Temporal Logic Planning with Soft Temporal Logic Constraints

Hazhar Rahmani  Jason M. O’Kane

Abstract— In this paper, we consider a temporal logic planning problem in which the objective is to find an infinite trajectory that satisfies an optimal selection from a set of soft specifications expressed in linear temporal logic (LTL) while nevertheless satisfying a hard specification expressed in LTL. Our previous work considered a similar problem in which linear dynamic logic for finite traces (LDL_f), rather than LTL, was used to express the soft constraints. In that work, LDL_f was used to impose constraints on finite prefixes of the infinite trajectory. By using LTL, one is able not only to impose constraints on the finite prefixes of the trajectory, but also to set ‘soft’ goals across the entirety of the infinite trajectory. Our algorithm first constructs a product automaton, on which the planning problem is reduced to computing a lasso with minimum cost. Among all such lassos, it is desirable to compute a shortest one. Though we prove that computing such a shortest lasso is computationally hard, we also introduce an efficient greedy approach to synthesize short lassos nonetheless. We present two case studies describing an implementation of this approach, and report results of our experiment comparing our greedy algorithm with an optimal baseline.

I. INTRODUCTION

Temporal logics have become one of the most powerful and expressive tools for planning in robotics [10], [12], [18], [24]. Such logics, including linear temporal logic (LTL) specifically, offer high-level, user-friendly languages for specifying complex missions and tasks. In fact, as simple and intuitive as temporal logic is for humans to understand, it is also precise and rigorous for robot algorithms to manipulate. In particular, temporal logic has disrupted the classical conception of motion and path planning —which deals with making a finite, point to point trajectory that avoids obstacles— by allowing the imposition of other kinds of temporal or spatial constraints and by allowing the robot to make infinite, rather than only finite, trajectories.

In this paper, we consider a temporal logic planning problem in which a robot is tasked to accomplish a mission specified by an LTL formula while optimally satisfying a set of additional, possibly conflicting, LTL formulas. These extra constraints could be user preferences, safety rules, soft goals, or other constraints.

To illustrate the setting, see Figure 1 in which a social enrichment robot, capable of making animal balloons and juggling, visits the residents of a retirement home. The robot’s basic mission is to visit the two common rooms, each infinitely often. In addition to this basic mission, however, the robot is also charged with satisfying a collection of soft constraints, given in order of their relative importance. For example, we might prefer to maintain fairness by ensuring, if possible, that after making animal balloons in room 1, it should also do the same act in room 2. Or perhaps the manager wants the robot to eventually perform juggling in room 2, if its current act in that room is making animal balloons. The residents of room 2 might even prefer not to see the balloon animal act at all. The essence of our problem is to determine how the robot can act, to satisfy its primary mission, along with some optimal subset of these kinds of soft constraints.

Our prior work [17] considered a related problem in which the soft constraints were expressed in linear dynamic logic for finite traces (LDL_f) [6]. Such formulas can express constraints only on finite prefixes of the trajectory, rather than on the entire trajectory as a whole. A limitation of that work is that LDL_f soft constraints cannot express soft goals that are satisfied only by infinite (rather than finite) trajectories. As an example, a task that requires the social enrichment robot to infinitely often perform the act of juggling is a simple soft goal that cannot be expressed by LDL_f. The difference in the language used to express the soft constraints not only improves the expressivity of the approach, but it leads to significant (and new, compared to the LDL_f case) algorithmic challenges.

We contribute in this paper, a general formulation of the kind of problem in Figure 1. To do so, we first review related work in Section II and then present our problem statement in Section III. In Section IV we propose our algorithm, which first makes a state-weighted product automaton from the inputs, over which a lasso with minimum weight should
be synthesized. We prove that computing a shortest such lasso, even with any constant approximation factor, is computationally hard. Thus, we introduce an algorithm using a greedy approach to synthesize a short (but not necessarily shortest) lasso with minimum weight. In Section VI we present two case studies, and finally, in Section VI we present concluding remarks and discuss future work.

II. RELATED WORK

Our temporal logic planning is related to, but distinct from, several threads of prior work, which consider temporal logic planning in situations where no plan satisfying a given temporal logic formula can be synthesized. Fainekos [9] introduced an LTL revision problem, which, upon failures to plan a trajectory for an LTL formula, provides information about why that failure occurred, and how the LTL formula can be revised so that the transition system has a satisfiable temporal logic formula can be synthesized. Lahijanian et al. [16] later extended that idea to introduce the main problem we address.

Lahijanian et al. [15] propose, based on a user-defined priority over atomic propositions, which they assumed to be low level tasks, an approach to measure how “close” is a trajectory to satisfy a given formula, and accordingly, propose an algorithm that generates a trajectory that has the minimum distance to the satisfaction of that given formula. Lahijanian and Kwiatkowska [16] later extended that idea for probabilistic environments modeled by MDPs.

