

Sampling-Based Temporal Logic Path Planning

Cristian Ioan Vasile and Calin Belta

Abstract—In this paper, we propose a sampling-based motion planning algorithm that finds an infinite path satisfying a Linear Temporal Logic (LTL) formula over a set of properties satisfied by some regions in a given environment. The algorithm has three main features. First, it is incremental, in the sense that the procedure for finding a satisfying path at each iteration scales only with the number of new samples generated at that iteration. Second, the underlying graph is sparse, which guarantees the low complexity of the overall method. Third, it is probabilistically complete. Examples illustrating the usefulness and the performance of the method are included.

I. INTRODUCTION

Motion planning is a fundamental problem that has received a lot of attention from the robotics community [1]. The goal is to generate a feasible path for a robot to move from an initial to a final configuration while avoiding obstacles. Exact solutions to this problem are intractable, and relaxations using potential fields, navigation functions, and cell decompositions are commonly used [2]. These approaches, however, become prohibitively expensive in high dimensional configuration spaces. Sampling-based methods were proposed to overcome this limitation. Examples include the probabilistic roadmap (PRM) algorithm proposed by Kavraki et.al. in [3], which is very useful for multi-query problems, but is not well suited for the integration of differential constraints. In [4], Kuffner and LaValle proposed rapidly-exploring random trees (RRT). RRTs grow randomly, are biased to explore “new” space [4] (Voronoi bias), and find solutions quite fast. Moreover, PRM and RRT were shown to be probabilistically complete [3], [4], but not probabilistically optimal [5]. Karaman and Frazzoli proposed RRT* and PRM*, the probabilistically optimal counterparts of RRT and PRM in [5].

Recently, there has been increasing interest in improving the expressivity of motion planning specifications from the classical scenario (“Move from A to B and avoid obstacles.”) to richer languages that allow for Boolean and temporal requirements, e.g., “Visit A, then B, and then C, in this order infinitely often. Always avoid D unless E was visited.” It has been shown that temporal logics, such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), μ -calculus, and their probabilistic versions (PLTL, PCTL) [6] can be used as formal and expressive specification languages for robot motion [7], [8], [9], [10], [11]. In the above works, adapted model checking algorithms and automata

game techniques [7], [12] are used to generate motion plans and control policies for a finite model of robot motion, which is usually obtained through an abstraction process based on partitioning the configuration space [13]. The main limitation of these approaches is their high complexity, as both the synthesis and abstraction algorithms scale at least exponentially with the dimension of the configuration space.

To address this issue, in [10], Karaman and Frazzoli proposed a sampling-based path planning algorithm from specifications given in deterministic μ -calculus. However, deterministic μ -calculus formulae have unnatural syntax based on fixed point operators, and are difficult to use by untrained human operators. In contrast, Linear Temporal Logic (LTL), has very friendly syntax and semantics, which can be easily translated to natural language. One idea would be to translate LTL specifications to deterministic μ -calculus formulae and then proceed with generation of motion plans as in [10]. However, there is no known procedure to transform an LTL formula ϕ into a μ -calculus formula Ψ such that the size of Ψ is polynomial in the size of ϕ (for details see [14]).

In this paper, we propose a sampling-base path planning algorithm that finds an infinite path satisfying an LTL formula over a set of properties that hold at some regions in the configuration space. The procedure is based on the incremental construction of a transition system followed by the search for one of its satisfying paths. One important feature of the algorithm is that, at a given iteration, it only scales with the number of samples and transitions added to the transitions system at that iteration. This, together with a notion of “sparsity” that we define and enforce on the transition system, play an important role in keeping the overall complexity at a manageable level. In fact, we show that, under some mild assumptions, our definition of sparsity leads to the best possible complexity bound for finding a satisfying path. Finally, while the number of samples increases, the probability that a satisfying path is found approaches 1, i.e., our algorithm is probabilistically complete.

Among the above mentioned papers, the closest to this work is [10]. As in this paper, the authors of [10] can guarantee probabilistic completeness and scalability with added samples only at each iteration of their algorithm. However, in [10], the authors employ the fixed point (Knaster-Tarski) theorem to find a satisfying path. Their method is based on maintaining a “product” graph between the transition system and every sub-formula of their deterministic μ -calculus specification and checking for reachability and the existence of a “type” of cycle on the graph. On the other hand, our algorithm maintains the product automaton between the

This work was partially supported by the ONR under grants MURI N00014-09-1051 and MURI N00014-10-10952 and by the NSF under grant NSF CNS-1035588.

The authors are with the Division of Systems Engineering, Boston University, Boston MA, {cvasile, cbelta}@bu.edu.

transition system and a Büchi automaton corresponding to the given LTL specification. Note that, as opposed to LTL model checking [6], we use a modified version of product automaton that ensures reachability of the final states. Moreover, we impose that the states of the transition system be bounded away from each other (by a given function decaying in terms of the size of the transition system). Sparseness is also explored by Dobson and Berkis in [15] for PRM using different techniques.

Due to space limitations, the results in this paper are stated without proofs. The proofs and additional details can be found in [16].

II. PRELIMINARIES

For a finite set Σ , we use $|\Sigma|$ and 2^Σ to denote its cardinality and power set, respectively.

