \mid p(x, \ldots, x) \mid \neg\varphi \mid \bigcirc\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \overline{\bigcirc}\varphi \mid \overline{\Diamond}\varphi \mid \overline{\Box}\varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi$$

where x is a variable of $\mathcal{X}$, p a predicate of $\mathcal{P}$, and τ a type. In the following, we will denote $\Lambda = \{\wedge, \vee, \Rightarrow, \mathsf{U}, \bigcirc, \Diamond, \Box\}$ the set of all logical operators. For each operator $\lambda \in \Lambda$, we also note $ar(\lambda) \in \{1, 2\}$ the arity of the operator.

This formulation is akin to Linear Temporal Logic on finite traces (LTL_f) [17], where propositional variables are replaced with first-order predicates and variables. Notice that we only work with formulas in prenex normal form.

Semantics Any quantifier-free formula φ of $\mathcal{L}_{\text{TL}}$ can be evaluated against a trace $t = (s_0, \ldots, s_n)$, at any step. When $i \in [\![0, n]\!]$, we write $t, i \models \varphi$ to denote that formula φ is true at state s_i of trace t. In that case, temporal modalities, such as $\bigcirc$, $\Diamond$, $\Box$, etc., are used to reason over the states that follow or precede the current state s_i.

For instance, $\bigcirc\varphi$ intuitively means that property φ is true in the next state, while $\Diamond\varphi$ means that φ is eventually true, in one of the (iterated) successors of the current state. $\Box\varphi$ means that φ is true from this state on, until the end of the trace, and $\varphi_1 \mathsf{U} \varphi_2$ means that φ_2 is true in some successor state, and until then, φ_1 is true. Operators $\overline{\bigcirc}$, $\overline{\Diamond}$ and $\overline{\Box}$ are the respective *past* counterparts of the previous connectors: $\overline{\bigcirc}\varphi$ means that φ is true in the previous state, $\overline{\Diamond}\varphi$ that φ is true in some previous state, and $\overline{\Box}\varphi$ that φ is true in every previous state.

To illustrate the language, we introduce the Childsnack problem, which originates from the International Planning Competition (IPC). It consists in making sandwiches and serving them to a group of children, some of whom are allergic to gluten. Sandwiches can only be prepared in the kitchen, and then have to be put on trays, which is the only way they can be brought to the children for service. Among the following FTL formulas, the first indicates that "All children will eventually be served" (and will be satisfied by any solution-plan). The second formula indicates that every sandwich x will eventually be put on some tray, at a moment $t+1$. For every moment that precedes moment t, x will not be prepared yet (which indicates that the sandwich is actually put on the tray right after being prepared).

$$\forall x \in \text{Child}.\ \Diamond\, \text{served}(x) \tag{1}$$

$$\forall x \in \text{Sandwich}.\ \exists y \in \text{Tray}.\ \text{notprepared}(x)\ \mathsf{U}\ \bigcirc\, \text{on}(x, y) \tag{2}$$

Temporal modalities can be expressed in terms of one another - the same way propositional connectors can be expressed in terms of one another. For any quantifier-free formula φ, we have $\Diamond\varphi \equiv \top \mathsf{U} \varphi$, $\Box\varphi \equiv \neg\Diamond\neg\varphi$ and $\overline{\Box}\varphi \equiv \neg\overline{\Diamond}\neg\varphi$. This leads us to an inductive definition of the semantics of our language, for quantifier-free formulas of $\mathcal{L}_{\text{TL}}$:

$$\begin{aligned}
&t, i \models p(x, \ldots, x) && \text{iff } p(x, \ldots, x) \in s_i \\
&t, i \models \neg\varphi && \text{iff } t, i \not\models \varphi \\
&t, i \models \varphi_1 \wedge \varphi_2 && \text{iff } t, i \models \varphi_1 \text{ and } t, i \models \varphi_2 \\
&t, i \models \bigcirc\varphi && \text{iff } i < n \text{ and } t, (i+1) \models \varphi \\
&t, i \models \overline{\bigcirc}\varphi && \text{iff } i > 0 \text{ and } t, (i-1) \models \varphi \\
&t, i \models \overline{\Diamond}\varphi && \text{iff } \exists j \in [\![0, i]\!] \text{ s.t. } t, j \models \varphi \\
&t, i \models \varphi_1 \mathsf{U} \varphi_2 && \text{iff } \exists j \in [\![i, n]\!] \text{ s.t. } t, j \models \varphi_2 \\
& && \quad \text{and } \forall k \in [\![i, j-1]\!], t, k \models \varphi_1
\end{aligned}$$

We write $t \models \varphi$ as a shorthand for $t, 0 \models \varphi$, which means that trace t satisfies the formula φ, since it is true in the initial state of t.

A formula $\psi \in \mathcal{L}_{\text{FTL}}$ is evaluated against instantiated traces, as defined below:

Definition 7 (Instantiated trace) *An* instantiated trace *is a pair $\langle t, \mathcal{I} \rangle$ such that t is a trace where fluents are built on the objects of the planning instance $\mathcal{I}$.*

For any formula ϕ of $\mathcal{L}_{\text{FTL}}$, let us denote $\phi\,[x/y]$ the formula of $\mathcal{L}_{\text{FTL}}$ where each occurrence of y is replaced by x. The semantics of $\mathcal{L}_{\text{FTL}}$ is defined as follows:

$$\begin{aligned}
&\langle t, \mathcal{I} \rangle \models \forall x \in \tau.\psi && \text{iff for all } o \in \mathcal{O} \text{ s.t. } \tau(o) = \tau, \\
& && \quad \langle t, \mathcal{I} \rangle \models \psi\,[o/x] \\
&\langle t, \mathcal{I} \rangle \models \exists x \in \tau.\psi && \text{iff there exists } o \in \mathcal{O} \text{ s.t. } \tau(o) = \tau, \\
& && \quad \langle t, \mathcal{I} \rangle \models \psi\,[o/x] \\
&\langle t, \mathcal{I} \rangle \models \varphi && \text{iff } t \models \varphi
\end{aligned}$$

where x is a variable, and φ is a formula of $\mathcal{L}_{\text{TL}}$ (thus quantifier-free).