Two recent results by Dimitrova et al. [7] and Tomita et al. [21] consider a problem, called maximum realizability problem, which is a synthesis problem from a hard constraint and a set of soft constraints in the form of LTL formulas. The aim of this problem is to synthesize a reactive transition system (rather than a trajectory within a transition system). They consider the case where the soft constraints are of a specific kind of LTL formulas, those who assert that something globally holds. Their ideas are based on optimally refining or relaxing the soft specifications such that the resulting soft specifications along with the hard specification are realizable by a reactive transition system.

The closest work to ours is by Tumova et al. [22], who address a similar problem but without the hard constraint; they consider the problem of making a trajectory that maximizes the sum of rewards from satisfying a set of conflicting LTL formulas. Their algorithm first makes a generalized Büchi automaton for each LTL formula, and then from those automata, using the idea of converting a generalized Büchi automaton to a Büchi automaton, it makes a transition-weighted Büchi automaton, in which the Cartesian product of the state spaces of the automata are copied into different layers, a layer for each of the original automata, to keep track of the set of LTL formulas for which a run over the automaton is satisfying. From this transition-weighted Büchi automaton and the transition system, a product automaton is constructed, on which an accepting lasso is synthesized using a modified version of nested-DFS [5]. Our algorithm, which is simpler, constructs a state-weighted (rather than a transition-weighted) product automaton, and then uses a greedy approach to synthesize on this product automaton, a short accepting lasso with minimum weight; we prove that an accepting lasso with minimum length and weight is computationally hard to find.

Our synthesis process over this product automaton is performed in two phases, first is the one-pass DFS of Tarjan’s algorithm [20] to compute the set of strongly connected components (SCCs) of the product automaton, and second is a pass that synthesizes the prefix of the lasso as a simple path from the initial state to a leader of a SCC with minimum weight and the suffix of the lasso as a cycle within that SCC using BFS iteratively. The work of Tumova et al. [22] does not consider synthesis of a shortest accepting lasso. Two other results from the same authors [3], [23] consider for the classical setting of path planning, generating a finite trajectory that minimizes the amount of time the robot deviates only the less important ones of a set of conflicting safety rules. This problem is for finite trajectories and is treated differently.

III. DEFINITIONS AND PROBLEM STATEMENT

In this section, we review some preliminary tools and introduce the main problem we address.

A. Preliminaries

The set of infinite-length words over an alphabet $\Sigma$ is denoted $\Sigma^\omega$, and the infinite repetition of a finite word $r \in \Sigma^*$ is denoted $r^\omega$. A lasso is formed when such an infinite repetition is concatenated to a finite word, that is, a lasso is an infinite word of the form $r_1(r_2)^\omega$, in which $r_1 \in \Sigma^*$ and $r_2 \in \Sigma^+$. The environment is modeled as a transition system.

**Definition 1:** A transition system $T = (S, R, s_0, AP, L)$ consists of a finite set of states $S$; a transition relation $R \subseteq S \times S$; an initial state $s_0 \in S$; a set of atomic propositions $AP$; and a labeling function $L : S \rightarrow 2^{AP}$, which assigns to each state, a set of atomic propositions, which are properties that hold at that state.

An execution of the system goes through an infinite path $\pi = s_0s_1s_2\cdots \in S^\omega$, in which $s_0$ is the initial state and for each $i \geq 0$, $(s_i, s_{i+1}) \in R$. The trace of this path is $\text{trace}(\pi) = (s_0) L(s_1)L(s_2)\cdots \in (2^{AP})^{\omega}$. The transition systems we deal with should be free of blocking states— those states that do not have outgoing edges. To specify a set of traces, one can use a variety of logical formulas, including those in LTL.

**Definition 2:** An LTL formula is generated over a set of atomic propositions $AP$ by the following grammar

$$\varphi ::= T \mid p \mid \neg \varphi \mid \varphi \lor \psi \mid \varphi U \psi,$$

in which $p \in AP$, $T$ represents the constant true, and $U$ (‘until’) and $\lor$ (‘next’) are temporal operators.
An LTL formula \( \varphi \) specifies a set of infinite words over \( 2^{AP} \), denoted \( \text{Words}(\varphi) \), which consists of those words that satisfy \( \varphi \). To see if a word (trace) \( \sigma = A_0A_1A_2\cdots \in (2^{AP})^\omega \) satisfies an LTL formula \( \varphi \), denoted \( \sigma \models \varphi \), one can use rules that

1. \( \varphi \models \top \) if \( p \in A_0 \)
2. \( \varphi \models \neg \varphi \) if \( \varphi \not\models \varphi \)
3. \( \varphi \models \psi \lor \varphi_2 \) if \( \varphi \models \varphi_1 \) or \( \varphi \models \varphi_2 \)
4. \( \varphi \models \bigcup \varphi_i \) if \( \varphi \models \varphi_{i[\ldots]} \) or \( \varphi \models \varphi_{2[\ldots]} \) and \( \forall 0 \leq i < j, \sigma[i..j] \models \varphi \). For simplicity, two other temporal operators \( \bigwedge \) ("eventually", defined as \( \bigwedge \varphi := \top \lor \varphi \)), and \( \bigvee \) ("globally"), defined as \( \bigvee \varphi := \neg \bigwedge \neg \varphi \), are also used. We can also use the dual of \( \lor \), which is \( \land \), as well as the Boolean operator \( \neg \).