Definition 2.1 (Deterministic Transition System):

A deterministic transition system (DTS) is a tuple $\mathcal{T} = (X, x_0, \Delta, \Pi, h)$, where: X is a finite set of states, $x_0 \in X$ is the initial state, $\Delta \subseteq X \times X$ is a set of transitions, Π is a set of properties (atomic propositions), $h : X \rightarrow 2^\Pi$ is a labeling function.

We denote a transition $(x, x') \in \Delta$ by $x \rightarrow_{\mathcal{T}} x'$. A trajectory (or run) of the system is an infinite sequence of states $\mathbf{x} = x_0 x_1 \dots$ such that $x_k \rightarrow_{\mathcal{T}} x_{k+1}$ for all $k \geq 0$. A state trajectory \mathbf{x} generates an *output trajectory* $\mathbf{o} = o_0 o_1 \dots$, where $o_k = h(x_k)$ for all $k \geq 0$.

A Linear Temporal Logic (LTL) formula over a set of properties (atomic propositions) is defined using standard Boolean operators, \neg (negation), \wedge (conjunction) and \vee (disjunction), and temporal operators, **X** (next), **U** (until), **F** (eventually), **G** (always). The semantics of LTL formulae over Π are given with respect to infinite words over 2^Π , such as the output trajectories of the DTS defined above. Any infinite word satisfying a LTL formula can be written in the form of a finite prefix followed by infinitely many repetitions of a suffix. LTL formulae can be used to describe rich mission specifications. For example, formula $\mathbf{G}(\mathbf{F}(R_1 \wedge \mathbf{F}R_2) \wedge \neg O_1)$ specifies a persistent surveillance task: “visit regions R_1 and R_2 infinitely many times and always avoid obstacle O_1 ” (see Figure 1). Formal definitions for the LTL syntax, semantics, and model checking can be found in [6].

Definition 2.2 (Büchi Automaton): A (nondeterministic) Büchi automaton is a tuple $\mathcal{B} = (S_{\mathcal{B}}, S_{\mathcal{B}_0}, \Sigma, \delta, F_{\mathcal{B}})$, where: $S_{\mathcal{B}}$ is a finite set of states, $S_{\mathcal{B}_0} \subseteq S_{\mathcal{B}}$ is the set of initial states, Σ is the input alphabet, $\delta : S_{\mathcal{B}} \times \Sigma \rightarrow 2^{S_{\mathcal{B}}}$ is the transition function, $F_{\mathcal{B}} \subseteq S_{\mathcal{B}}$ is the set of accepting states.

A transition $(s, s') \in \delta(s, \sigma)$ is denoted by $s \xrightarrow{\sigma}_{\mathcal{B}} s'$. A trajectory of the Büchi automaton $s_0 s_1 \dots$ is generated by an infinite sequence of symbols $\sigma_0 \sigma_1 \dots$ if $s_0 \in S_{\mathcal{B}_0}$ and $s_k \xrightarrow{\sigma_k}_{\mathcal{B}} s_{k+1}$ for all $k \geq 0$. An input infinite sequence over Σ is said to be accepted by a Büchi automaton \mathcal{B} if it generates at least one trajectory of \mathcal{B} that intersects the set $F_{\mathcal{B}}$ of accepting states infinitely many times.

It is shown in [6] that for every LTL formula ϕ over Π there exists a Büchi automaton \mathcal{B} over alphabet $\Sigma = 2^\Pi$ such that \mathcal{B} accepts all and only those infinite sequences over Π

that satisfy ϕ . There exist efficient algorithms that translate LTL formulae into Büchi automata [17].

Model checking a DTS against an LTL formula is based on the construction of the product automaton between the DTS and the Büchi automaton corresponding to the formula. In this paper, we used a modified definition of the product automaton that is optimized for incremental search of a satisfying run. Specifically, the product automaton is defined such that all its states are reachable from the set of initial states.

Definition 2.3 (Product Automaton): Given a DTS $\mathcal{T} = (X, x_0, \Delta, \Pi, h)$ and a Büchi automaton $\mathcal{B} = (S_{\mathcal{B}}, S_{\mathcal{B}_0}, 2^\Pi, \delta_{\mathcal{B}}, F_{\mathcal{B}})$, their product automaton, denoted by $\mathcal{P} = \mathcal{T} \times \mathcal{B}$, is a tuple $\mathcal{P} = (S_{\mathcal{P}}, S_{\mathcal{P}_0}, \Delta_{\mathcal{P}}, F_{\mathcal{P}})$ where: $S_{\mathcal{P}_0} = \{x_0\} \times S_{\mathcal{B}_0}$ is the set of initial states; $S_{\mathcal{P}} \subseteq X \times S_{\mathcal{B}}$ is a finite set of states which are reachable from some initial state: for every $(x^*, s^*) \in S_{\mathcal{P}}$ there exists a sequence of $\mathbf{x} = x_0 x_1 \dots x_n x^*$, with $x_k \rightarrow_{\mathcal{T}} x_{k+1}$ for all $0 \leq k < n$ and $x_n \rightarrow_{\mathcal{T}} x^*$, and a sequence $\mathbf{s} = s_0 s_1 \dots s_n s^*$ such that $s_0 \in S_{\mathcal{B}_0}$, $s_k \xrightarrow{h(x_k)}_{\mathcal{B}} s_{k+1}$ for all $0 \leq k < n$ and $s_n \xrightarrow{h(x_n)}_{\mathcal{B}} s^*$; $\Delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ is the set of transitions, defined by: $((x, s), (x', s')) \in \Delta_{\mathcal{P}}$ iff $x \rightarrow_{\mathcal{T}} x'$ and $s \xrightarrow{h(x)}_{\mathcal{B}} s'$; $F_{\mathcal{P}} = (X \times F_{\mathcal{B}}) \cap S_{\mathcal{P}}$ is the set of accepting states.

