Learning Linear Temporal Properties for Autonomous Robotic Systems

Enrico Ghiorzi[1], Michele Colledanchise[1], Gianluca Piquet[2], Stefano Bernagozzi[1,2], Armando Tacchella[2],[2], and Lorenzo Natale[1]

***Abstract*—The problem of passive learning of linear temporal logic formulae consists in finding the best explanation for how two sets of execution traces differ, in the form of the shortest formula that separates the two sets. We approach the problem by implementing an exhaustive search algorithm optimized for execution speed. We apply it to the use-case of a robot moving in an unstructured environment as its battery discharges, both in simulation and in the real world. The results of our experiments confirm that our approach can learn temporal formulas explaining task failures in a case of practical interest.**

***Index Terms*—Formal Methods in Robotics and Automation, Autonomous Agents, Failure Detection and Recovery**

I. Introduction

AUTONOMOUS robotic systems must accomplish tasks outside strictly controlled environments and minimizing the need for human intervention. In this context and in spite of the efforts made by system developers, not all reasons leading to a failure can be foreseen and accounted for. Understanding the root issue of a task failure is difficult, especially if no human operator is observing the task execution and the cause of the failure has to be figured out from the execution logs. Thus, it is useful to employ learning techniques that, by examining execution logs of both successful and unsuccessful executions, can learn the reason why the system has failed. The problem is known as *passive learning of temporal properties*.

We approach the learning problem by employing linear temporal logic (LTL) formulae as learning target, assuming that system executions produce Boolean traces labeled as either successful or unsuccessful. The goal is to find a LTL formula that is satisfied by all the traces produced by successful executions, and by none of the traces produced by unsuccessful ones. Thus, the formula encodes the reason why some executions are successful and some are not. The level of generality and clarity of the formula is linked to its size: the shorter the formula, the more general and clear it is as an explanation of the issue the system incurs into. Thus, our goal is to find the shortest such formula.

We recall some background in section III and stating the problem formulation in section IV. Then we develop a learning algorithm based on exhaustive search, described in section V and optimized with various techniques detailed in sections V-A to V-C. In section VI we evaluate the theoretical guarantees of the algorithm. In section VII we discuss the concrete implementation of the algorithm. In section VIII we compare the performances of our approach against those of the reference solvers proposed in [1]–[4]. Finally, in section IX we apply the learning algorithm to the concrete use-case of an autonomous robot trying to grasp a token and take it to destination, while preventing its battery from draining and possibly replenishing it at a charging station. We implement such experiment as two distinct simulations (sections IX-A and IX-B) and in the real world (section IX-C).

II. Related work

The problem of learning temporal properties has been studied extensively, with various kinds of underlying logic, semantics, and learning strategy. Some basic theory of the complexity of learning LTL formulae from examples is presented in [5] together with the proof that the problem is NP-hard (at least for some choice of logical and temporal operators).

In the literature, a common approach to learning formulae from examples is to search through some subset of formulae matching templates while optimizing some metric [6]–[13]. As we want to make no assumption about the shape of the solution, we cannot use this approach.

Another common approach is to reduce the learning problem to a satisfiability problem, searching in order of increasing size of the formulae [1]–[3], [14]–[17]. In [1], the problem is reduced to a Boolean satisfiability problem and then a solution in the form of a syntax directed acyclic diagram (DAG) is found with the assistance of a SAT-solver. The overhead this approach introduces makes it unsuitable to solve large samples arising from real-world data. The work in [1] has been extended by searching for approximate solutions to noisy data via a Max-SAT solver and employing finite-trace semantics (which we adopt too) [2], [3], while [15] extends it to the property specification language temporal logic (PSL). The work of [1], [2] has also been extended to signal temporal logic (STL), using SMT solvers instead of SAT solvers [3], [16]. Listing exhaustively all DAGs and then solving the Boolean satisfiability problem for each DAG, [14] refines the work in [1], obtaining a speed-up compared to the original method. Finally,

Please cite this paper as: E. Ghiorzi, M. Colledanchise, G. Piquet, A. Tacchella, and L. Natale, "Learning linear temporal properties for autonomous robotic systems," IEEE Robotics and Automation Letters, Vol. 8, no. 5, 2023, doi:10.1109/LRA.2023.3263368.

This work received funding from the European Union's Horizon Europe research and innovation program under GrantAgreement No. 101070227 (CONVINCE).

[1]Enrico Ghiorzi, Michele Colledanchise, Stefano Bernagozzi and Lorenzo Natale are with Istituto Italiano di Tecnologia, Genoa, Italy. lorenzo.natale@iit.it

[2]Gianluca Piquet, Stefano Bernagozzi and Armando Tacchella are with DIBRIS, Università degli Studi di Genova, Genoa, Italy. armando.tacchella@unige.it

[17] uses a SAT-solver to learn LTL formulae encoded as alternating finite automata as opposed to the syntax DAGs used by [1]. These learning algorithms have been applied to robotics. For example, [18]–[20] use the learning algorithm from [1], [2].

Genetic algorithms perform well but have to allow for some misclassification and cannot guarantee minimality [21].

Finally, there are exhaustive enumeration algorithms [4], [22]. SySLite,[1] a state-of-the-art solver based on an enumerative algorithm, encodes the problem as a bit-vector function synthesis problem [4] to be solved by CVC4, a Syntax-Guided Synthesis engine, via an optimized exhaustive search.