Each LTL formula is equivalent to a set of B"uchi automata.

**Definition 3:** A B"uchi automaton \( A = (Q, \Sigma, \delta, q_0, F) \) consists of a finite set of states \( Q \); an alphabet \( \Sigma \); a transition relation \( \delta \subseteq Q \times \Sigma \times Q \); an initial state \( q_0 \in Q \), and a set of accepting (final) states \( F \subseteq Q \).

A run over the automaton is an infinite sequence \( r = q_0q_1q_2\cdots \in Q^\omega \) in which \( q_0 \) is the initial state and for each \( i \geq 0 \), \( (q_i, \delta, q_{i+1}) \in \delta \) for an \( A \in \Sigma \). Sequence \( r \) is a run for an infinite word \( A_0A_1A_2\cdots \in \Sigma^\omega \) if for each integer \( i \geq 0 \), \( (q_i, A_i, q_{i+1}) \in \delta \). The set of states that appear infinitely many times in an infinite run \( r \) is denoted \( \text{inf}(r) \). Accordingly, \( r \) is accepting if \( \text{inf}(r) \cap F \neq \emptyset \). Consequently, the language of \( A \), denoted \( L_\omega(A) \), is

\[
L_\omega(A) = \{ w \in \Sigma^\omega \mid \text{there exists an accepting run for } w \}.
\]

A connection between LTL formulas and B"uchi automata is that for any LTL formula \( \varphi \) over a set of atomic propositions \( AP \), one can construct a B"uchi automaton \( A_\varphi \) with alphabet \( 2^{AP} \) such that \( L_\omega(A_\varphi) = \text{Words}(\varphi) \). Several algorithms for this kind of construction are available [1], [11], [19], [26].

In our algorithm, we need to be sure that each B"uchi automaton created for a soft constraint is nonblocking, that is, for any state \( q \) in the automaton and every letter \( a \in \Sigma \), there is at least one state \( q' \) such that \( (q, a, q') \in \delta \). Any B"uchi automaton is converted to a nonblocking B"uchi automaton by adding a trapping state, to which all missing transitions are added. We also consider a variant of B"uchi automaton called generalized B"uchi automaton, which has the same syntax of B"uchi automaton except that it has a set \( \mathcal{F} \subseteq 2^Q \) rather than a set \( F \subseteq Q \) as its acceptance set. More precisely, the acceptance set of the automaton is a set \( \mathcal{F} \) consisting of sets \( F_1, F_2, \ldots, F_k \) with \( F_i \subseteq Q \) for each \( i \in \{1, \ldots, k\} \). Accordingly, an infinite run \( r \) over a generalized B"uchi automaton \( \mathcal{G} \) is accepting if for each \( F \in \mathcal{F} \), it holds that \( \text{inf}(r) \cap F \neq \emptyset \). The language of \( \mathcal{G} \), \( L_\omega(\mathcal{G}) \), is the set of all infinite words for each of which there is an accepting run.

**B. LTL planning with soft constraints**

Our goal in this problem is to find, in a transition system modeling the environment, an infinite path whose trace satisfies a goal mission expressed as an LTL formula \( \varphi \) while optimally satisfying a prioritized list of soft constraints \( \psi_1, \psi_2, \cdots, \psi_n \), where each \( \psi_i \) is an LTL formula, given in order of decreasing importance. For this purpose, we define a cost function \( f_\omega : (2^{AP})^\omega \to \mathbb{Z}_{\geq 0} \), such that for any \( \sigma \in (2^{AP})^\omega \),

\[
f_\omega(\sigma) = \sum_{i: \sigma \in \text{Words}(\psi_i)} n^{n-i}.
\]

Note that this cost function guarantees to impose the standard lexicographic ordering between all Boolean vectors, where each vector has an entry for each LTL constraint showing the language of \( \mathcal{G}_\omega(\mathcal{G}) \), which consists of those words who specify \( \varphi \). Words(\sigma) is the set of all \( \text{Words}(\sigma) \).

For simplicity, two other temporal operators \( \bigwedge \) ("eventually", defined as \( \bigwedge \varphi := \top \lor \varphi \)), and \( \bigvee \) ("globally"), defined as \( \bigvee \varphi := \neg \bigwedge \neg \varphi \), are also used. We can also use the dual of \( \lor \), which is \( \land \), as well as the Boolean operator \( \neg \).

Each LTL formula is equivalent to a certain type of automaton.

**Problem: Optimal LTL Planning with Soft Constraints (OLPSC)**

**Input:** A transition system \( T \), an LTL formula \( \varphi \), and a prioritized list of \( n \) LTL formulas \( \psi_1, \psi_2, \cdots, \psi_n \).