A transition in \mathcal{P} is denoted by $(x, s) \rightarrow_{\mathcal{P}} (x', s')$ if $((x, s), (x', s')) \in \Delta_{\mathcal{P}}$. A trajectory $\mathbf{p} = (x_0, s_0)(x_1, s_1) \dots$ of \mathcal{P} is an infinite sequence, where $(x_0, s_0) \in S_{\mathcal{P}_0}$ and $(x_k, s_k) \rightarrow_{\mathcal{P}} (x_{k+1}, s_{k+1})$ for all $k \geq 0$. Such a trajectory is said to be accepting if and only if it intersects the set of final states $F_{\mathcal{P}}$ infinitely many times. It follows by construction that a trajectory $\mathbf{p} = (x_0, s_0)(x_1, s_1) \dots$ of \mathcal{P} is accepting if and only if the trajectory $s_0 s_1 \dots$ is accepting in \mathcal{B} . As a result, a trajectory of \mathcal{T} obtained from an accepting trajectory of \mathcal{P} satisfies the given specification encoded by \mathcal{B} . For $x \in X$, we define $\beta_{\mathcal{P}}(x) = \{s \in S_{\mathcal{B}} : (x, s) \in S_{\mathcal{P}}\}$ as the set of Büchi states that correspond to x in \mathcal{P} . Also, we denote the projection of a trajectory $\mathbf{p} = (x_0, s_0)(x_1, s_1) \dots$ onto \mathcal{T} by $\gamma_{\mathcal{T}}(\mathbf{p}) = x_0 x_1 \dots$. A similar notation is used for projections of finite trajectories.

For both DTS and automata, we use $|\cdot|$ to denote size, which is the cardinality of the corresponding set of states. A state of a DTS or an automaton is called non-blocking if it has at least one outgoing transition.

III. PROBLEM FORMULATION AND APPROACH

Let $\mathcal{D} \subset \mathbb{R}^n$ be a compact set denoting the configuration space of a robot. Let \mathcal{R} be a set of disjoint regions in \mathcal{D} and Π be a set of properties of interest corresponding to these regions. A map $\sim : \mathcal{R} \rightarrow 2^\Pi$ specifies how properties are associated to the regions. Throughout this paper, we will assume that \mathcal{R} is composed of connected sets with non-empty interior, which implies they have non-zero Lebesgue measure (i.e. all regions of interest have full dimension). Also, all connected sets in $\mathcal{D} \setminus \bigcup_{R \in \mathcal{R}} R$ have full dimension. Examples of properties include “obstacle”, “target”, “drop-off”, etc. (see Fig. 1).

Problem 3.1: Given an environment described by $(\mathcal{D}, \mathcal{R}, \Pi, \sim)$, the initial configuration of the robot $x_0 \in \mathcal{D}$ and an LTL formula ϕ over the set of properties Π , find a satisfying (infinite) path for the robot originating at x_0 .

A possible approach to Problem 3.1 is to construct a partition of the configuration space that contains the regions of interest as elements of the partition. By using input - output linearizations and vector field assignments in the regions of the partition, it was shown that “equivalent” abstractions in the form of finite (not necessarily deterministic) transition systems can be constructed for a large variety of robot dynamics that include car-like vehicles and quadrotors [13], [18], [19]. Model checking and automata game techniques can then be used to control the abstractions from the temporal logic specification [20]. The main limitation of this approach is its high complexity, as both the synthesis and abstraction algorithms scale at least exponentially with the dimension of the configuration space.

In this paper, we propose a sampling-based approach that can be summarized as follows: (1) the LTL formula ϕ is translated to the Büchi automaton \mathcal{B} ; (2) a transition system \mathcal{T} is incrementally constructed from the initial position x_0 using an RRG-based algorithm; (3) concurrently with (2), the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{B}$ is updated and used to check if there is a trajectory of \mathcal{T} that satisfies ϕ . As it will become clear later, our proposed algorithm is *probabilistically complete* [1], [5] (i.e., it finds a solution with probability 1 if one exists and the number of samples approaches infinity) and the resulting transition system is *sparse* (i.e., its states are “far” away from each other).

IV. PROBLEM SOLUTION

The starting point for our solution to Problem 3.1 is the RRG algorithm, which is an extension of RRT [5] that maintains a digraph instead of a tree, and can therefore be used as a model for general ω -regular languages [10]. However, we modify the RRG to obtain a “sparse” transition system that satisfies a given LTL formula. More precisely, a transition system \mathcal{T} is “sparse” if the minimum distance between any two states of \mathcal{T} is greater than a prescribed function dependent only on the size of \mathcal{T} ($\min_{x, x' \in \mathcal{T}} \|x - x'\|_2 \geq \eta(|\mathcal{T}|)$). The distance used to define sparsity is inherited from the underlying configuration space and is not related to the graph theoretical distance between states in \mathcal{T} . Throughout this paper, we will assume that this distance is Euclidean.