Note that it is well known that the past modalities do not change the expressivity of LTL. As a consequence, our language could have expressed the same properties without modalities $\overline{\bigcirc}$, $\overline{\Diamond}$ or $\overline{\Box}$. However, as these modalities can make some properties exponentially more succinct to express [14], we chose to include them in our language.

2.3 The MaxSAT problem

Let Var be a set of propositional variables. The Boolean satisfiability problem (SAT) is concerned with finding a valuation that satisfies a propositional formula ϕ. Propositional formulas are defined as follows, where $x \in \text{Var}$ is a propositional variable:

$$\phi := \top \mid x \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

The maximum Boolean satisfiability problem (MaxSAT) is a variant of SAT, in which a valuation of the variables Var of a set of formulas $\{\phi_1, \ldots, \phi_n\}$ is sought. Each formula ϕ_i is assigned a *weight* $w(\phi_i) \in \mathbb{R} \cup \{\infty\}$. The MaxSAT problem consists in finding a valuation v of Var such that the sum of the weights of the formulas that are not satisfied by v is minimal.

3 The $\mathcal{L}_{\text{FTL}}$ learning problem

In this section, we introduce the main problem we are tackling in this paper. In the following, we use $[\langle t, \mathcal{I}\rangle \models \psi]$ as a shorthand for the function equal to 1 if $\langle t, \mathcal{I}\rangle \models \psi$ and equal to 0 otherwise.

Problem 1 *$\mathcal{L}_{FTL}$ learning*

Input: *$\mathcal{D}$ a domain*
$\mathcal{T}$ a set of instantiated traces
$r \in \mathbb{N}$ the maximum number of logical operators in the output formula
$q \in \mathbb{N}$ the maximum number of quantifiers
$\sigma : \mathcal{T} \to \mathbb{R}$ a function called the score function

Output: *A formula $\psi \in \mathcal{L}_{FTL}$ such that ψ has at most r logical operators, and q quantifiers, and*
$$\sum_{\langle t, \mathcal{I}\rangle \in \mathcal{T}} \sigma(\langle t, \mathcal{I}\rangle)\,[\langle t, \mathcal{I}\rangle \models \psi]$$
is maximal

Even though the problem above is expressed as an optimization problem, various associated decision problems can be of interest. For example, a problem of interest is the one where ψ must be satisfied by all instantiated traces $\langle t, \mathcal{I}\rangle$ such that $\sigma(\langle t, \mathcal{I}\rangle) \geq 0$ and falsified by all other instantiated traces given in input.

Conjecture 1 *The decision problem associated to the $\mathcal{L}_{FTL}$ learning problem is NP-hard.*

Various authors tried to settle the complexity of problems related to ours, without always succeeding, even when dealing with simpler languages. Notable works include [6], where the authors show interest in the problem of learning various fragments of LTL. Even though the learning problems associated to several fragments were shown to be NP-complete, the complexity of the problem associated to the whole language is still open.

Membership in PSPACE Given an environment e, a trace t, and a formula $\varphi \in \mathcal{L}_{\text{TL}}$, checking that $t, e \models \varphi$ can be done in space polynomial in $|t|$, $|e|$ and $|\varphi|$ (ex. [7]). The model-checking of $\psi \in \mathcal{L}_{\text{FTL}}$ against some $\langle t, \mathcal{I}\rangle$ can be done by enumerating all relevant environments $e \in O^q$, and checking that $t, e \models \varphi$, where φ is the quantifier-free part of ψ. As a consequence, the $\mathcal{L}_{\text{FTL}}$ learning problem is in PSPACE. Even though this shows membership, the potential PSPACE-hardness of our problem is still an open problem.

Score function The choice of the score function allows us to express preferences on which traces are the most important to capture in the output formula, and which traces are the most important to avoid. In the rest of this article, we will say that an instantiated trace $\langle t, \mathcal{I}\rangle$ is *positive* iff $\sigma(\langle t, \mathcal{I}\rangle) \geq 0$. Otherwise, the instantiated trace is said to be *negative*.

4 Planning problem preprocessing

We present in this section the transformations we bring to the PDDL planning problem before it is passed to our algorithm for learning $\mathcal{L}_{\text{FTL}}$ formulas. As our algorithm is based on a compilation of the $\mathcal{L}_{\text{FTL}}$ learning problem into MaxSAT, reducing the size of the compiled form is crucial for it to run in reasonable time.

Predicate splitting Each predicate is split into several predicates of size 2, in order to curb the number of fluents while conserving the links between pairs of objects. This allows us to synthesize formulas containing predicates of high arity, while keeping the number of quantifiers of the formula low.

Concretely, a predicate of the form $p(x, y, z)$ will be split into newly-created predicates $p_{12}(x, y)$, $p_{13}(x, z)$, and $p_{23}(y, z)$. Notice that mathematically, predicate splitting leads to significantly fewer fluents than if the task was to be grounded as is: for a predicate of arity $n \geq 2$, to be grounded with instance $\mathcal{I}$, there are $O(n^2|O|^2)$ associated fluents, while there would be $O(|O|^n)$ if the predicate was not split.