In this work we present an algorithm based on exhaustive search and featuring domain-specific optimizations, with the goal of finding complex solutions on large samples with many variables. We verify that, for practical applications in robotics, our approach is more efficient than the already available tools. To obtain such performances, though, we have to limit our search to LTL formulae, instead of learning STL formulae as other works do [6], [7], [9]–[11], [13], [16], [21]–[24].

III. Background

Linear Temporal Logic (LTL) is an extension of Boolean logic with temporal operators. LTL describes not just the state of a system characterized by Boolean variables, but also its evolution through time. For example, it can express concepts such as "P will be true in the next time step", "P is always true", "P will eventually be true", or "P is going to be true until Q becomes true", where P and Q are Boolean propositions.

A. Syntax

We give the syntax for LTL formulae via an inductive definition. Let Var be a non-empty finite set of *Boolean variables* (which we usually denote as $p, q, r \dots$).

Definition 1. *A* well-formed LTL formula *over* Var *is defined inductively as*

$$p \mid \neg\phi \mid \phi \vee \psi \mid \mathrm{X}\,\phi \mid \phi\,\mathrm{U}\,\psi$$

where $p \in \mathrm{Var}$ *and* ϕ, ψ *are well-formed LTL formulae. The temporal operators extending Boolean logic are the* neXt *operator* X *and the* Until *operator* U. *The* size *of a formula is given by the number of symbols it contains (not counting the parenthesis).*

Derived operators can be defined from the base ones:

$$\begin{aligned} \top &= p \vee \neg p \\ \bot &= \neg\top \\ \phi \wedge \psi &= \neg((\neg\phi) \vee (\neg\psi)) \\ \phi \implies \psi &= \psi \vee (\neg\phi) \\ \mathrm{F}\,\phi &= \top\,\mathrm{U}\,\phi \\ \mathrm{G}\,\phi &= \neg(\mathrm{F}(\neg\phi)) \\ \phi\,\mathrm{R}\,\psi &= \neg((\neg\phi)\,\mathrm{U}\,(\neg\psi)) \end{aligned}$$

Aside from the usual derived logical operators, we have the derived temporal operators *Finally* (F), *always* or *Globally* (G), and *Release* (R). Notice that $\top$ is defined by a choice of variable $p \in \mathrm{Var}$, and thus uses the hypothesis that Var is non-empty. Moreover, the definition of $\top$ assumes the use of the excluded middle, which is fine as we are working in the context of classical logic.

In the implementation of the algorithm described in the rest of the section, we use formulae including some derived operators in addition to the primitive ones, as some derived operators are particularly useful in practice. Specifically, we include G, F, and $\wedge$.

B. Semantics

While Boolean formulae are interpreted over Boolean evaluations, LTL formulae are interpreted over infinite traces, i.e., infinite sequences of Boolean evaluations (this semantics is used in [1]). We consider evaluation of LTL formulas on finite traces instead [25], but we omit any qualification (e.g., $\mathrm{LTL_f}$ for LTL over finite traces) as in the context of this paper there is no potential for confusion.

Definition 2. *A* Boolean valuation *is a mapping* $\mathrm{Var} \to \mathrm{Bool}$*, and a* trace $t = (t_i)_{i=0,\dots,l}$ *is a finite sequence of Boolean valuations. An LTL formula* ϕ *is interpreted over a trace* t *at a given moment in time* i *where* $0 \le i \le l$*. We inductively define that* t satisfies ϕ *at time* i*, and we write* $t, i \vDash \phi$*, as follows:*

- $t, i \vDash p$ *iff* $t_i(p)$
- $t, i \vDash \neg\phi$ *iff* $t, i \nvDash \phi$
- $t, i \vDash \phi \vee \psi$ *iff* $t, i \vDash \phi$ *or* $t, i \vDash \psi$
- $t, i \vDash \mathrm{X}\,\phi$ *iff* $i < l$ *and* $t, i+1 \vDash \phi$
- $t, i \vDash \phi\,\mathrm{U}\,\psi$ *iff* $\exists_{i \le j \le l}$ *s.t.* $t, j \vDash \psi$ *and* $\forall_{i \le k < j}\ t, k \vDash \phi$

If $t, 0 \vDash \phi$*, we say that* t satisfies ϕ*, and we write* $t \vDash \phi$*.*

Intuitively, we interpret an atomic Boolean variable by the corresponding Boolean value provided by the 0-th Boolean valuation in the trace (or, leveraging the temporal paradigm, we can also say "provided by the trace at time 0"), and the logical operators are interpreted as usual. The temporal operator X moves the time forward onto the next time step. Finally, the U operator states that, at some point, its right-hand-side argument must become true, and that until that moment its left-hand-side argument holds true.

IV. Problem formulation

Consider an autonomous system repeatedly executing a task. Assuming that truth assignments to Boolean variables are sufficient to characterize the state of the system, during each execution the system produces a trace. Traces corresponding to successful executions are labeled as "positive," whereas traces corresponding to unsuccessful executions are labeled as "negative." This leads to the following definition.

Definition 3. *A* sample $S = (P, N)$ *is given by two sets of traces, the* positive *traces* P *and the* negative *traces* N*. A formula* ϕ *is* consistent *with a sample* $S = (P, N)$ *if*

- $t \vDash \phi$ *for all* $t \in P$*, and*
- $t \nvDash \phi$ *for all* $t \in N$*.*

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

In other words, a formula is consistent with a sample if it satisfies all of its positive traces and none of the negative ones.

Intuitively, a formula that is consistent with a sample provides an explanation of what causes some traces in the sample to be positive and others to be negative. From a philosophical point of view, the Occam's Razor claims that the best explanation of a phenomenon (that is, the one that is most likely to be correct and extend to further examples) is the shortest one. Moreover, "small formulas are easier for humans to comprehend than large ones" [1]. Thus, the problem of *passive learning of LTL formulae* consists in finding a LTL formula *of minimal size* consistent with the sample [1].