A. Sparse RRG

We first briefly introduce the functions used by the algorithm.

a) Sampling function: The algorithm has access to a sampling function $Sample : \mathbb{N} \rightarrow \mathcal{D}$, which generates independent and identically distributed samples from a given distribution P . We assume that the support of P is the entire configuration space \mathcal{D} .

b) Steer function: The steer function $Steer : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is defined based on the robot’s dynamics.¹ Given a configuration x and goal configuration x_g , it returns a new configuration x_n that can be reached from x by following the dynamics of the robot and that satisfies $\|x_n - x_g\|_2 < \|x - x_g\|_2$.

c) Near function: $Near : \mathcal{D} \times \mathbb{R} \rightarrow 2^{\mathcal{X}}$ is a function of a configuration x and a parameter η , which returns the set of states from the transition system \mathcal{T} that are at most at η distance away from x . In other words, $Near$ returns all states in \mathcal{T} that are inside the n -dimensional sphere of center x and radius η .

d) Far function: $Far : \mathcal{D} \times \mathbb{R} \times \mathbb{R} \rightarrow 2^{\mathcal{X}}$ is a function of a configuration x and two parameters η_1 and η_2 . It returns the set of states from the transition system \mathcal{T} that are at most at η_2 distance away from x . However, the difference from the $Near$ function is that Far returns an empty set if any state of \mathcal{T} is closer to x than η_1 . Thus, x has to be “far” away from all states in its immediate neighborhood (see Figure 1). This function is used to achieve the “sparseness” of the resulting transition system.

e) isSimpleSegment function: $isSimpleSegment : \mathcal{D} \times \mathcal{D} \rightarrow \{0, 1\}$ is a function that takes two configurations x_1, x_2 in \mathcal{D} and returns 1 if the line segment $[x_1, x_2]$ ($\{x \in \mathbb{R}^n : x = \lambda x_1 + (1 - \lambda)x_2, \lambda \in [0, 1]\}$) is simple, otherwise it returns 0. A line segment $[x_1, x_2]$ is simple if $[x_1, x_2] \subset \mathcal{D}$ and the number of times $[x_1, x_2]$ crosses the boundary of any region $R \in \mathcal{R}$ is at most one. Therefore, $isSimpleSegment$ returns 1 if either: (1) x_1 and x_2 belong to the same region R and $[x_1, x_2]$ does not cross the boundary of R or (2) x_1 and x_2 belong to two regions R_1 and R_2 , respectively, and $[x_1, x_2]$ crosses the common boundary of R_1 and R_2 once. R or at most one of R_1 and R_2 may be a free space region (a connected set in $\mathcal{D} \setminus \bigcup_{R \in \mathcal{R}} R$). See Figure 1 for examples. In Algorithm 1, a transition is rejected if it corresponds to a non-simple line segment (i.e. $isSimpleSegment$ function returns 0). Under this condition, the satisfaction of the mission specification can be checked by only looking at the properties corresponding to the states of the transition system.

f) Bound functions: $\eta_1 : \mathbb{Z}_+ \rightarrow \mathbb{R}$ (lower bound) and $\eta_2 : \mathbb{Z}_+ \rightarrow \mathbb{R}$ (upper bound) are functions that define the bounds on the distance between a configuration in \mathcal{D} and the states of the transition system \mathcal{T} in terms of the size of \mathcal{T} . These are used as parameters for functions Far and $Near$. We impose $\eta_1(k) < \eta_2(k)$ for all $k \geq 1$. We also assume that $c\eta_1(k) > \eta_2(k)$, for some finite $c > 1$ and all $k \geq 0$. Also, η_1 tends to 0 as k tends to infinity. The rate of decay of $\eta_1(\cdot)$ has to be fast enough such that a new sample may be generated. Specifically, the set of all configurations where the center of an n -sphere of radius $\eta_1/2$ may be placed such that it does not intersect any of the n -spheres corresponding to the states in \mathcal{T} has to have non-zero measure with respect to the probability measure P used by the sampling function.

¹In this paper, we will assume that we have access to such a function. For more details about planning under differential constraints see [1].

To simplify the notation, we drop the parameter for these functions and assume that k is always given by the current size of the transition system, $k = |\mathcal{T}|$.