Even though the planning model thus obtained is less rich than the original one, we argue that predicate splitting allows us to learn formulas that would be otherwise out of computational reach of our procedure.

Goal predicates The language $\mathcal{L}_{\text{FTL}}$ naturally allows us to reason on the initial state. However, in its current state, it does not allow reasoning on the goal conditions, which depend on the instance and are trace-agnostic.

We fix this issue by introducing *goal predicates*. For every predicate $p \in \mathcal{P}$, we introduce the predicate p^G. Then, for each instance $\mathcal{I}$, we introduce the *latent state* $s_{\mathcal{I}}$, which is intuitively a set of fluents that are true in every state of every trace associated to $\mathcal{I}$.

For every fluent $p(o_1, \ldots, o_{ar(p)})$ of the goal state G of $\mathcal{I}$, we add the fluent $p^G(o_1, \ldots, o_{ar(p)})$ to $s_{\mathcal{I}}$.

For readability reasons, rather than writing $p^G(o_1, \ldots, o_{ar(p)})$, we will often denote this using a (fictitious) modality named Goal. We thus write $\mathsf{Goal}\left(p(o_1, \ldots, o_{ar(p)})\right)$, as this is a clearer way to indicate that $p(o_1, \ldots, o_{ar(p)})$ is true in any goal state.

Equality predicates In addition to goal predicates, we also add to the latent state *equality predicates*. These are simply predicates of the form $=_\tau\ (x, y)$, where τ is a type. During the preprocessing of instantiated trace $\langle t, \mathcal{I} \rangle$, we add the fluents $=_{\tau(o)}\ (o, o)$ to the latent state, for every object $o \in O$.

5 Topology-based guiding

TL chains An interesting representation for formulas φ of $\mathcal{L}_{\text{TL}}$ is a representation as *TL chains*. They are the adaptation to our language of the notion of chain [13, 19], which is useful for representing formulas of modal or propositional logic.

A *TL chain* is a Directed Acyclic Graph (DAG) which has three types of nodes: logical connector nodes (represented as $\circ$ in the example of Figure 1), predicate nodes (represented as $\diamond$) and variable nodes (represented as $\square$). In order to represent a correct $\mathcal{L}_{\text{TL}}$ formula, logical connector nodes can only be children of logical connector nodes, predicate nodes children of logical connector nodes, and variable nodes children of predicate nodes. We also impose that every leaf is a variable node. In addition, to stay consistent with our language and the choices we made in Section 4, we only work with TL chains that are binary trees, whose inner nodes have exactly two children.

By assigning a symbol of the correct type (i.e., a logical connector, a predicate symbol or a variable) to each node, we end up with a representation of a $\mathcal{L}_{\text{TL}}$ formula. Figure 1 shows the representation as a TL chain of the formula $(q(v, u) \wedge r(z, y)) \,\mathsf{U}\, p(t, x)$.

For each connector node i of the TL chain, we will denote $succ_L(i)$ (resp. $succ_R(i)$) the left (resp. right) child of node i. It is guaranteed to exist, even though it might sometimes be a predicate node. In the case of connectors $\alpha \in \{\neg, \bigcirc, \overline{\bigcirc}, \Diamond, \overline{\Diamond}, \Box, \overline{\Box}\}$ that have arity 1, we will use the convention that the value of the right successor is ignored (and will not appear in the $\mathcal{L}_{\text{FTL}}$ formula that ensues), and the left successor will be the root of the formula under the operator α.

In order to alleviate the pressure on the MaxSAT solver, we impose the topology of the output quantifier-free formula before encoding the problem into a propositional formula. This idea was first introduced in [19], in an attempt to speed up the search for an LTL formula. In addition, we also fix the quantifiers of the formula before the encoding, as well as the types they quantify on. At the end of the day, all that is left to the MaxSAT solver is to "fill in the blanks" in the TL chains that it is given, so that the associated $\mathcal{L}_{\text{FTL}}$ formula fits the input as well as possible.

An interesting aspect of the constraints we impose on the form of the output formula, is that they force the algorithm to produce a wide diversity of formulas.

Quantifiers In the rest of the article, for practical reasons, we restrict ourselves to learning formulas of the form $\forall x_1 \cdots \forall x_k \exists x_{k+1} \cdots \exists x_b \varphi$, where φ is a formula of $\mathcal{L}_{\text{TL}}$, for which every argument of every predicate is a variable x_i. This choice makes some properties impossible to express, and it was made with the aim of limiting the number of MaxSAT intances to solve, as well as to curb the size of the MaxSAT encoding. However, as will be shown by our experimental evaluation, interesting formulas can still be learnt. Our encoding could be easily modified so that any sequence of quantifiers can be worked with, but we leave the experimental evaluation of such a program to future work.

6 Reduction to MaxSAT

6.1 Learning algorithm

Algorithm 1 summarizes the procedure that we use to learn $\mathcal{L}_{\text{FTL}}$ formulas out of our input. The subroutines work as follows: gen_TLchains(r) enumerates every TL chain having exactly r connectors. gen_quantifiers(q) enumerates sequences of quantifier symbols of size q, such that all universal quantifiers $\forall$ appear before existential quantifiers $\exists$. gen_types($\mathcal{D}$, q) enumerates every q-combination of types in the type tree $\mathcal{T}$ of $\mathcal{D}$. Finally, the main subroutine, find_formula($\mathcal{D}, \mathsf{T}, \rho, \{Q_i\}, \{\tau_i\}, \sigma$), encodes the problem of finding an $\mathcal{L}_{\text{FTL}}$ formula fitting the instantiated traces of T, with the constraints imposed by the TL chain ρ, the quantifiers $\{Q_i\}$, and the types $\{\tau_i\}$. find_formula then returns (one of) the best formula(s) it finds, or the token FAIL is none is found.