Assuming that P and N are disjoint, the problem of passive learning always admits a solution. Indeed, all we need to show is the existence of a formula consistent with the sample. If that is the case, a solution of minimal size exists (although not necessarily unique) by the well-ordering of the natural numbers (the property that any non-empty set of natural numbers has a least element). We prove the existence of such a formula with the following lemma.

Lemma 1. *If P and N are disjoint sets of traces, then there exists a formula satisfying the sample $S = (P, N)$.*

Proof. Consider traces $t \in P$ and $s \in N$. Since, by hypothesis, $t \neq s$, there exist a time instant i and an atomic variable p such that $t_i(p) \neq s_i(p)$. Let $\phi_{t,s}$ be the formula $\mathrm{X}^i\, p$ if $t_i(p)$ is true, and $\neg\, \mathrm{X}^i\, p$ otherwise (where X^i is the next operator repeated i-many times) so that $t \vDash \phi_{t,s}$ and $s \nvDash \phi_{t,s}$. Then the formula

$$\phi \coloneqq \bigvee_{t \in P} \bigwedge_{s \in N} \phi_{t,s}$$

is such that $t \vDash \phi$ for every $t \in P$ and, $s \nvDash \phi$ for every $s \in N$, i.e., it satisfies the sample S. □

In spite of the theoretical significance, the formula produced by lemma 1 is of little practical use, since it overfits the sample and it is generally way too long to be understood intuitively.

V. Proposed solution

We propose that a practical approach to the learning problem is simply by exhaustive search. Indeed, it is possible to recursively generate all LTL formulae of a given size by starting the recursive construction from the atomic propositions and then combining unary and binary operators with formulae of (suitably chosen) smaller size. Storing all the generated formulae is memory-expensive, so we develop an algorithm, detailed in section V-A, to produce them in small batches. We then generate all LTL formulae in order of increasing size, and test each of them against the sample until one is found to be consistent with it. Furthermore, it is possible to make the exhaustive search more efficient through some optimizations, which we discuss in sections V-B and V-C.

A. Generating formulae via skeleton trees

As a matter of fact, the memory required to store all of the generated formulae quickly exceeds availability. It thus becomes necessary to partition the generation of formulae into smaller subsets. We do so by first generating "skeleton" formula trees, i.e., formula trees without labels for the nodes, thus only carrying information about the arity of their nodes. In other words, a skeleton tree represents a general "shape" for a formula, but does not specify the operators and atomic propositions in it. We will then generate all possible formula trees from a skeleton tree by "fleshing it out", i.e., by adding suitable operators and variables as labels to its nodes.

Formally, the grammar of such skeleton trees is

$$\mathsf{L} \mid \mathsf{U}\, t \mid t\, \mathsf{B}\, t'$$

where L represents a leaf node, U a unary node, and B a binary node, and t and t' are subtrees.

Given an LTL formula, we can compute its underlying skeleton tree by removing the labels from all the nodes. We define such operation $\mathcal{T}$ as follows:

$$\mathcal{T}(p) = \mathsf{L} \tag{1}$$

$$\mathcal{T}(\neg t \mid \mathrm{X}\, t) = \mathsf{U}\, \mathcal{T}(t) \tag{2}$$

$$\mathcal{T}(t \vee t' \mid t\, \mathrm{U}\, t') = \mathcal{T}(t)\, \mathsf{B}\, \mathcal{T}(t') \tag{3}$$

Skeleton trees, just like formulae, have a size given by the total number of nodes and leaves. We can generate all trees of a given size by using again a recursive algorithm, and then, for each tree t, we generate all the formulae ϕ such that $\mathcal{T}(\phi) = t$ by recursively replacing L with atomic propositions, U with unary operators $\neg$ and X, and B with binary operators $\vee$ and U. We proceed testing each generated formula for consistency with the sample, as before.

This updated algorithm, compared to the original one, presents the advantage that formulae are generated in small batches, checked and discarded before the next batch of formulae is generated, thus preventing the computer from running out of memory. As the great majority of computational time is spent evaluating formulae, rather than generating them, the increased complexity of the generating portion of the algorithm does not result in a noticeable degradation of performances.

B. Curbing logically equivalent formulae

Even with skeleton-tree optimization, the formula search algorithm could still be too slow for practical applications. Indeed, while lemma 1 guarantees that a solution to the passive learning problem exists, the size of such solution might be quite large, and the number of generated formulae grows exponentially with their size. To mitigate this issue, there are two viable solutions: to somehow reduce the number of formulae to check, and to speed up the process of checking if a formula is consistent with the given sample.

We now discuss a technique to implement the former solution. It is a standard mathematical fact that two formulae ϕ and ψ can be logically equivalent, i.e., such that for every trace t we have $t \vDash \phi$ if and only if $t \vDash \psi$. Logical equivalence is an equivalence relation over the set of formulae, and logically equivalent formulae are all consistent with the same samples, so it is sufficient to check for consistency relative to the sample a single formula out of each equivalence class. Unfortunately, verifying whether two LTL formulae are logically equivalent is a PSPACE-complete problem [26], [27].