**Output:** An infinite path \( \pi \) over \( T \) such that \( \text{trace}(\pi) \models \varphi \) and \( f_\omega(\text{trace}(\pi)) \) is minimized.

**IV. ALGORITHM DESCRIPTION**

This section presents an algorithm for solving the OLPSC problem. See Algorithm [1] The two main steps of this algorithm are constructing a product automaton (line 6), and computing a lasso with minimum cost on the product automaton (line 7). In the sequel, we explain these steps.

**A. The product automaton**

The first step of the algorithm is, following an established pattern in the literature [3], [15], [22], [23], to construct a form of product automaton [25]. To that end, the algorithm first makes the B"uchi automata representations of the LTL formulas — an automaton \( \mathcal{A} \) for \( \varphi \), and an automaton \( \mathcal{B}_i \) for each \( \psi_i \). It then ensures that \( \mathcal{B}_i ' s \) are nonblocking and uses all those automata along with the transition system to construct a product automaton based on the following definition.

**Definition 4:** For a B"uchi automaton \( \mathcal{A} = (Q, 2^{AP}, \delta, q_0, F) \), a transition system \( T = (S, R, s_0, AP, L) \), and a prioritized list of \( n \) nonblocking B"uchi automata \( \mathcal{B}_i = (Q_i, 2^{AP}, \delta_i, q_{0,i}, F_i) \) for \( i \in \{1, \ldots, n\} \), the product automaton is a tuple \( \mathcal{P} = (Q_P, \delta_P, q_{0,P}, F_P, w) \) in which

1) \( Q_P = Q \times S \times Q_1 \times \cdots \times Q_n \) is a finite set of states;
2) \( q_{0,P} = (q_0, s_0, q_{0,1}, \ldots, q_{0,n}) \) is the initial state;
3) \( \delta_P \subseteq Q_P \times Q_P \) is a transition relation, such that \( ((q, s, q_1, \ldots, q_n), (q', s', q_1', \ldots, q_n')) \in \delta_P \) if and only if \( (s, s') \in R \), \( (q, L(s), q') \in \delta_i \), and \( (q_i, L(s), q_i') \in \delta_i \) for each \( i \in \{1, \ldots, n\} \); 4) \( F_P = F \times S \times Q_1 \times \cdots \times Q_n \) is the set of accepting states;
5) \( w : Q_P \to \{T, F\}^n \) is a state-weighting function that assigns to each state \( (q, s, q_1, \ldots, q_n) \in Q_P \), a Boolean vector \( v \) such that for any \( 1 \leq i \leq n \), it holds that \( v[i] = T \) if and only if \( q_i \in F_i \).
This product automaton can be thought of as a Büchi automaton, which is known to be NP-hard [4], [8]. For even infinitely many optimal runs; in fact, there could exist an optimal run whose sequence of states cannot be specified by any pattern; however, we are interested in only one kind, which is revealed by the following result.

**Lemma 3:** Given a product automaton \( P \), let \( r_P = (r_1, r_2, \ldots, r_n) \) be a run over \( P \). It holds that:

1. If \( r_P \) is accepting for \( P \), then the sequence \( \pi = (s_0, s_1, \ldots, s_n) \) is a path over \( T \) such that \( \text{trace}(\pi) \in L_\omega(A) \).
2. For any \( i \in \{1, \ldots, n\} \), if \( \inf(\text{trace}(\pi)) = T \), then the sequence \( \pi = (s_0, s_1, \ldots, s_n) \) is a path over \( T \) such that \( \text{trace}(\pi) \in L_\omega(B_i) \).

**Proof:** (1) From the construction of \( P \), it follows that the sequence \( \pi = (s_0, s_1, \ldots, s_n) \) is a path over \( T \) such that \( \text{trace}(\pi) \in L_\omega(A) \). Since \( r_P \) is accepting for \( P \), there are infinitely many states \( q_i \) in \( F \), implying that \( r \) is accepting, and thus, \( \text{trace}(\pi) \in L_\omega(A) \).

(2) The proof is similar to the proof of (1) with the consideration that in this case, for each \( i \), sequence \( r_i = (q_0, q_{0,1}, q_{0,2}, \ldots) \) is a run for \( \text{trace}(\pi) = \) for any path \( \pi = (s_0, s_1, \ldots, s_n) \) in \( T \) such that \( \text{trace}(\pi) \in L_\omega(A) \) and \( \text{trace}(\pi) \in \bigcap_{i \in I} L_\omega(B_i) \), there exists an accepting run \( r_P = (q_0, q_0,1, q_0,2, \ldots) \) over \( \Pi \) such that \( \inf(\text{trace}(\pi)) = T \) for all \( i \).
Our algorithm uses graph algorithms to minimize \(|r_1|\) and \(|r_2|\) separately. Algorithm 2 shows the process. Consider that \(P\) can be thought of as a directed graph with vertex set \(Q_P\) and edge set \(\delta_P\). Additionally, all vertices (states) in \(r_2\) are in a strongly connected component (SCC) of the graph given that they are contained in a cycle, \(r_2, r_2\) [0]. With these in mind, our algorithm first decomposes the graph into its strongly connected components, (Line 1); then finds a SCC that contains \(r_2\) of a lasso \(r_P = r_1(\epsilon r_2)^*\) with minimum \(f_w(r_P)\) (Lines 2-11); and then construct \(r_1\) and \(r_2\) (Lines 13 and Line 14 respectively). See Figure 2.