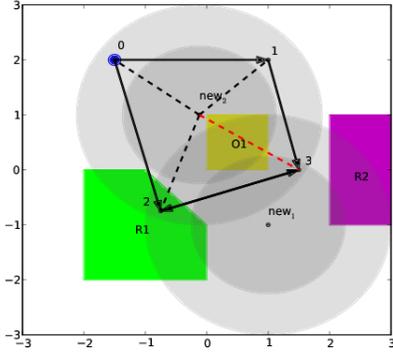


Fig. 1. A simple map with three features: an obstacle O_1 and two regions R_1 , R_2 . The robot is assumed to be a fully actuated point. At the current iteration the states of \mathcal{T} are $\{0, 1, 2, 3\}$. The transitions of \mathcal{T} are represented by the black arrows. The initial configuration is 0 and is marked by the blue disk. The radii of the dark gray (inner) disks and the light gray (outer) disks are η_1 and η_2 , respectively. A new sample $new_1 \in \mathcal{D}$ is generated, but it will not be considered as a potential new state of \mathcal{T} , because it is within η_1 distance from state 3 ($Far(new_1, \eta_1, \eta_2) = \emptyset$). Another sample $new_2 \in \mathcal{D}$ is generated, which is at least η_1 distance away from all states in \mathcal{T} . In this case, $Far(new_2, \eta_1, \eta_2) = \{0, 1, 2, 3\}$ and the algorithm attempts to create transition to and from the new sample new_2 . The transitions $\{(new_2, 0), (0, new_2), (new_2, 1), (1, new_2), (new_2, 2), (2, new_2)\}$ (marked by black dashed lines) are added to \mathcal{T} , because all these transitions correspond to simple line segments ($isSimpleSegment$ returns 1 for all of them). On the other hand, the transitions $\{(new_2, 3), (3, new_2)\}$ (marked by red dashed lines) are not added to \mathcal{T} , since they pass over the obstacle O_1 .

The goal of the modified RRG algorithm (see Algorithm 1) is to find a satisfying run, but such that the resulting transition system is “sparse”, i.e. states are “sufficiently” apart from each other. The algorithm iterates until a satisfying run originating in x_0 is found.

At each iteration, a new sample x_r is generated (line 6 in Algorithm 1). For each state x in \mathcal{T} which is “far” from the sample x_r ($x \in Far(x_r, \eta_1, \eta_2)$), a new configuration x'_r is computed such that the robot can be steered from x to x'_r and the distance to x_r is decreased (line 10). The two loops of the algorithm (lines 7–13 and 16–21) are executed if and only if the Far function returns a non-empty set. However, x'_r is regarded as a potential new state of \mathcal{T} , and not x_r . Thus, the $Steer$ function plays an important role in the “sparsity” of the final transition system. Next, it is checked if the potential new transition (x, x'_r) is a simple segment (line 9). It is also verified if x'_r may lead to a solution, which is equivalent to testing if x'_r induces at least one non-blocking state in \mathcal{P} (see Algorithm 2). If configuration x'_r and the corresponding transition (x, x'_r) pass all tests, then they are added to the list of new states and list of new transitions of \mathcal{T} , respectively (lines 12–13).

After all “far” neighbors of x_r are processed, the transition system is updated. Note that at this point \mathcal{T} was only extended with states that explore “new space”. However, in

order to model ω -regular languages the algorithm must also close cycles. Therefore, the same procedure as before (lines 7–14) is also applied to the newly added states X' (lines 15–21 of Algorithm 1). The difference is that it is checked if states from X' can steer the robot back to states in \mathcal{T} in order to close cycles. Also, because we know that the states in X' are “far” from their neighbors, the $Near$ function will be used instead of the Far function. The algorithm returns a (prefix, suffix) pair in \mathcal{T} obtained by projection from the corresponding path $(p_0 \xrightarrow{*} p_F)$ and cycle $(p_F \xrightarrow{+} p_F)$ in \mathcal{P} , respectively. The $*$ above the transition symbol means that the length of the path can be 0, while $+$ denotes that the length of the cycle must be at least 1.

Algorithm 1: Sparse RRG

Input: \mathcal{B} – Büchi automaton corresponding to ϕ
Input: x_0 initial configuration of the robot
Output: (prefix, suffix) in \mathcal{T}

- 1 Construct \mathcal{T} with x_0 as initial state
- 2 Construct $\mathcal{P} = \mathcal{T} \times \mathcal{B}$
- 3 Initialize $scc(\cdot)$
- 4 **while** $\neg(x_0 \models \phi)$ ($\equiv \neg(\exists p \in F_{\mathcal{P}} \text{ s.t. } |scc(p)| > 1)$) **do**
- 5 $X' \leftarrow \emptyset, \Delta' \leftarrow \emptyset, \Delta'_{\mathcal{P}} \leftarrow \emptyset$
- 6 $x_r \leftarrow Sample()$
- 7 **foreach** $x \in Far(x_r, \eta_1, \eta_2)$ **do**
- 8 $x'_r \leftarrow Steer(x, x_r)$
- 9 **if** $isSimpleSegment(x, x'_r)$ **then**
- 10 $added \leftarrow updatePA(\mathcal{P}, \mathcal{B}, (x, x'_r))$
- 11 **if** $added$ is True **then**
- 12 $X' \leftarrow X' \cup \{x'_r\}$
- 13 $\Delta' \leftarrow \Delta' \cup \{(x, x'_r)\}$
- 14 $\mathcal{T} \leftarrow \mathcal{T} \cup (X', \Delta')$
- 15 $\Delta' \leftarrow \emptyset, \Delta'_{\mathcal{P}} \leftarrow \emptyset$
- 16 **foreach** $x'_r \in X'$ **do**
- 17 **foreach** $x \in Near(x'_r, \eta_2)$ **do**
- 18 **if** $(x = Steer(x'_r, x)) \wedge isSimpleSegment(x'_r, x)$
- 19 **then**
- 20 $added \leftarrow updatePA(\mathcal{P}, \mathcal{B}, (x, x'_r))$
- 21 **if** $added$ is True **then**
- 22 $\Delta' \leftarrow \Delta' \cup \{(x, x'_r)\}$
- 22 $\mathcal{T} \leftarrow \mathcal{T} \cup (X', \Delta')$
- 23 **return** $(\gamma_{\mathcal{T}}(p_0 \xrightarrow{*} p_F), \gamma_{\mathcal{T}}(p_F \xrightarrow{+} p_F))$, where $p_F \in F_{\mathcal{P}}$