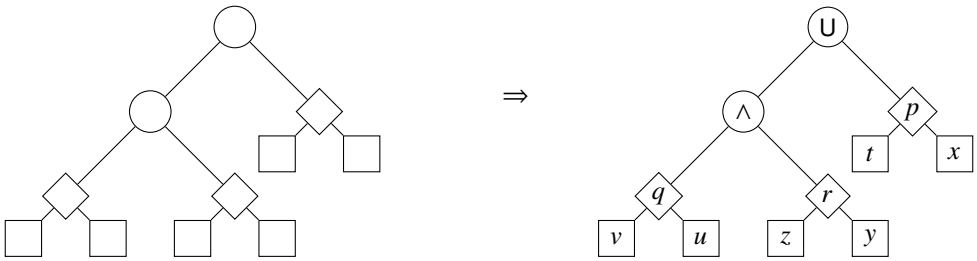

Figure 1: A TL chain example, as well as a possible assignation of symbols to its nodes. The TL chain on the right has been assigned symbols to every one of its nodes, and represents the formula $(q(v,u) \wedge r(z,y)) \,\mathsf{U}\, p(t,x)$

Algorithm 1: $\mathcal{L}_{\text{FTL}}$ learning

Input: Domain $\mathcal{D}$, traces T, parameters r, q, and function σ
Output: A set of $\mathcal{L}_{\text{FTL}}$ formulas
found_formulas := [ ]
for $\rho \in$ *gen_TLchains(r)* **do**
  for $Q_1, \ldots, Q_q \in$ *gen_quantifiers(q)* **do**
    for $\tau_1, \ldots, \tau_q \in$ *gen_types($\mathcal{D}$, q)* **do**
      $\psi \leftarrow$ find_formula($\mathcal{D}, \mathsf{T}, \rho, \{Q_i\}, \{\tau_i\}, \sigma$);
      if $\psi \neq$ *FAIL* **then**
        found_formulas.add(ψ);
    end
  end
end
return *found_formulas*

6.2 Preliminaries to the encoding

Notion of environment The constraints require us to reason on which combinations of objects may satisfy a formula or not. Indeed, as we work with a logic reminiscent of first-order logic (on finite domains), we are required to reason on various sets of objects at once. We now introduce the concept of environment, which allows us to work on assignations of (sets of) objects to variables of the formula.

Let us suppose that the formula we try to synthesize ranges over the variables $\mathcal{X} = (x_1, \ldots, x_q)$. In addition, let $\mathcal{I}$ be an instance, with objects $O = \{o_1, \ldots, o_{|O|}\}$. We call a *partial* environment any assignation of some of the variables $x_1, \ldots, x_q$ to an object of O. Let us denote $\mathsf{var}(e)$ the variables that are assigned an object within the partial environment e. When $\mathsf{var}(e) = \mathcal{X}$, we simply say that e is an environment.

We denote any (partial) environment $e = \{x_1 := o_{i_1}, \ldots, x_q := o_{i_q}\}$, where $i_1, \ldots, i_q \in [\![1, |O|]\!]$.

We also denote $p(x,y)[e]$ the grounding of an atom $p(x,y)$ by an environment e such that $x, y \in \mathsf{var}(e)$. If $e = \{x := o_1, y := o_2, \ldots\}$, then $p(x,y)[e] = p(o_1, o_2)$. By extension, the formula obtained when grounding each atom of φ with e is written $\varphi[e]$.

The main difficulty that rises when working with environments is that there are $|O|^q$ different environments. As O can be large, the number of environments being exponential in the number of quantifiers quickly makes the problem intractable, if no restriction is posed. This is why we ensure that each variable of the $\mathcal{L}_{\text{FTL}}$ formula to learn is assigned a type, so that not every environment has to be considered during search: the types are chosen before proceeding to the encoding.

Variables Our MaxSAT encoding is built on the set of variables that follows. When possible, we use the following conventions, as closely as possible: nodes of the FL-chain are denoted by i when they are logical connectors (represented by ◯ in Figure 1), by ℓ when they are predicate nodes (represented by ◊), and by v when they are first-order variable nodes (represented by □). A trace is denoted by t, and a position in this trace is denoted by k (i.e., the k-th state). Moreover, j is an index for a variable of the quantifiers, and p is a predicate.

This leads us to the following variables, as will be used in the MaxSAT encoding. Greek letters denote decision variables while latin characters are for "technical" variables.

- $y_i^{t,k}[e]$: In position k of trace t, with environment e, the formula rooted at node i is true.
- $\delta_{j,v}^{\ell}$: The v-th variable of predicate node ℓ is the variable of quantifier j.
- θ_{ℓ}^{p}: The predicate of node ℓ is p.
- λ_i^q: The logical connector at node i is q.
- s_t: Trace t is currently satisfied by the first order formula

"Exactly one" constraints In the encoding of a problem into SAT, some situations require that *at most one* variable, out of a set of variables, is true. There exist encodings that are more efficient than the naive one to define such "at most one" constraints: see for instance [10, 16] for a

survey on these encodings. Like many other solvers, the MaxSAT solver that we use proposes a built-in function for this. More specifically, it offers a built-in function for *exactly one* constraints, where exactly one variable of a set must be true in a model of the formula.