To find the set of SCCs of the graph, we use the well-known algorithm of Tarjan [20]. This algorithm uses depth first search (DFS) to traverse all the vertices (states) of the graph in one pass. During this traversal, each vertex \(p\) is assigned a unique integer \(p\).\_number, which is, in fact, the traversal’s step number at which \(p\) is reached. Each vertex is assigned another integer \(p\).\_lowlink, whose value is set to the smallest index of any node reachable from \(p\), including \(p\) itself. During this algorithm, all vertices that are assigned the same value of \(\text{lowlink}\) will be in the same SCC of the graph, and among those vertices, the one whose number is equal to its \(\text{lowlink}\) is the leader (representative) of the SCC.

As Tarjan’s algorithm executes, we also compute for each SCC \(C\), the value of \(C\).accepting, which gets \textit{true} only when \(C\) contains an accepting state of \(P\) and that \(C\) is not a singleton vertex who does not have a loop. We also compute the value of Boolean vector \(C\).w, whose value is set to \(C\).w = \(\sum_{q \in C} w(q)\). Notice that during the same pass of the Tarjan’s algorithm, one can keep for each vertex, a link to its parent in the DFS search. Accordingly, later, the algorithm can use those links to find, for each state, a path from the initial state to that state. These paths can be used in Line 13 to compute \(r_1\), which is, in fact, a path from the initial state to the leader of the component which we choose to construct \(r_2\) from. However, a path \(r_1\) that is constructed in that way may not have minimum length. Accordingly, we use BFS to find a shortest simple path from \(q_0, p\) to \(C\).leader.

The final phase of finding an accepting lasso is to synthesize the suffix \(r_2\) of it. Line 14 of Algorithm 2. This suffix is synthesized inside an optimal SCC \(O\) using the following greedy algorithm. Let \(r'_P = \bigcup_{n \geq 0} w(r_P)\), and let \(I = \{1 \leq i \leq n \mid w(r'_P)[i] = T\}\). Let for each \(i \in I\), \(F_i\) be the set of states in \(O\) that are accepting for \(\text{Büchi automaton } B_i\), i.e., \(F_i \cap Q = \{q \in Q \mid w(q)[i] = T\}\), and let \(F_{i, P}\) be the set of accepting states in \(O\), i.e., \(F_{i, P} = F_i \cap O\). Our greedy algorithm synthesizes \(r_2\) as the vertices (states) of a cycle starting from \(O\).leader such that the cycle contains at least a state of \(F_{i, P}\) and at least a state in \(F_{i, P}\) for each \(i \in I\). To do so, it uses variable \(U\) with initial value \(F_{0, P} \cup \{F_i \mid i \in I\}\). It starts from \(O\).leader, and performs breadth first search (BFS) until it finds a state \(s\) for which there is a set \(M \subseteq U\) such that \(s \in M\). Using the parent links set during BFS, the algorithm records the shortest path from \(O\).leader to \(s\) as a first portion of \(r_2\), and
removes from \( U \) all those sets \( M \) for which \( s \in M \). It then
does a similar task, BFS traversal, starting from \( s \), and then
records the path traversed from \( s \) to the new found state. It
repeatedly does this process until \( U \) becomes \( \emptyset \). At this time,
it does a BFS to find the shortest path back to \( O.leader \).
By this time, it has made \( r_2 \) as the vertices of a cycle. Figure 2
illustrates how inside SCC \( C_5 \), the algorithm constructs \( r_2 \).

Given this discussion, we now analyze the time complexity
of our algorithm.

**Lemma 5:** For any automaton \( \mathcal{P} = (Q, \delta, \rho, Q_0, \mathcal{F}, \mathcal{W}) \), a lasso with minimum cost
can be generated in time \( O(n(|\delta| + |Q|)) \).

**Proof:** It takes \( O(|\delta| + |Q|) \) time to find the set of
strongly connected components. BFS is called at most \( n \) times,
each of which takes \( O(|\delta| + |Q|) \) time. This bound simplifies to
\( O(|\delta| + |Q|) \) if the number of soft constraints \( n \) is treated as a constant.

We can slightly improve the quality of solution by letting
instead of the leader, the first state to which BFS reaches
from the leader and who is either a final state or it corresponds
to a final state of the Büchi automata for a soft constraint,
to be the midpoint of the lasso.