A different perspective would be to provide a confluent term rewriting system, and consider only irreducible formulae. Such a rewriting system exists for Boolean logic, and can be derived from the axioms for Boolean ring via the Knuth-Bendix reduction algorithm [28], [29]. The temporal operators U, G and F of LTL, though, are fixed point operators, as they can be defined as the smallest operators such that the following equivalences hold [30].

$$\phi \,\mathrm{U}\, \psi \equiv \psi \vee (\phi \wedge \mathrm{X}(\phi \,\mathrm{U}\, \psi)) \tag{4}$$

$$\mathrm{G}\, \phi \equiv \phi \wedge \mathrm{X}\, \mathrm{G}\, \phi \tag{5}$$

$$\mathrm{F}\, \phi \equiv \phi \vee \mathrm{X}\, \mathrm{F}\, \phi \tag{6}$$

Unfortunately, we know of no technique to extend the Knuth-Bendix reduction algorithm to recursive operators.

Nonetheless, there are many known notable equivalences we can use to filter out at least some formulae, such as the excluded middle, $\phi \vee (\neg\phi) \equiv \top$, and the duality of temporal operators, e.g., $\neg(\mathrm{F}\, \phi) \equiv \mathrm{G}(\neg\phi)$. We apply these equivalences, with no claim to exhaustivity, to the generation of the list of formulae: whenever a generated formula is known, via pattern-matching, to be equivalent to a shorter one, we discard it as we know that the equivalent formula has already been generated and tested. Since the formula-generation algorithm is recursive and the number of generated formulae grows exponentially with size, discarding even a few formulae of a given size results in way fewer formulae generated at larger sizes. Special care is further required to deal with the associativity, commutativity and idempotency properties of logical conjunction and disjunction. The former is dealt with by conventionally choosing to always associate on the left. The latter two are dealt with by imposing an (arbitrary) total ordering $\preceq$ on the set of formulae, and discarding all formulae $\phi \wedge \psi$ and $\phi \vee \psi$ such that $\psi \preceq \phi$.

From a computational complexity point-of-view, the formulae-curbing operation does not change the fact that the number of generated formulae is exponential with respect to the size of the formulae, but it allows us to lower the base of such exponential. Thanks to the curbing of formulae, we obtain a $\sim 2\times$ speed-up on the sample used in section IX, and we expect such speed-up to grow exponentially as the size of the learned formulae increases.

C. Interleaving traces

A further $\sim 2\times$ speed-up is obtained by observing that, informally speaking, some formulae tend to be more likely than others to satisfy a random trace. For example, by definition tautologies are satisfied by every trace, and contradictions are satisfied by no trace. This means that checking a formula against all positive traces first, and then against all negative traces later, is inefficient if the formula tends to be easily satisfied (such as a tautology); conversely, checking a formula against all negative traces first, and then against all positive traces later, is inefficient if the formula tends to be rarely satisfied (such as a contradiction). Thus, instead of testing a formula on a sample by first checking if the formula satisfies all positive traces and then if it fails to satisfy all negative traces, it is generally more convenient to interleave positive and negative traces, thus alternating between checking the ones and checking the others.

VI. Theoretical evaluation

We need to ensure that the proposed learning algorithm is sound, complete, and it has the minimality property. Here, sound means that the algorithm does not provide a wrong answer, i.e., a formula inconsistent with the sample; complete means that no potential solution is ignored; and the minimality property means that the solution has minimal size among all formulae consistent with the sample.

By design, the proposed algorithm is sound, in that the produced formula is checked directly for consistency with the sample, and so it is guaranteed to satisfy the consistency requirement.

If the algorithm generates and tests all possible formulae, then it must also be complete, as it cannot miss a solution to the learning problem. The curbing of equivalent formulae described in section V-B has the effect of skipping some formulae, both in checking them against the sample and in inductively using them as subformulae to construct formulae of larger size. Still, consistency of a formula with a sample is a semantic-equivalence invariant problem, so as long as we only exclude formulae that are equivalent to other formulae that we check for consistency and use as subformulae of larger formulae, we are not going to miss any solution (up to equivalence).

Finally, since the algorithm generates formulae of increasing size, the first solution it finds must also be a minimal one. Again, the curbing of equivalent formulae does not break this property, because the curbed formula is always of size greater or equal than the retained equivalent one.

VII. Implementation details

We implemented the algorithm presented in section V in the Rust programming language.[2] The solver is provided with a sample encoded as a text file in either json or ron format, and the first formula that is verified to satisfy the sample is printed out as output.

Profiling, using tools like *hyperfine*,[3] *perf*,[4] *inferno*[5] and *cargo-flamegraph*,[6] shows that the vast majority of the execution time is spent evaluating the formulae on the traces, whereas memory usage is mostly due to the storing of all the generated formulae. By contrast, generating the formulae takes relatively little time, and evaluating them on the traces does not require significant memory usage.

The algorithm we propose is highly parallelizable. Indeed, testing whether a formula is consistent with the given sample can be done independently from the testing of the other formulae. Thus, parallelizing the algorithm via multi-threading on the CPU is trivial, and, while on small samples with solutions of small size the overhead of parallelism outweigh its benefits, on large samples with solutions of large size it yields a

[2] https://github.com/EnricoGhiorzi/learn_ltl
[3] https://github.com/sharkdp/hyperfine
[4] https://perf.wiki.kernel.org/index.php/Main_Page
[5] https://github.com/jonhoo/inferno
[6] https://github.com/flamegraph-rs/flamegraph

speed-up directly proportional to the number of available cores. This has also been experimentally tested on a high-performance computer cluster with up to 40 CPUs.