In the following, we will denote $\mathsf{ExactlyOne}_{s\in S}(v_s)$ the set of propositional constraints enforcing that at most one of the variables of $\{v_s \mid s \in S\}$ is true.

6.3 Core constraints

Some of the constraints below are adapted from [8, 19, 15], which are concerned with LTL. Our main contribution is the adaptation of the encoding to our language $\mathcal{L}_{\text{FTL}}$, which differs from LTL by its tighter links with PDDL planning models through first-order components.

In the following, we suppose that an empty TL chain ρ has been computed, and that the associated quantifiers and types have been decided. We will denote n its number of connector nodes, and m its number of predicate nodes. As a consequence, there are $2m$ variables nodes. As previously, the number of quantifiers is denoted q. The first $b \leq q$ quantifiers are universal, while the other are existential.

We also suppose that the types on which the quantifiers range, denoted $\tau_1, \ldots, \tau_q$, are already chosen. As a consequence, in this section, the set of relevant environments for instance $\mathcal{I}_t$ associated to trace t, denoted $E_{\mathcal{I}_t}$, only consists of environments of the form $\{x_u := o_u\}_{1\leq u\leq q}$ where, $\tau(o_u) = \tau_u$, for $u \in [\![1, q]\!]$.

Syntactic constraints This section describes the constraints that ensure that the formula is syntactically well-formed.

The following constraints ensure that every logical connector node has exactly one logical connector assigned, and every predicate node has exactly one predicate, respectively. Recall that Λ is the set of all logical operators.

$$\bigwedge_{i\leq n} \mathsf{ExactlyOne}_{c\in\Lambda}(\lambda_i^c) \tag{3}$$

$$\bigwedge_{\ell\leq m} \mathsf{ExactlyOne}_{p\in\mathcal{P}}(\theta_\ell^p) \tag{4}$$

Finally, the following constraints force each argument of each predicate to be bound to a variable on which the formula quantifies.

$$\bigwedge_{\ell\leq m}\bigwedge_{s\in\{1,2\}} \mathsf{ExactlyOne}_{j\leq b}(\delta_{j,s}^\ell) \tag{5}$$

Semantic constraints These constraints ensure that the formula found by the solver is consistent with the traces. It mimics the model-checking algorithm for modal logic.

The following clauses ensure that the formula ψ that is synthesized is consistent with the traces of T. This is made in accordance with the environments imposed by the quantifier, which are iterated upon. The variable s_t is true iff for every required environment e, $\varphi\,[e]$ is satisfied by t (where $\varphi\,[e]$ is the evaluation of formula φ in environment e, and φ is the quantifer-free part of the formula we synthesize). Thus, for every trace $t \in \mathsf{T}$, we add the following:

$$s_t \Leftrightarrow \left(\bigwedge_{\substack{o_1\in O_1\\ \cdots\\ o_k\in O_k}} \bigvee_{\substack{o_{k+1}\in O_{k+1}\\ \cdots\\ o_q\in O_q}} y_1^{t,1}[\{x_u := o_u\}_{1\leq u\leq q}] \right) \tag{6}$$

The following constraints ensure that formulas that consist of a single literal (i.e., a positive or negative fluent) are consistent with the y variables, that give the truth value of a trace at a certain position in the trace, at each node of the TL chain.

Such constraints appear once for every trace $t \in \mathsf{T}$, for every position $k \leq |t|$ of this trace, for every predicate node $\ell \leq m$ and every predicate $p \in \mathcal{P}$, for every pair of quantifiers (positions) $j_1, j_2 \leq q$, and for each relevant environment $e \in E_{\mathcal{I}_t}$.

$$\theta_\ell^p \wedge \delta_{j_1,1}^\ell \wedge \delta_{j_2,2}^\ell \Rightarrow \begin{cases} y_\ell^{t,k}[e] & \text{if } t[k] \models p(x_{j_1}, x_{j_2})[e] \\ \neg y_\ell^{t,k}[e] & \text{otherwise} \end{cases} \tag{7}$$

The constraints sketched in equations (8) to (11) appear once for each connector node $i \leq n$ of the formula, each position $k \leq |t|$ of each trace $t \in \mathsf{T}$, and for each environment $e \in E_{\mathcal{I}_t}$. They ensure that the logical operators are correctly interpreted.

In the case where the logical connector at node i is a negation $\neg$, we have:

$$\lambda_i^\neg \Rightarrow \left(y_i^{t,k}[e] \Leftrightarrow \neg y_{succ_L(i)}^{t,k}[e] \right) \tag{8}$$

In the case of $\Delta \in \{\wedge, \vee, \Rightarrow\}$:

$$\lambda_i^\Delta \Rightarrow \left(y_i^{t,k}[e] \Leftrightarrow \left(y_{succ_L(i)}^{t,k}[e] \;\Delta\; y_{succ_R(i)}^{t,k}[e] \right) \right) \tag{9}$$

In the case of the next operator $\bigcirc$, we have the following:

$$\lambda_i^\bigcirc \Rightarrow \left(y_i^{t,k}[e] \Leftrightarrow y_{succ_L(i)}^{t,k+1}[e] \right) \tag{10}$$

with the convention that $y_{succ_L(i)}^{t,|t|+1}[e]$ is replaced by $\bot$ during the encoding itself.