Though Algorithm 2 generates a lasso of minimum cost,
it is not guaranteed to produce the shortest such lasso. In
the next section, we compare our algorithm with an optimal
brute-force algorithm, which finds the shortest lasso by
letting any state within an optimal SCC to be the midpoint
of the lasso, and synthesizes the suffix of the lasso by searching
from the shortest ones, all cycles, simple or otherwise, that
start from the midpoint until it finds a satisfactory cycle or
the length will be longer than the length of a shortest lasso
computed for other midpoints.

V. CASE STUDIES AND EXPERIMENTS

We have implemented Algorithms 1 and 2 in Java. The
computed results were executed on an Ubuntu 16.04 com-
puter with a 3.6GHz processor.

A. Case study: Hospital deliveries

Figure 3 shows a hospital which has an emergency de-
partment (e), a primary care department (p), a maintenance
department (n), a pharmacy (h), and a warehouse (w). In
this hospital, a robot delivers first aid items (f) and medicine
(m) from the warehouse to the other departments. Each state
of the transition system for this case study has an atomic
proposition indicating a unit within the hospital, along with
other propositions indicating which items the robot is caring
at that unit. Those states are connected according to the
connectivity of the units within the hospital and the items
the robot can take or leave at a unit. The robot’s primary tasks
are to deliver first aid items to the emergency department,
deliver first aid items to primary care, deliver medicine to
the pharmacy, and report for maintenance, each infinitely
often. In addition, suppose the robot is given these soft
specifications, ordered from most to least important:

1) If first aid items are delivered to the primary care
department, then do not deliver additional first aid items
there until first aid items have been delivered to the
emergency department, and vice versa.
2) If the first aid items are picked from the warehouse,
then they must not be delivered to the primary care
department until the emergency department receives the
first aid items.
3) Do not carry first aid items and medicine together.
4) Always pick both first aid items and medicine when
leaving warehouse.

Notice that, in particular, the first two constraints cannot
be expressed in LDL\(_P\). Thus, the algorithm from our prior
work [17] cannot generate a plan for this instance. The
box below shows how to formulate these constraints into
an instance of OLPSC, along with the solution computed by
our implementation.

**Goal:** \( \square \diamond (p \land f) \land \square \diamond (e \land f) \land \square \diamond (h \land m) \land \square \square n \)

**Soft constraints:**

1) \( \square ((p \land f) \rightarrow \square \neg (p \land e \land f)) \land \square ((e \land f) \rightarrow \square \neg (e \land (p \land f))) \)
2) \( \square ((e \land f) \rightarrow (\neg p \land e)) \)
3) \( \square (\neg w \rightarrow \neg (f \land m)) \)
4) \( \square ((w \land c) \land \land (f \land m)) \)

**Solution:**

\[ \varphi \equiv \psi \tau \land \rho \]

**Satisfied constraints:** 1, 3

**Computation time:** 201.50s

In the sequence shown for the solution, a letter is the
location of the robot, and the subscript of the letter, if any,
is what the robot is carrying. Not that this optimal solution
satisfies only the first and third soft constraints.

For comparison purposes, we also executed on this example,
the brute-force algorithm to compute a shortest accepting
lasso that minimizes the cost function \( f_\omega \). That algorithm
failed to compute such a lasso in 15 hours.

B. Case study: Retirement home

In this section, we look back to the retirement home exam-
ple from Section I. The transition system in that example
has atomic propositions for locations — \( r_1 \) for common room
1, \( r_2 \) for common room 2, and \( f \) for toy room — and also
for the robot acts —g for juggling, and b for making animal balloons. The robot is tasked to visit r1, r2, and t, each one infinitely often. The robot’s specification also includes 6 soft constraints (some of which were mentioned in Section I):

1) After making animal balloons in room 1, the robot should immediately do the same act in room 2.
2) The robot should perform each of the acts in room 1 infinitely often.
3) The robot should not perform the act of making animal balloons in room 2.
4) If the current act in room 2 is making animal balloons, then the robot should eventually make animal balloons in that room.
5) The robot should not stay in a common room once it performed an act.
6) The robot should perform at least two acts in each common room once it has entered that room. Those two acts can be different or not.

We can formalize this scenario as an instance of OLPSC:

**Goal:** □♦r1 ∧ □♦r2 ∧ □♦t

**Soft constraints:**
1) □(r1 ∧ b) → (¬b ∧ ¬g) U (r2 ∧ b)
2) □□(r1 ∧ g) ∧ □□(r1 ∧ b)
3) □(r2 → ¬b)
4) □(r2 ∧ b) → (r2 ∧ g)
5) □(r1 → ¬r1) ∧ □(r2 → ¬r2)
6) □(¬r1 ∧ r1) → (¬r1 ∧ r1)

**Solution:**

\[ s1s_{r1}, g\sigma1s_{r2}, b(\sigma1s_{r1}, b\sigma1s_{r2}, g\sigma1s_{r2}, g\sigma1s_{r1}, b\omega)\]

Satisfied constraints: 1, 2, 4, 5

**Computation time:** 0.02s (excluding product automaton construction)

For comparison, the brute-force algorithm computed \( r^*_p \) as a shortest accepting lasso minimizing \( f_w \) in 3,254 seconds. The shortest accepting lasso has length 16, while the lasso generated by our algorithm has length 26.