An implementation of a CUDA version of the algorithm has also been attempted. A kernel has been written to evaluate a formula on a trace, so that, after generating all formulae of a given size, these are evaluated in parallel to find one that is consistent with the sample. Unfortunately, experiments show that this approach does not yield improved performances compared to the CPU implementation, neither on a personal computer nor on a computer cluster with multiple GPUs. We conjecture that the operation of evaluating a formula on a trace is, intrinsically, a poor fit for the GPU, as it makes essential use of loops and branching, and thus it does not conform to the "single instruction, multiple data" paradigm.

VIII. Comparison with state-of-the-art

We developed our own algorithm to solve the passive learning problem because we want to learn linear temporal properties with large samples (such as those produced by autonomous robotic systems, which we expect to have up to a dozen variables and tens of traces of hundreds of time-steps of length, for a total of tens or hundreds of thousands of Boolean values) and we need to compute them fast (if we want a response in near-real-time) or on limited hardware (if the computation is done on the on-board hardware of a robotic system).

No off-the-shelf tool seems to offer the combination of features and performances we seek. We substantiate such statement by providing a comparative performance analysis of our tool pitted against two state-of-the-art solvers, which implement, respectively, the constraint-based approach and the enumerative approach. Although there exist other approaches to learn LTL formulae, we are not interested in comparing with those that use approximate methods, such as those that search among a restricted class of formulae, that use templates, or that use stochastic methods, as these search for sub-optimal (i.e., non-minimal) solutions in exchange for increased speed.

Finally, it is worth noting that all of the examples in this section are relatively easy to solve, and so our tool is able to produce a solution in a short time and with minimal memory footprint (a few MB at most). Still, since the computational complexity of the problem is exponential, it is easy to find samples that either cause the solver to run out of memory or that take a prohibitively long time to solve. For example, searching among formulae of size greater than 12–14 (depending also on the number of Boolean variables) is usually enough to crash the solver because of lack of memory.

A. Comparison with SAT approach

Our main references on the problem of passive learning of LTL formulae propose an algorithm based on SAT solvers [1], [2]. In a nutshell, given a sample S and a positive integer n, the algorithm defines a Boolean formula $\phi_{S,n}$ whose potential models, found by a SAT solver, induce an LTL formula of size n that is consistent with S. Notice that the notion of "size" in this paper differs from the one in [1], [2], but for the practical purposes of the performance comparison it does not make a qualitative difference, especially since on short formulae the two notions tend to agree. Unfortunately, the algorithm has, in the worst case, exponential complexity not only on the size of the formulae that are being searched, but also on the size of the sample (meaning the cumulative length of all the traces in the sample). Indeed, the number of variables in $\phi_{S,n}$ is proportional to n and to the size of S (see [1, remark 1]), and in the worst case a SAT solver has exponential complexity in the number of variables in the input formula. This makes the algorithm unsuitable to deal with large samples, even though the solution might be a short formula.

Samples	Cumulative search time (s)		
	samples2LTL (SAT)	our tool s.t.	our tool m.t.
universality	10 660	0.2057	0.0941
absence	43 194	0.1783	0.1123

Table I: Benchmark comparison with SAT solver.

The work in [1], [2] comes with an implementation of the proposed SAT-based algorithm, to which our tool aims to be a competitive alternative. Thus, we compare our tool, benchmarked with the aforementioned hyperfine tool, with the the SAT-based algorithm presented in [2],[7] passing the `--test_sat_method` option (the direct comparison with [1] is more problematic because it uses an infinite-traces sematics). We use the same sets of synthetically generated samples "absence" and "universality" used in [2],[8] and report the cumulative time to solve all samples in the sets. We repeat the experiment for both the single- and multi-thread implementations of our algorithm, keeping in mind that, while the former is deterministic, the latter is not and thus the results are subject to larger experimental error. The benchmarks in table I, which we run on a laptop PC,[9] show that our approach is faster by 5 orders of magnitude. Thus, in spite of the limitations in comparing different algorithms solving the passive learning problem, we claim that our approach is overall significantly faster and more capable than the one proposed in [1], [2]. On the one hand, the optimizations we applied to the exhaustive search algorithm and described in sections V-B and V-C leverage specific domain knowledge about LTL which would be harder to apply to SAT-based algorithms, as these have to rely on a third-party SAT solver which works as a black box. On the other hand, those optimizations account for a speed up of only one order of magnitude, so this shows that even a naive exhaustive search would be significantly faster than any SAT-based algorithm currently proposed in the literature, as those are all within a margin of one order of magnitude faster then those in [1], [2].

B. Comparison with SySLite

We now compare our tool with SySLite. We consider a set of samples from the SySLite repository, which have also been used in [4] to perform a performance comparison with the

[7] https://github.com/cryhot/samples2LTL

[8] https://github.com/cryhot/samples2LTL/tree/master/traces/finite/perfect_class

[9] Dell Inc. XPS 15 9500 with 16.0 GiB RAM, Intel® Core™ i7-10750H CPU @ 2.60 GHz × 12 cores, and GeForce GTX 1650 Ti Mobile.

Sample (1250 traces)	Search time (s)		
	SySLite	our tool s.t.	our tool m.t.
bank_transaction	3.144	0.0072	0.0068
chinese_wall_policy	31.568	0.0076	0.0067
dynamic_separation_of_duty	44.725	0.0097	0.0080
financial_institute	53.318	0.0320	0.0134
glba_6802	24.214	0.0164	0.0100
hippa_16450a2	23.840	0.0161	0.0097
hippa_16450a3	3.151	0.0072	0.0080

Table II: Benchmark comparison with SySLite.

algorithm from [1], [2]. Other than converting the sample files from the original format to our own, we reverse the order of the valuations in each trace to account for the fact that SySLite uses past-time LTL [4], so that the solutions found by the two tools match (up to renaming of the temporal operators). The benchmark results are reported in table II, from which we see that our implementation is significantly faster than SySLite. Moreover, SySLite only searches among formulae starting with a G operator, while our tool is not subject to such restriction and thus has to search among a larger set of formulae. We conjecture that our tool performs better because, unlike SySLite, it hard-codes the learning problem instead of relying on an external, more general solver.