In the case of the finally operator $\Diamond$:

$$\lambda_i^\Diamond \Rightarrow \left(y_i^{t,k}[e] \Leftrightarrow \bigvee_{\substack{k'\\ k\leq k'\leq |t|}} y_{succ_L(i)}^{t,k'}[e] \right) \tag{11}$$

The case of the temporal operators $\Box$, $\overline{\bigcirc}$, $\overline{\Diamond}$, $\overline{\Box}$ and U can be encoded in a way that is similar to the constraints above.

Well-formed fluents constraints The following constraints ensure that, in the output formula ψ, there is a consistency between the types of the variables and the arguments of predicates are assigned to. Otherwise said, when a variable x of type τ is chosen to be the v-th argument of a predicate p that occurs in ψ, we require that $\tau = \tau_p(v)$. This can be done through the following constraints:

$$\bigwedge_{j \leq q} \bigwedge_{\ell \leq m} \bigwedge_{p \in \mathcal{P}} \bigwedge_{j \leq q} \bigwedge_{\substack{v \leq 2 \\ \tau_p(v) \neq \tau_j}} \neg\theta^p_\ell \vee \neg\delta^\ell_{j,v} \tag{12}$$

Weights for the MaxSAT solver Recall that we wish to find a formula ψ that maximizes the following function:

$$\sum_{\langle t, \mathcal{I} \rangle \in \mathsf{T}} \sigma(\langle t, \mathcal{I} \rangle) \, [\langle t, \mathcal{I} \rangle \models \psi]$$

The objective of the MaxSAT solver is to minimize the total weight of the falsified soft clauses. As such, for each instantiated trace $\langle t, \mathcal{I} \rangle$, we add the clause s_t, with weight $\sigma(\langle t, \mathcal{I} \rangle)$. This penalizes formulas that falsify traces with a positive score, while rewarding formulas that falsify traces with a negative score.

Pruning non-discriminatory formulas With a given configuration of TL chain, quantifiers and types, it is not guaranteed that there exists a formula ψ that captures (some of) the positive traces while falsifying (some of) the negative traces. Without further precautions, our algorithm can output formulas that are true on all traces, or false on all traces. These formulas are often tautologies or unsatisfiable, and still have a non-zero score as they completely capture the positive or negative traces.

The following clauses ensure that at least one positive trace and one negative trace are captured:

$$\bigvee_{\substack{t \in \mathsf{T} \\ \sigma(t) \geq 0}} s_t \wedge \bigvee_{\substack{t \in \mathsf{T} \\ \sigma(t) < 0}} \neg s_t \tag{13}$$

6.4 Formula quality enhancement

The constraints presented in this section filter the solutions so that less interesting formulas, or formulas that could be computed by a run of our algorithm with smaller parameters, are barred from being output.

Syntactic redundancies prevention These constraints prevent idempotent and involutive modalities and operators from being chained in the output formula. These include the negation $\neg$, as well as the temporal operators $\Diamond$ (for which $\Diamond\Diamond\varphi \equiv \Diamond\varphi$) and $\Box$ (which is, likewise, idempotent). In order to prevent the operator $\alpha \in \{\neg, \Diamond, \overline{\Diamond}, \Box, \overline{\Box}\}$ from appearing in a node of the TL chain and its left-successor, we add the following constraints, when possible (i.e. when both i and $succ_L(i)$ are defined):

$$\neg\lambda^\alpha_i \vee \neg\lambda^\alpha_{succ_L(i)} \tag{14}$$

In addition, we prevent redundancies of the form $p(x, y) \, \Delta \, p(x, y)$, where $\Delta \in \{\wedge, \vee, \mathsf{U}, \Rightarrow\}$ is a binary operator. In every case, there exists a smaller (sub-)formula that can be found and that expresses the same thing, without the redundant atom. For each connector node i which has two predicate nodes as children, denoted as $\ell_l := succ_L(i)$ and $\ell_r := succ_R(i)$, we add the following clauses:

$$\bigwedge_{p \in \mathcal{P}} \bigwedge_{\substack{j_1, j_2 \leq q \\ j_2 \neq j_1}} \neg \left(\theta^p_{\ell_l} \wedge \delta^1_{j_1, \ell_l} \wedge \delta^2_{j_2, \ell_l} \wedge \theta^p_{\ell_r} \wedge \delta^1_{j_1, \ell_r} \wedge \delta^2_{j_2, \ell_r} \right)$$

The constraints above actually prevent two fluents that are adjacent (i.e. that are connected by a binary operator) to be equal, regardless of the actual operator that links them.

Variable visibility As the size of the encoding is exponential in the number q of expected quantifiers, we wish to ensure that every variable that we quantify upon in the output formula ψ also appears in an atom of ψ. Otherwise, an equivalent formula could be found by running the algorithm with fewer quantifiers. This is why we force each variable to appear at least once in some atom. For space reasons, we skip the presentation of the constraints.

7 Experiments

This section presents the experiments we ran in order to assess the performances of our method. The implementation was done in Python 3.10, using the MaxSAT solver Z3 [5]. Experiments were conducted on a machine running Rocky Linux 8.5, powered by an Intel Xeon E5-2667 v3 processor, with a 24-hours cutoff and using at most 16GB of memory per run. The code of our implementation can be found online[1]. The repository also includes tools to evaluate formulas against a dataset and a planning model.

7.1 Performances of the learning algorithm

To assess the performances of our algorithms, we considered 6 domains from various editions of the International Planning Competition (IPC), some of which are described in Section 7.2. For each of these domains, we generated 10 instances that model problems with similar goals. We then used 5 planners from the IPC to generate plans for each instance, that our algorithm converted to traces. On average, plans had 10.2 operators, but some instances include plans of size up to 23.