B. Incremental search for a satisfying run

The proposed approach of incrementally constructing a transition system raises the problem of how to efficiently check for a satisfying run at each iteration. As mentioned in the previous section, the search for satisfying runs is performed on the product automaton. Note that testing whether there exists a trajectory of \mathcal{T} from the initial position x_0 that satisfies the given LTL formula ϕ is equivalent to searching for a path from an initial state p_0 to a final state p_F in the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{B}$ and for a cycle containing p_F of length greater than 1, where \mathcal{B} is the Büchi automaton corresponding to ϕ . If such a path and a cycle are found then their projection onto \mathcal{T} represents a satisfying infinite trajectory (line 23 of Algorithm 1). Testing whether p_F belongs to a non-degenerate cycle (length greater than

1) is equivalent to testing if p_F belong to a non-trivial strongly connected component – SCC (the size of the SCC is greater than 1). Checking for a satisfying trajectory in \mathcal{P} is performed incrementally as the transition system is modified.

The reachability of the final states from initial ones in \mathcal{P} is guaranteed by construction (see Definition 2.3). However, we need to define a procedure (see Algorithm 2) to incrementally update \mathcal{P} when a new transition is added to \mathcal{T} . Consider the (non-incremental) case of constructing $\mathcal{P} = \mathcal{T} \times \mathcal{B}$. This is done by a traversal of $\bar{\mathcal{P}} = (X \times S_{\mathcal{B}}, \bar{\Delta}_{\mathcal{P}})$ from all initial states, where $((x, s), (x', s')) \in \bar{\Delta}_{\mathcal{P}}$ if $p \rightarrow p'$ and $s \xrightarrow{h(x)} s'$. $\bar{\mathcal{P}}$ is a product automaton but without the reachability requirement. This suggests that the way to update \mathcal{P} when a transition (x, x') is added to \mathcal{T} , is to do a traversal from all states p of \mathcal{P} such that $\gamma_{\mathcal{T}}(p) = x$. Also, it is checked if x' induces any non-blocking states in \mathcal{P} (lines 1-3 of Algorithm 2). The test is performed by computing the set $S'_{\mathcal{P}}$ of non-blocking states of \mathcal{P} (line 1) such that $p' \in S'_{\mathcal{P}}$ has $\gamma_{\mathcal{T}}(p') = x'$ and p' is obtained by a transition from $\{(x, s) : s \in \beta_{\mathcal{P}}(x)\}$. If $S'_{\mathcal{P}}$ is empty then the transition (x, x') of \mathcal{T} is discarded and the procedure stops (line 3). Otherwise, the product automaton \mathcal{P} is updated recursively to add all states that become reachable because of the states in $S'_{\mathcal{P}}$. The recursive procedure is performed from each state in $S'_{\mathcal{P}}$ as follows: if a state p (line 7) is not in \mathcal{P} , then it is added to \mathcal{P} together with all its outgoing transitions (line 10) and the recursive procedure continues from the outgoing states of p ; if p is in \mathcal{P} then the traversal stops, but its outgoing transitions are still added to \mathcal{P} (line 14). The incremental construction of \mathcal{P} has the same overall complexity as constructing \mathcal{P} from the final \mathcal{T} and \mathcal{B} , because the recursive procedure just performs traversals that do not visit states already in \mathcal{P} . Thus, we focus our complexity analysis on the next step of the incremental search algorithm.

The second part of the incremental search procedure is concerned with maintaining the strongly connected components (SCCs) of \mathcal{P} (line 14 of Algorithm 2) as new transitions are added (these are stored in $\Delta'_{\mathcal{P}}$ in Algorithm 2). To incrementally maintain the SCCs of the product automaton, we employ the soft-threshold-search algorithm presented in [21]. The algorithm maintains a topological order of the super-vertices corresponding to each SCC. When a new transition is added to \mathcal{P} , the algorithm proceeds to re-establish a topological order and merges vertices if new SCCs are formed. The details of the algorithm are presented in [21]. The authors also offer insight about the complexity of the algorithm. They show that, under a mild assumption, the incremental algorithm has the best possible complexity bound.

Theorem 4.1: The execution time of the incremental search algorithm 2 is $O(m^{\frac{3}{2}})$, where m is the number of transitions added to \mathcal{T} in Algorithm 1.