IX. Experimental validation

To validate the application of passive learning of LTL formulae in the context of autonomous systems and the computational viability of the proposed learning algorithm, we performed experiments in which the R1 robot [31] attempts to grasp an item handed from a user and carry it to a destination point. There is also a charging station where the robot can recharge its battery. Potential issues which can prevent R1 from completing its task include running out of battery, finding a physical obstacle blocking the way, and incurring into an uncooperative user who refuses to hand the item.

The log produced during the execution of the simulation provides us with the following data:

- Whether R1 is grasping an object, as a Boolean value.
- Battery level, as a floating-point percentage.
- Spatial coordinates x and y as floating-point numbers in the ranges $[x_0, x_1]$ and $[y_0, y_1]$, respectively.

For every new log message we consider the updated state of the system, translate all data into Boolean variables, and append a new Boolean valuation to the trace. Boolean variables, such as the one determined by the grasping status, translate directly into the trace. Numerical variables, such as battery level or spatial coordinates, require choosing some sort of encoding.

We encode a floating-point numerical variable x in a range $[x_0, x_1)$ using a chosen number n of Booleans $X_1, \ldots, X_n$ by dividing the range in 2^n-many sub-intervals and using the Boolean variables to encode the binary representation of the index of the interval within which x is found.

We proceed as follows:

- We normalize the interval by letting $\bar{x} = 2^n \frac{x - x_0}{x_1 - x_0}$, so that $\bar{x} \in [0, 2^n)$.
- We take the integer part of $\bar{x}$ and we consider its binary representation $b_1, \ldots, b_n$ (n bits are sufficient).
- We let X_i be true if and only if $b_i = 1$.

Notice that this algorithm requires the upper bound of the interval to be open. Pragmatically, when dealing with a closed interval $[x_0, x_1]$, we consider the upper bound x_1 as belonging to the last interval.

A further issue is given by the length of the traces: since variables are sampled frequently, if we add a time-step to the trace at every variable sampling, we can end up with traces that are very long, while containing the same values repeated multiple times. This issue is exacerbated by the above encoding of numerical variables: even when the value of a numerical variable changes, its Boolean encoding might remain the same. The extreme size a sample with these kind of traces can reach actually provides an example of a sample on which our tool takes a prohibitive amount of time to terminate, although it does not run out of memory (we finally halted such an experiment after 16 hours of computation still yield no solution).

In principle, repeated and consecutive occurrences of the same variable evaluation in a trace are significative: the traces (in one variable) `[true]` and `[true, true]` are different and they do not satisfy the same LTL formulae. In practice though, the number of repeated occurrences of the variable evaluations are quite irrelevant: indeed, we make no assumption on the frequency or even regularity of the sampling, so that no reliable temporal information can be deduced from the number of repetitions. The traces from the example above might be produced by identical executions, where in the second trace the sampling just happens to have increased frequency. Thus, to reduce the length of the traces, and consequently speed up the execution of the learning algorithm, we deduplicate all consecutive repeated occurrences of the variable evaluations, obtaining a data compression ratio of up to 86.

From the execution logs we extract the following variables:

- `IsGrasping` is true if the robot is grasping an object.
- `B_1dd`, `B_d1d` and `B_dd1` encode battery level as a three-bit binary number, where `1` means that the bit in that position is 1 if the variable is true and 0 otherwise, and `d` that we "don't care" about that bit.
- `X_1ddd`, `X_d1dd`, `X_dd1d` and `X_ddd1` encode the spatial coordinate on the x axis as a four-bit binary number, with the same notation used for the battery level.
- `Y_1ddd`, `Y_d1dd`, `Y_dd1d` and `Y_ddd1` encode the spatial coordinate on the y axis as a four-bit binary number, with the same notation used for the battery level.

We repeat the experiments in both simulators and the real world, with different scenarios and increasing complexity. All of the obtained data is available at https://github.com/piquet8/masterThesisProject-Piquet. All of the following computations are executed run on the same laptop used for the benchmarks in section VIII, but without using multi-threading to make the results deterministic and more easily reproducible.

A. YARP simulation

The YARP simulation[10] used in [32] takes place in the IIT open space and "kitchen" area (see fig. 1a). In this 2D simulation, it is not possible to place obstacles in the scenario, and the grasping of the item will always succeed, but all the other features of the experiment, notably battery discharging and recharging, are implemented.

Consider the sample `RAL1.json` obtained from this experiment. The learning algorithm solves the sample and yields the formula `F(¬(X_d1dd ∨ X_ddd1))`. This means that R1 successfully completes the task only when eventually reaches an area approximately corresponding to the destination point.