[1] https://github.com/arnaudlequen/LearningEngine

Table 1: Average amortized time in seconds (s) required to learn a single formula from our dataset, depending on the quantifiers imposed and the maximum number of logical operators. Values in the table represent the total running time of the algorithm, divided by the total number of formulas found. This represents the average time between two formula outputs. Entries labeled by a dot (.) represent training instances that reached the time cutoff.

$\lvert\varphi\rvert$	Quantifiers				
	$\forall$	$\exists$	$\forall\forall$	$\forall\exists$	$\exists\exists$
2	4.2	4.3	36.5	41.0	34.8
3	9.8	8.7	53.8	56.9	48.7
4	22.2	20.5	.	.	.

We built our training instances by selecting 3 planning instances of each domain, and the associated traces for each planner - for a total of 15 instantiated traces per training instance. We then created the tasks of finding a formula recognizing the behavior of each planner, and ran our algorithm on each such task. The remaining instances were used for our test set.

The aim of these experiments was to test our algorithm in a setting where it would struggle. Indeed, planners often output plans that are similar to each others, as they are close to optimal, and do not exhibit particularly distinctive behaviors. This makes finding a formula that perfectly recognizes the behavior of a planner hard. Nonetheless, our algorithm still managed to output formulas that *imperfectly* recognize a planner's behaviour.

The average amortized times to synthesize a formula on our dataset are summarized in Table 1. In our tests, a formula was found in 87.5% of the solving attempts of our algorithm. Since 73.2% of the running time is dedicated to the encoding, we ask the MaxSAT solver to output multiple solutions for each encoding.

7.2 Examples of learnt formulas

In this section, we present some formulas that have been learnt by our algorithm. We considered three domains among the ones used in our data set. For the first two domains, since the plans found by the 5 planners were all very similar, we handcrafted various agents that tackle the problem in a distinctive way. We kept off-the-shelf planners for the last domain. We then built sets of plans in a similar way as in the previous section.

Spanner Instances of the Spanner domain involve an operator that has to go from a shed to a gate to tighten some nuts, passing through a sequence of locations where single-use spanners can be picked up. Once a location is left, it can not be returned to. Thus, collecting enough spanners before reaching the gate is seminal for solving the problem.

We developed three different behaviours for this domain. Agent ALL picks every possible spanner on its way to the gate, while agent SME picks exactly as many spanners as are needed to tighten the nuts at the gate. Agent SGL takes a single spanner and rushes to the gate, and can then only tighten one nut.

Among the formulas that perfectly recognize plans belonging to agent ALL, we have the following:

$$\forall x \in \text{Spanner}.\, \exists y \in \text{Operator}.\, \Diamond \Box\, \text{carrying}(y, x) \tag{15}$$

$$\forall x \in \text{Spanner}.\, \exists y \in \text{Location}.\, \text{at}(x, y) \wedge \Diamond \neg \text{at}(x, y) \tag{16}$$

Formula (15) expresses that every spanner will be picked up by the (only) operator and carried for the rest of the plan, and Formula (16) expresses that every spanner will be moved from its initial position at some point.

Even though we also managed to learn a formula that perfectly discriminates agent SGL from the others, we failed to learn a formula completely capturing SME's behaviour.

When searching formulas with a single variable, we split the predicates so that the maximum arity of a fluent is 1. Our algorithm output the following formulas, which were learnt in a few seconds, and completely characterize the behavior of agent ALL:

$$\forall x \in \text{Spanner}.\, \Diamond\, \text{carrying}_2(x) \tag{17}$$

$$\forall x \in \text{Spanner}.\, \text{useable}(x)\, \mathsf{U}\, \text{carrying}_2(x) \tag{18}$$

Predicate splitting allows us to obtain a concise formula, where the focus is clear (every spanner x is eventually carried), as uninformative elements (who carries the spanner) are omitted.

Childsnacks Domain Childsnacks, as introduced in Section 2, consists in making sandwiches and serving them to a group of children, some of whom are allergic to gluten. Sandwiches can only be prepared in the kitchen, and then have to be put on trays, which is the only way they can be brought to the children.

We designed three different agents that solve Childsnacks instances. Agents NGF and NGL compute solution plans of minimal size, and differ in that agent NGF makes sandwiches with *no gluten first*, and agent NGL makes sandwiches with *no gluten last*. Both agents make all sandwiches, put them on a tray, then serve the children. Agent GS greedily serves children: as soon as a sandwich is made, it is put on a tray and brought to a child. It also prioritizes gluten-free sandwiches.

In every instance, 2 trays are initially in the kitchen. The only difference between instances is in the number of children to serve, the smallest having 2. This is, however, enough to learn a wide variety of formulas that perfectly recognize the behavior of agent GS (among others) on our

test set. Such formulas include, for instance, the following:

$$\forall x \in \text{Kitchen}.\, \exists y \in \text{Tray}.\, \Diamond(\text{at}(y,x) \wedge \overline{\Diamond}\neg\text{at}(y,x)) \quad (19)$$

$$\forall x \in \text{Kitchen}.\, \exists y \in \text{Tray}.\Diamond(\neg\text{at}(y,x) \wedge \bigcirc\text{at}(y,x)) \quad (20)$$

Formula (19) expresses that Agent GS eventually comes back to the kitchen with some tray y, even though the tray was brought out of the kitchen at some point in the past. Formula (20) expresses the same idea, but pinpoints the moment when a tray is brought back to the kitchen.