Now suppose there is a change in the relative ordering of the soft constraints, in which the last two are swapped. This induces a change to the set of constraints that can be satisfied, but not to the basic structure of the product automaton.

**Goal:** □♦r1 ∧ □♦r2 ∧ □♦t

**Soft constraints:**
1) – 4) Same as above.
5) □((¬r1 ∧ □r1) → □r1)
6) □(r1 → □¬r1) ∧ □(r2 → □¬r2)

**Solution:**

\[ s1s_{r2}, b(\sigma1s_{r2}, g\sigma1s_{r2}, b\sigma1s_{r2}, g\sigma1s_{r1}, g\sigma1s_{r2}, g\sigma1s_{r1}, b\sigma1s_{r2}, g\sigma1s_{r2}, g\sigma1s_{r1}, b\omega)\]

Satisfied constraints: 1, 2, 4, 5

**Computation time:** 0.02s (excluding product automaton construction)

In fact, if we have already computed the product automaton for the instance above, we need only to synthesize an accepting lasso, without any need to reconstruct the product automaton again. To synthesize the new lasso, for each state of the product automaton, we swap the elements of vector \( w \) according to the new order of constraints, and then synthesize the lasso.

It took only 20 milliseconds to synthesize an optimal run on the constructed automaton, while for the first one it took 21.91 seconds, much of which was spent to form the product automaton. We also use the brute-force algorithm to compute the shortest accepting lasso \( r^*_m \) that minimizes the \( f_w(r^*_m) \). It took 110 seconds, excluding the time of product automaton construction, for the brute-force algorithm to compute the shortest accepting lasso, which was \( r^*_p = s1s_{r1}, g\sigma1s_{r1}, b\sigma1s_{r2}, g\sigma1s_{r1}, b\sigma1s_{r2}, b\sigma1s_{r2}, g\sigma1s_{r1}, b\omega \), with length 16. Observe that the length of the lasso generated by our greedy algorithm was 23 while the length of the shortest accepting lasso was 16. The product automaton for this problem had 1440 states.

### C. Experiments

In this section, we present results of our experiment comparing our (greedy) algorithm with the brute-force algorithm. We performed all those experiments on the same machine on which we executed our case studies. In this experiment, we execute both the greedy algorithm and the brute-force algorithm on a large number of graphs (product automata) of different sizes that we generated randomly by the Erdős-Rényi model of \( G(n, p) \), according to which each edge will be included in the graph with probability \( p \) independent from any other edge.

Figure 4 shows results of our experiment. For each graph size —100, 200, 300, and 500— we generated 100 random graphs. The number of edges for each graph was approximately five times the number of vertices, and for each graph, approximately 20 percent of the states were final states. The number of soft constraints, the size of the Boolean vectors assigned to a state by \( w \), for each graph was 10. We report for each graph size, the average time to compute an accepting lasso with minimum cost by our algorithm, and also the average size of the generated lassos. For each test, the greedy algorithm had 20 minutes to find a shortest lasso. Figure 4 shows also for each graph size, the number of times the brute-force algorithm was able to compute a shortest lasso within the 20 minutes time window. Notice that for graph size 300, the brute-force algorithm failed four times to compute a minimal lasso within that time window, and for graph size 500, it was able to compute a shortest lasso only for 62 tests. Accordingly, we considered in computing all those averages shown for the brute-force algorithm in that figure, only those tests for which the algorithm was able to compute a solution in 20 minutes. This means that the actual time averages for graph size 300 and 500 are higher, and probably much higher, than those shown in that figure. The average number of constraints satisfied were ranged from 6.30 to 6.81.

For each test, we also computed approximation ratio, defined as the size of the lasso generated by the greedy algorithm over the size of the lasso generated by the brute-
force algorithm. The minimum, maximum, and the average of those ratios for each graph size is shown in Figure 4.

We observe from this experiment that the greedy algorithm generates a solution significantly faster than the brute-force algorithm while the quality of solution is still acceptable. For graph size 100, the distribution of the approximation ratios of the 100 tests we performed is shown in Figure 5.

We also executed a variant of our algorithm, in which we let any final state within an optimal SCC be the midpoint and then chose a shortest one among all lassos generated for those midpoints. This algorithm increases the running time by the magnitude of the number of final states. We observed that the quality of solution is slightly improved. For the graph sizes of 100, the average of approximation ratio was 2.01 for this new variant, compared to 2.08 to original algorithm. Because product automata are generally quite large, we may not need sacrifice computation time in favor of slightly improved quality.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we considered temporal logic planning given both hard and soft specifications of the goal. Future work can consider learning soft constraints. It can also consider the case where the environment is dynamic. In this case, the changes are reflected in the product automaton, for which one needs to maintain the SCCs of the automaton in a data structure that is able to quickly adapt to the changes.