The previous solution to the learning problem correctly identifies the destination area, but it's slightly disappointing in that it disregards the battery level. We run the learning algorithm after manually disabling the variables `X_d1dd` and `X_ddd1`, in the sense that the learning algorithm will disregard them and only consider the remaining variables to search for a solution. We then get the formula

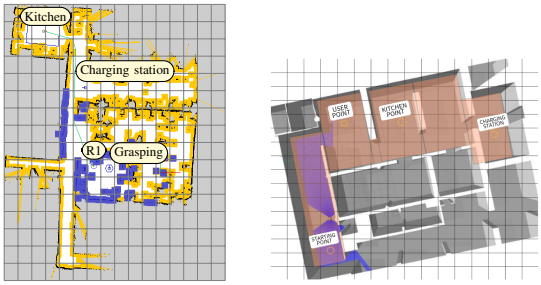
```
F(X_dd1d ∧ X(Y_1ddd ∧ (B_1dd ∨ B_d1d)))
```

in about 32 s and using around 200 MB (this is the only experiment in this section using significant resources). This means that eventually R1 needs to be in the kitchen area (although with greater approximation than the first solution). Moreover, the subformula `X(B_1dd ∨ B_d1d)` forces the robot to have at least a 25% charge the moment after entering the kitchen, which gives the robot sufficient battery charge to reach the destination. This solution correctly satisfies the negative trace corresponding to the execution in which R1 enters the kitchen without enough charge to reach the destination.

B. Tour Guide Robot (in Gazebo Simulator)

The Tour Guide Robot[11,12] takes place in a museum, simulated in 3D in the Gazebo Simulator (see fig. 1b). Compared to the simulator in section IX-A, we can also place obstacles to block the way of R1 to either the user, the charging station or the destination (still placed in a room conventionally called "kitchen"). We first execute some successful executions by not blocking any passage. Then, we block a single passage, execute some failures, and run our solver on the obtained sample. Blocking access to the user (`sample_w2.json`) we get `F(Grasp)`, meaning that R1 has to eventually grasp the item. Blocking the kitchen (`sample_w3.json`) we get `F(G X_d1dd)`, meaning that R1 has to eventually enter and remain in an area approximately corresponding to the kitchen. Block the charging station (`sample_w4.json`) we get again `F(G X_d1dd)`. If we disable the variable `X_d1dd`, though, we get that `F(G(¬X_1ddd))`, meaning that R1 should not go towards the charging station and remain there (evidently, stuck in front of the closed passage until its charge runs out).

[10]https://github.com/SCOPE-ROBMOSYS/Verification-experiments
[11]https://github.com/piquet8/tour-guide-robot
[12]https://www.youtube.com/watch?v=8L_4tDIS1Gs



(a) The YARP simulation scenario.

(b) The Tour Guide Robot simulation scenario.

Figure 1: Various scenarios. The grid reflects the binary encoding of the spacial coordinates.

C. Real-world experiments

For real-world experiments we used the actual R1 humanoid robot in the Robot Arena at IIT's center CRIS.[13] We verified that the results obtained in simulation in sections IX-A and IX-B can be replicated in the new, real-world scenario. Moreover, we had an uncooperative human user refuse to hand the item. This occurs in `sample_grasp.json`, which yields `F(Grasp)`, as expected.

X. Conclusions and Further Work

We have shown that a robust and optimized implementation of exhaustive search provides a practical approach to the problem of passive learning of LTL properties in the context of autonomous robotic systems, and it allows us to learn the root cause of failure of such a system in a realistic scenario.

In future work, we shall explore further strategies to optimize the learning algorithm. One improvement would be to develop a rigorous framework to curb equivalent formulae, instead of using a hand-picked library of notable equivalences as we currently do. Another improvement might be to evaluate the semantic of a formula on a trace by using bit-vectors computations [33], similar to SySLite. Finally, while the formulae cannot all be generated at the same time due to memory limitations, it might be worth using memoization techniques and precompute the semantic interpretation of at least some of the smaller formulae, so that it can be looked up when the formula is found as a subformula of a larger formula, instead of recomputing it every time.

Another area of research is to either extend or restrict the logical language. For example, restricting the language to a subset of LTL, such as the subsets of syntactically (co)safe formulae, could yield more useful results, and possibly improve performances of the learning algorithm too.

Expanding the area of research into system monitoring, we could extract online monitors from the learned formulae and use these to detect anomalies in the execution of the system. An even more ambitious goal would be to use these techniques not just to detect anomalies once these have occurred already, but to actively try to predict and prevent them.

[13]https://www.youtube.com/watch?v=qedEZL8t7cs

REFERENCES

[1] D. Neider and I. Gavran, "Learning linear temporal properties," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–10.

[2] J.-R. Gaglione, D. Neider, R. Roy, U. Topcu, and Z. Xu, "Learning linear temporal properties from noisy data: A maxsat-based approach," in *Automated Technology for Verification and Analysis*, Z. Hou and V. Ganesh, Eds. Cham: Springer International Publishing, 2021, pp. 74–90.

[3] ——, "Maxsat-based temporal logic inference from noisy data," *Innovations in Systems and Software Engineering*, 2022.

[4] M. F. Arif, D. Larraz, M. Echeverria, A. Reynolds, O. Chowdhury, and C. Tinelli, "Syslite: syntax-guided synthesis of pltl formulas from finite traces," in *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020*, ser. Conference Series: Formal Methods in Computer-Aided Design, A. Ivrii and O. Strichman, Eds., vol. 1. TU Wien Academic Press, 2020, pp. 93–103.