Both formulas manage to perfectly capture our test set, but do not perfectly capture our *training* set. This is due to the fact that the smallest instance of our training set contains as many children as there are trays, and thus, no tray has to be brought back to the kitchen. Our use of a reduction to MaxSAT allows us to be resilient to this kind of edge cases, and the formulas that are learnt are satisfactory despite not perfectly fitting the training set.

Even though our algorithm managed to learn concise formulas that perfectly capture agent NGL's behaviour, it failed to find in reasonable time a formula that discriminates agent NGF's behaviour with reasonable accuracy.

Rovers Domain Rovers simulates a planetary exploration mission, where a fleet of mobile rovers has to navigate between various waypoints on a planet to collect data or samples, and to transmit the data back to a lander. The rovers have instruments, which have to be calibrated before they can collect data.

The set of instances we designed all required a single rover to collect two kinds of samples, and to take a picture of an object. The only differences lie in the topology of the planet and the positions of objectives.

For this domain, we resorted to five different planners from the IPC 2018 and 2023 to generate plans, namely BFWS, Odin, TFTM-ArgMax, LAMA and DecStar. Our algorithm managed to learn the following formula (among others), which recognizes the plans of planner BFWS with 93.3% accuracy. The only traces that have been wrongfully recognized are traces of TFTM-ArgMax.

$$\begin{aligned}&\forall x \in \text{Rover}.\forall y \in \text{Waypoint}.\\ &\quad \mathsf{Goal}(\text{communicated\_rock\_data}(y))\\ &\quad \Rightarrow \bigcirc\text{have\_rock\_analysis}(x,y) \quad (21)\end{aligned}$$

$$\forall x \in \text{Rover}.\exists y \in \text{Store}.\text{store\_of}(y,x) \wedge \bigcirc\text{full}(y) \quad (22)$$

The formulas above express the fact that BFWS consistently collects samples as fast as possible, and starts by collecting the rock samples. By opposition, other planners tend to start by calibrating the instruments required to take the picture, before proceeding to explore the planet.

8 Conclusion

In this paper, we have presented a method to learn temporal logic formulas that recognize agents based on examples of their behaviours. We showed that such formulas can be learned using an algorithm that boils down to a reduction to MaxSAT, and that very few examples are sometimes enough to perfectly capture the behaviour of an agent on instances that can differ from the ones used in the training set. This justifies the cost of resorting to a first-order language, which generalizes to new instances.

In future works, we wish to tailor our algorithm and our datasets so that they can generate domain-specific control knowledge. Some other authors [2] have expressed search control knowledge in a language similar to ours, with the aim of guiding the search of a planner designed to use such knowledge. While this knowledge must be written by a human operator, previous works show that it could also be generated automatically [4].

Acknowledgements The author would like to thank Martin C. Cooper, for his insightful suggestions and his careful proofreading of this manuscript; and the reviewers of this paper, for their numerous helpful comments which helped us improve this article.

References

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

[2] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1):123–191, 2000.

[3] Alberto Camacho and Sheila A. McIlraith. Learning interpretable models expressed in linear temporal logic. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):621–630, May 2021.

[4] Tomás de la Rosa and Sheila McIlraith. Learning domain control knowledge for TLPlan and beyond. In *Proceedings of the ICAPS-11 Workshop on Planning and Learning (PAL)*, 2011.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[6] Nathanaël Fijalkow and Guillaume Lagarde. The complexity of learning linear temporal formulas from examples. In Jane Chandlee, Rémi Eyraud, Jeff Heinz,

Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 237–250. PMLR, 23–27 Aug 2021.

[7] Valeria Fionda and Gianluigi Greco. The complexity of LTL on finite traces: Hard and easy fragments. In *AAAI Conference on Artificial Intelligence*, 2016.

[8] Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. Learning linear temporal properties from noisy data: a MaxSAT-based approach. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*, pages 74–90. Springer, 2021.

[9] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[10] Steffen Hölldobler and Van-Hau Nguyen. An efficient encoding of the at-most-one constraint. *Technical Report. Technische Universitäat Dresden*, 2013.

[11] Rostislav Horčík and Daniel Fišer. Endomorphisms of lifted planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 174–183, 2021.

[12] Joseph Kim, Christian Muise, Ankit Jayesh Shah, Shubham Agarwal, and Julie A Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *International Joint Conference on Artificial Intelligence*, 2019.

[13] Donald Ervin Knuth. *The Art of Computer Programming*, volume 4A. Addison Wesley, 2020.

[14] Nicolas Markey. Temporal logic with past is exponentially more succinct. *Bull. EATCS*, 79:122–128, 2003.

[15] Daniel Neider and Ivan Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2018.

[16] Van-Hau Nguyen and Son T. Mai. A new method to encode the at-most-one constraint into SAT. In *Proceedings of the 6th International Symposium on Information and Communication Technology*, SoICT '15, page 46–53, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Amir Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57, 1977.

[18] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. Scalable anytime algorithms for learning fragments of linear temporal logic. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–280, Cham, 2022. Springer International Publishing.

[19] Heinz Riener. Exact synthesis of LTL properties from traces. In *2019 Forum for Specification and Design Languages*, pages 1–6. IEEE, 2019.

[20] Maayan Shvo, Andrew C Li, Rodrigo Toro Icarte, and Sheila A McIlraith. Interpretable sequence classification via discrete optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9647–9656, 2021.

[21] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243. Springer, 2022.

[22] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. A C-LSTM neural network for text classification. *arXiv*, 11 2015.