REFERENCES

[1] T. Babiak, M. Křetínský, V. Rehák, and J. Strejček, “Ltl to büchi automata translation: Fast and more deterministic,” in Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2012, pp. 95–109.

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

[3] L. I. R. Castro, P. Chaudhari, J. Tumova, S. Karaman, E. Frazzoli, and D. Rus, “Incremental sampling-based algorithm for minimum-violation motion planning,” in IEEE Conf. on Decision and Control. IEEE, 2013, pp. 3217–3224.

[4] E. M. Clarke, “Efficient generation of counterexamples and witnesses in symbolic model checking,” in Design Automation Conference, 1995.

[5] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, “Memory-efficient algorithms for the verification of temporal properties,” Formal Methods in System Design, vol. 1, no. 2-3, pp. 275–288, 1992.

[6] G. De Giacomo and M. Y. Vardi, “Linear temporal logic and linear dynamic logic on finite traces.” in IJCAI, vol. 13, 2013, pp. 854–860.

[7] I. Dimitrova, M. Ghasemi, and U. Topcu, “Maximum realizability for linear temporal logic specifications,” in Int. Symposium on Automated Technology for Verification and Analysis. Springer, 2018.

[8] A. Ehlers, “Short witnesses and accepting lassos in ω-automata,” in Int. Conf. on Language and Automata Theory and Applications. Springer, 2010, pp. 261–272.

[9] G. E. Fainekos, “Revising temporal logic specifications for motion planning,” in Proc. IEEE Int. Conf. on Robotics and Automation. IEEE, 2011, pp. 40–45.

[10] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, “Temporal logic motion planning for dynamic robots,” Automatica, vol. 45, no. 2, pp. 343–352, 2009.

[11] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, “Towards manipulation planning with temporal logic specifications,” in Proc. IEEE Int. Conf. on Robotics and Automation. IEEE, 2015, pp. 346–352.

[12] K. Kim, G. Fainekos, and S. Sankaranarayanan, “On the minimal revision problem of specification automata,” The International Journal of Robotics Research, vol. 34, no. 12, pp. 1515–1535, 2015.

[13] K. Kim, G. Fainekos, and S. Sankaranarayanan, “On the revision problem of specification automata,” in Proc. IEEE Int. Conf. on Robotics and Automation. IEEE, 2012, pp. 5171–5176.

[14] M. Lahijanian, S. Almagor, D. Fried, L. E. Kavraki, and M. Y. Vardi, “This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction,” in AAAI, 2015.

[15] M. Lahijanian and M. Kwiatkowska, “Specification revision for markov decision processes with optimal trade-off,” in Proc. Conf. on Decision and Control (CDC). IEEE, 2016, pp. 7411–7418.

[16] H. Rahmani and J. M. O’Kane, “Optimal temporal logic planning with cascading soft constraints,” in Proc. IEEERSI Int. Conf. on Intelligent Robots and Systems. IEEE, 2019, pp. 2524–2531.

[17] P. Schilling, M. Bürger, and D. V. Dimarogonas, “Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems,” The International Journal of Robotics Research, vol. 37, no. 7, pp. 818–838, 2018.

[18] F. Somenzi and R. Bloem, “Efficient büchi automata from ltl formulæ,” in Int. Conf. on Computer Aided Verification. Springer, 2000, pp. 248–263.

[19] R. Tarjan, “Depth-first search and linear graph algorithms,” SIAM Journal on Computing, vol. 1, no. 2, pp. 146–160, 1972.

[20] T. Tomita, A. Ueno, and M. Shimakawa, “Hagihara, and N. Yonezaki, “Safraless ltl synthesis considering maximal realizability,” Acta Informatica, vol. 54, no. 7, pp. 655–692, 2017.

[21] J. Tumova, L. I. R. Castro, S. Karaman, E. Frazzoli, and D. Rus, “Minimum-violation ltl planning with conflicting specifications,” in American Control Conference. IEEE, 2013, pp. 200–205.

[22] J. Tumova, G. C. Hall, S. Karaman, E. Frazzoli, and D. Rus, “Least-violating control strategy synthesis with safety rules,” in Proc. Int. Conf. on Hybrid systems: Computation and control, 2013.

[23] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, “Optimality and robustness in multi-robot path planning with temporal logic constraints,” The International Journal of Robotics Research, vol. 32, no. 8, pp. 880–911, 2013.

[24] M. Y. Vardi and P. Wolper, “An automata-theoretic approach to automatic program verification,” in Proc. First Symposium on Logic in Computer Science. IEEE Computer Society, 1986.

——, “Reasoning about infinite computations,” Information and Computation, vol. 115, no. 1, pp. 1–37, 1994.