[5] N. Fijalkow and G. Lagarde, "The complexity of learning linear temporal formulas from examples," in *Proceedings of the Fifteenth International Conference on Grammatical Inference*, ser. Proceedings of Machine Learning Research, J. Chandlee, R. Eyraud, J. Heinz, A. Jardine, and M. van Zaanen, Eds., vol. 153. PMLR, 8 2021, pp. 237–250. [Online]. Available: https://proceedings.mlr.press/v153/fijalkow21a.html

[6] Z. Kong, A. Jones, A. Medina Ayala, E. Aydin Gol, and C. Belta, "Temporal logic inference for classification and prediction from data," in *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 273–282. [Online]. Available: https://doi.org/10.1145/2562059.2562146

[7] Z. Kong, A. Jones, and C. Belta, "Temporal logics for learning and detection of anomalous behavior," *IEEE Transactions on Automatic Control*, vol. 62, no. 3, pp. 1210–1222, 2017.

[8] J. Kim, C. Muise, A. Shah, S. Agarwal, and J. Shah, "Bayesian inference of linear temporal logic specifications for contrastive explanations," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 5591–5598. [Online]. Available: https://doi.org/10.24963/ijcai.2019/776

[9] G. Bombara, C.-I. Vasile, F. Penedo, H. Yasuoka, and C. Belta, "A decision tree approach to data classification using signal temporal logic," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–10. [Online]. Available: https://doi.org/10.1145/2883817.2883843

[10] G. Bombara, "Learning temporal logic formulae from data," Ph.D. dissertation, Boston University, 2020.

[11] G. Bombara and C. Belta, "Offline and online learning of signal temporal logic formulae using decision trees," *ACM Trans. Cyber-Phys. Syst.*, vol. 5, no. 3, 3 2021. [Online]. Available: https://doi.org/10.1145/3433994

[12] Z. Xu, M. Ornik, A. A. Julius, and U. Topcu, "Information-guided temporal logic inference with prior knowledge," in *2019 American Control Conference (ACC)*, 2019, pp. 1891–1897.

[13] G. Chen, M. Liu, and Z. Kong, "Temporal-logic-based semantic fault diagnosis with time-series data from industrial internet of things," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 5, pp. 4393–4403, 2021.

[14] H. Riener, "Exact synthesis of ltl properties from traces." New York: IEEE, 2019. [Online]. Available: http://infoscience.epfl.ch/record/280073

[15] R. Roy, D. Fisman, and D. Neider, "Learning interpretable models in the property specification language," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, ser. IJCAI'20, 2021.

[16] N. Baharisangari, J.-R. Gaglione, D. Neider, U. Topcu, and Z. Xu, "Uncertainty-aware signal temporal logic inference," in *Software Verification*, R. Bloem, R. Dimitrova, C. Fan, and N. Sharygina, Eds. Cham: Springer International Publishing, 2022, pp. 61–85.

[17] A. Camacho and S. A. McIlraith, "Learning interpretable models expressed in linear temporal logic," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 29, no. 1, pp. 621–630, 5 2021. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/3529

[18] I. Gavran, E. Darulova, and R. Majumdar, "Interactive synthesis of temporal specifications from examples and natural language," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 11 2020. [Online]. Available: https://doi.org/10.1145/3428269

[19] G. Chou, N. Ozay, and D. Berenson, "Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations," in *Robotics science and systems*, 2020. [Online]. Available: http://www.roboticsproceedings.org/rss16/p097.pdf

[20] ——, "Learning temporal logic formulas from suboptimal demonstrations: theory and experiments," vol. 46, pp. 149–174, 1 2022.

[21] L. Nenzi, S. Silvetti, E. Bartocci, and L. Bortolussi, "A robust genetic algorithm for learning temporal specifications from data," in *Quantitative Evaluation of Systems*, A. McIver and A. Horvath, Eds. Cham: Springer International Publishing, 2018, pp. 323–338.

[22] S. Mohammadinejad, J. V. Deshmukh, A. G. Puranic, M. Vazquez-Chanlatte, and A. Donzé, "Interpretable classification of time-series data using efficient enumerative techniques," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3365365.3382218

[23] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric identification of temporal properties," in *Runtime Verification*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 147–160.

[24] P. Vaidyanathan, R. Ivison, G. Bombara, N. A. DeLateur, R. Weiss, D. Densmore, and C. Belta, "Grid-based temporal logic inference," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017, pp. 5354–5359.

[25] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. Association for Computing Machinery, 2013, pp. 854–860.

[26] A. P. Sistla and E. M. Clarke Jr., "The complexity of propositional linear temporal logics," *J. ACM*, vol. 32, no. 3, p. 733–749, 7 1985. [Online]. Available: https://doi.org/10.1145/3828.3837

[27] M. Y. Vardi, *An automata-theoretic approach to linear temporal logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 238–266. [Online]. Available: https://doi.org/10.1007/3-540-60915-6_6

[28] D. E. Knuth and P. B. Bendix, *Simple Word Problems in Universal Algebras*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 342–376. [Online]. Available: https://doi.org/10.1007/978-3-642-81955-1_23

[29] J. Hsiang and N. Dershowitz, "Rewrite methods for clausal and non-clausal theorem proving," in *Automata, Languages and Programming*, J. Diaz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 331–346.

[30] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 4 2008.

[31] A. Parmiggiani, L. Fiorio, A. Scalzo, A. V. Sureshbabu, M. Randazzo, M. Maggiali, U. Pattacini, H. Lehmann, V. Tikhanoff, D. Domenichelli, A. Cardellino, P. Congiu, A. Pagnin, R. Cingolani, L. Natale, and G. Metta, "The design and validation of the r1 personal humanoid," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 674–680.

[32] M. Colledanchise, G. Cicala, D. E. Domenichelli, L. Natale, and A. Tacchella, "Formalizing the execution context of behavior trees for runtime verification of deliberative policies," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 9841–9848.

[33] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "Efficient scalable verification of ltl specifications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 711–721.