Remark 4.2: Note that the execution time of the incremental procedure is better by a polynomial factor than naively running a linear-time SCC algorithm at each step, since this

Algorithm 2: Incremental Search for a Satisfying Run

Input: \mathcal{P} – product automaton
Input: \mathcal{B} – Büchi automaton
Input: (x, x') – new transition in \mathcal{T}
Output: Boolean value – indicates if \mathcal{P} was modified

```

1  $S'_{\mathcal{P}} \leftarrow \{(x', s') : s \xrightarrow{h(x)}_{\mathcal{B}} s', s \in \beta_{\mathcal{P}}(x), s' \text{ non-blocking}\}$ 
2  $\Delta'_{\mathcal{P}} \leftarrow \{((x, s), (x', s')) : s \in \beta_{\mathcal{P}}(x), s \xrightarrow{h(x)}_{\mathcal{B}} s', (x', s') \in S'_{\mathcal{P}}\}$ 
3 if  $S'_{\mathcal{P}} \neq \emptyset$  then
4    $\mathcal{P} \leftarrow \mathcal{P} \cup (S'_{\mathcal{P}}, \Delta'_{\mathcal{P}})$ 
5    $stack \leftarrow S'_{\mathcal{P}}$ 
6   while  $stack \neq \emptyset$  do
7      $p_1 = (x_1, s_1) \leftarrow stack.pop()$ 
8     foreach  $p_2 \in \{(x_2, s_2) : x_1 \rightarrow_{\mathcal{T}} x_2, s_1 \xrightarrow{h(x_1)}_{\mathcal{B}} s_2\}$  do
9       if  $p_2 \notin S_{\mathcal{P}}$  then
10         $\mathcal{P} \leftarrow \mathcal{P} \cup (\{p_2\}, \{(p_1, p_2)\})$ 
11         $\Delta'_{\mathcal{P}} \leftarrow \Delta'_{\mathcal{P}} \cup \{(p_1, p_2)\}$ 
12         $stack \leftarrow stack \cup \{p_2\}$ 
13       else if  $(p_1, p_2) \notin \Delta_{\mathcal{P}}$  then
14         $\Delta_{\mathcal{P}} \leftarrow \Delta_{\mathcal{P}} \cup \{(p_1, p_2)\}$ 
15         $\Delta'_{\mathcal{P}} \leftarrow \Delta'_{\mathcal{P}} \cup \{(p_1, p_2)\}$ 
16   updateSCC( $\mathcal{P}, scc, \Delta'_{\mathcal{P}}$ )
17   return True
18 return False
```

will have complexity $O(m^2)$.

The presented RRG-based algorithm retains the probabilistic completeness of RRT, since the constructed transition system is composed of an RRT-like tree and some transitions which close cycles.

Theorem 4.3: Algorithm 1 is probabilistically complete.

V. IMPLEMENTATION AND CASE STUDIES

We implemented the algorithms presented in this paper in Python2.7. In this section, we present some examples in configuration spaces of dimensions 2, 10 and 20. In all cases, we assume for simplicity that the *Steer* function is trivial, i.e., there are no actuation constraints at any given configuration. All examples were ran on an iMac system with a 3.4 GHz Intel Core i7 processor and 16GB of memory.

Case Study 1: Consider the configuration space depicted in Figure 2. The initial configuration is at $(0.3; 0.3)$. The specification is to visit regions r_1, r_2, r_3 and r_4 infinitely many times while avoiding regions o_1, o_2, o_3 and o_4 . The corresponding LTL formula for the given mission specification is $\phi_1 = \mathbf{G}(\mathbf{F}r_1 \wedge (\mathbf{F}r_2 \wedge (\mathbf{F}r_3 \wedge (\mathbf{F}r_4))) \wedge \neg(o_1 \vee o_2 \vee o_3 \vee o_4))$.

A solution to this problem is shown in Figures 2 and 3. We ran the overall algorithm 20 times and obtained an average execution time of 6.954 sec, out of which the average of the incremental search algorithm was 6.438 sec. The resulting transition system had a mean size of 51 states and 277 transitions, while the corresponding product automaton had a mean size of 643 states and 7414 transitions. The Büchi automaton corresponding to ϕ_1 had 20 states and 155 transitions.

Case Study 2: Consider a 10-dimensional unit hypercube configuration space. The specification is to visit regions r_1, r_2, r_3 infinitely many times, while avoiding region o_1 . The

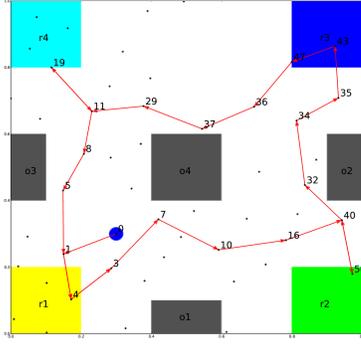


Fig. 2. One of the solutions corresponding to Case Study 1: the specification is to visit all the colored regions labelled $r1$ (yellow), $r2$ (green), $r3$ (blue) and $r4$ (cyan) infinitely often, while avoiding the dark gray obstacles labelled $o1$, $o2$, $o3$, $o4$. The black dots represent the states of the transition system \mathcal{T} (51 states and 264 transitions). The starting configuration of the robot (the initial state of \mathcal{T}) is denoted by the blue circle. The red arrows represent the satisfying run (finite prefix, suffix pair) found by Algorithm 1, which is composed of 21 states from \mathcal{T} . In this case, the prefix and suffix are $[0, 1, 4, 3]$ and $[7, 10, 16, 40, 50, 40, 32, 34, 35, 43, 47, 36, 37, 29, 11, 19, 11, 8, 5, 1, 4, 3]$, respectively.

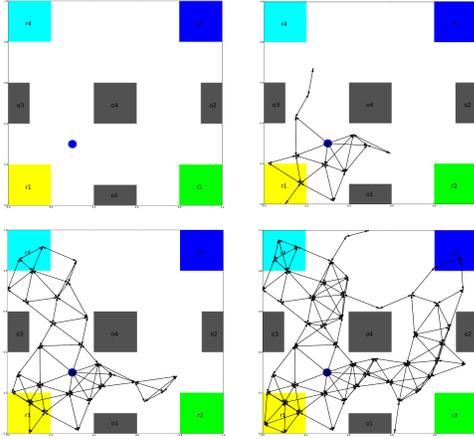


Fig. 3. Transition systems obtained at earlier iterations corresponding to the solution shown in Figure 2 (to be read from left to right and top to bottom). The black dots and arrows represent the state and transitions of \mathcal{T} , respectively.

LTL formula corresponding to this specification is $\phi_2 = \mathbf{G}(\mathbf{F}r1 \wedge (\mathbf{F}r2 \wedge (\mathbf{F}r3)) \wedge \neg o1)$.

The corresponding Büchi automaton has 9 states and 43 transitions. Regions $r1$, $r2$, $r3$ and $o1$ are hypercubes and their volumes are 0.03, 0.03, 0.013 and 0.012, respectively. $r1$, $r2$, $r3$ are positioned in the corners of the configuration space, while $o1$ is positioned in the center. In this case, the algorithm took 16.75 sec on average (20 experiments), while just the incremental search procedure for a satisfying run took 14.471 sec. The transition system had a mean size of 69 states and 1578 transitions, while the product automaton had a mean size of 439 states and 21300 transitions.

Case Study 3: We also considered a 20-dimensional unit hypercube configuration space. Two hypercube regions $r1$ and $r2$ were defined and the robot was required to visit both of them infinitely many times ($\phi_3 = \mathbf{G}(\mathbf{F}(r1 \wedge \mathbf{F}r2))$).

The overall algorithm took 7.45 minutes, while the transition system grew to 414 states and 75584 transitions. The corresponding product automaton had a size of 1145 states and 425544 transitions.

REFERENCES

- [1] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at <http://planning.cs.uiuc.edu/>.
- [2] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Boston, MA: MIT Press, 2005.
- [3] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.
- [4] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," in *IEEE International Conference on Robotics and Automation*, 1999, pp. 473–479.
- [5] S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.
- [6] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.
- [7] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Where's Waldo? Sensor-based temporal logic motion planning," in *IEEE International Conference on Robotics and Automation*, 2007, pp. 3116–3121.
- [8] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding Horizon Temporal Logic Planning for Dynamical Systems," in *Conference on Decision and Control (CDC) 2009*, 2009, pp. 5997–6004.
- [9] A. Bhatia, L. Kavraki, and M. Vardi, "Sampling-based motion planning with temporal goals," in *Robotics and Automation (ICRA), IEEE International Conference on*. IEEE, 2010, pp. 2689–2696.
- [10] S. Karaman and E. Frazzoli, "Sampling-based Motion Planning with Deterministic μ -Calculus Specifications," in *IEEE Conference on Decision and Control (CDC)*, Shanghai, China, December 2009.
- [11] X. C. Ding, M. Kloetzer, Y. Chen, and C. Belta, "Formal Methods for Automatic Deployment of Robotic Teams," *IEEE Robotics and Automation Magazine*, vol. 18, pp. 75–86, 2011.
- [12] Y. Chen, J. Tumova, and C. Belta, "LTL Robot Motion Control based on Automata Learning of Environmental Dynamics," in *IEEE International Conference on Robotics and Automation (ICRA)*, Saint Paul, MN, USA, 2012.
- [13] C. Belta, V. Isler, and G. J. Pappas, "Discrete abstractions for robot planning and control in polygonal environments," *IEEE Trans. on Robotics*, vol. 21, no. 5, pp. 864–874, 2005.
- [14] S. Cranen, J. Groote, and M. Reniers, "A linear translation from LTL to the first-order modal μ -calculus," *Technical Report 10-09, Computer Science Reports*, 2010.
- [15] A. Dobson and K. E. Bekris, "Improving Sparse Roadmap Spanners," in *IEEE International Conference on Robotics and Automation (ICRA)*, March 2013.
- [16] C.-I. Vasile and C. Belta, "Sampling-Based Temporal Logic Path Planning," 2013, available at: <http://arxiv.org/abs/1307.7263>.
- [17] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer, jul 2001, pp. 53–65.
- [18] S. R. Lindemann and S. M. LaValle, "Simple and Efficient Algorithms for Computing Smooth, Collision-Free Feedback Laws Over Given Cell Decompositions," *International Journal of Robotics Research*, vol. 28, no. 5, pp. 600–621, 2009.
- [19] A. Ulusoy, M. Marrazzo, K. Oikonomopoulos, R. Hunter, and C. Belta, "Temporal Logic Control for an Autonomous Quadrotor in a Nondeterministic Environment," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.
- [20] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Transactions on Automatic Control*, vol. 53, no. 1, pp. 287–297, 2008.
- [21] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan, "Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 3:1–3:33, Jan. 2012.