Robotics: Science and Systems 2012
Sydney, NSW, Australia, July 09-13, 2012

Optimal Control with Weighted Average Costs and Temporal Logic Specifications

Eric M. Wolff
Control and Dynamical Systems
California Institute of Technology
Pasadena, California 91125
Email: ewolff@caltech.edu

Ufuk Topcu
Control and Dynamical Systems
California Institute of Technology
Pasadena, California 91125
Email: utopcu@cds.caltech.edu

Richard M. Murray
Control and Dynamical Systems
California Institute of Technology
Pasadena, California 91125
Email: murray@cds.caltech.edu

***Abstract*—We consider optimal control for a system subject to temporal logic constraints. We minimize a weighted average cost function that generalizes the commonly used average cost function from discrete-time optimal control. Dynamic programming algorithms are used to construct an optimal trajectory for the system that minimizes the cost function while satisfying a temporal logic specification. Constructing an optimal trajectory takes only polynomially more time than constructing a feasible trajectory. We demonstrate our methods on simulations of autonomous driving and robotic surveillance tasks.**

I. Introduction

As the level of autonomy expected of robots, vehicles, and other cyberphysical systems increases, there is a need for expressive task-specification languages that can encode desired behaviors. Temporal logics such as linear temporal logic (LTL) are promising formal languages for robotics. LTL provides a natural framework to specify desired properties such as response (if A, then B), liveness (always eventually A), safety (always not B), stability (eventually always A), and priority (first A, then B, then C).

Temporal logics have been used in the robotics and control communities to reason about system properties. A framework for automatically generating controllers for linear systems that satisfy a given LTL specification is presented in [16]. Sampling-based approaches for more general dynamical systems are given in [11, 19]. Controllers can be constructed that satisfy a specification in the presence of an adversarial environment [17], and receding horizon control can reduce the resulting computational complexity of synthesizing such control policies [23]. While these approaches all generate feasible control policies that satisfy a temporal logic specification, no practical optimality notions can be imposed in their settings.

Often there are numerous control policies for a system that satisfy a given temporal logic specification, so it is desirable to select one that is optimal with respect to some cost function, e.g., time or fuel consumption. Since temporal logic specifications include properties that must be satisfied over infinite state sequences, it is important that the form of the cost function is also well-defined over infinite sequences. We consider an average cost, which is bounded under certain mild assumptions discussed in Section 1. Additionally, it may be desired to give varying weights to different behaviors, i.e., repeatedly visit set of regions, but visit a high-weight region more often than others. Thus, we minimize a weighted average cost function over system trajectories subject to the constraint that a given temporal logic specification is satisfied. This cost function generalizes the average cost-per-stage cost function commonly studied in discrete-time optimal control [4].

Optimality has been considered in the related area of vehicle routing [21]. Vehicle routing problems generalize the traveling salesman problem, and are thus NP-complete. A different approach to control with LTL specifications converts the controller design problem into a mixed-integer linear program [12]. However, this approach is restricted to properties specified over a finite horizon. Chatterjee et al. [5] create control policies that minimize an average cost function in the presence of an adversary. The approach in [20] is the most closely related to our work. Motivated by surveillance tasks, they minimize the maximum cost between visiting specific regions. Our work is complementary to [20] in that we instead minimize a weighted average cost function.

The main contribution of this paper is a solution to the problem of, given a transition system model, creating a system trajectory that minimizes a weighted average cost function subject to temporal logic constraints. We solve this problem by searching for system trajectories in the product automaton, a lifted space that contains only behaviors that are valid for the transition system and also satisfy the temporal logic specification. An optimal system trajectory corresponds to a cycle in the product automaton, which is related to the well-studied cost-to-time ratio problem in operations research. We give computationally efficient dynamic programming algorithms for finding the optimal system trajectory. In fact, it takes only polynomially more effort to calculate an optimal solution than a feasible solution, i.e., one that just satisfies the specification.

We present preliminaries on system models, task-specification languages, and graph theory in Section II. The main problem is formulated in Section III and reformulated as an equivalent graph problem in Section IV. We solve this equivalent problem for finite-memory and infinite-memory policies in Sections V-A and V-B respectively. These techniques are demonstrated on numerical examples motivated by autonomous driving and surveillance tasks in Section VI. We conclude with future directions in Section VII.

II. PRELIMINARIES

We now provide preliminaries on the modeling and specification languages, weighted transition systems and linear temporal logic respectively, used throughout the paper.

An *atomic proposition* is a statement that has a unique truth value ($True$ or $False$).

A. Modeling Language

We use finite transition systems to model the system behavior. In robotics, however, one is usually concerned with continuous systems that may have complex dynamic constraints. This gap is partially bridged by constructive procedures for exactly abstracting relevant classes of continuous systems, including unicycle models, as finite transition systems [3, 2, 9]. Additionally, sampling-based methods, such as rapidly-exploring random trees [18] and probabilistic roadmaps [15], gradually build a finite transition system that approximates a continuous system, and have been studied in this context [11, 19]. Examples of how one can abstract continuous dynamics by discrete transition systems are given in [2, 3, 9, 11, 19].

Definition 1. A *weighted (finite) transition system* is a tuple $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ consisting of (i) a finite set of states S, (ii) a transition relation $R \subseteq S \times S$, (iii) an initial state $s_0 \in S$, (iv) a set of atomic propositions AP, (v) a labeling function $L : S \to 2^{AP}$, (vi) a cost function $c : R \to \mathbb{R}$, (vii) and a weight function $w : R \to \mathbb{R}_{\geq 0}$.

We assume that the transition system is non-blocking, so for each state $s \in S$, there exists a state $t \in S$ such that $(s, t) \in R$.

A *run* of the transition system is an infinite sequence of its states, $\sigma = s_0 s_1 s_2 \ldots$ where $s_i \in S$ is the state of the system at index i (also denoted σ_i) and $(s_i, s_{i+1}) \in R$ for $i = 0, 1, \ldots$. A *word* is an infinite sequence of labels $L(\sigma) = L(s_0)L(s_1)L(s_2)\ldots$ where $\sigma = s_0 s_1 s_2 \ldots$ is a run.

Let S_k^+ be the set of all runs up to index k. An *infinite-memory* control policy is denoted by $\pi = (\mu_0, \mu_1, \ldots)$ where $\mu_k : S_k^+ \to R$ maps a partial run $s_0 s_1 \ldots s_k \in S_k^+$ to a new transition. A policy $\pi = (\mu, \mu, \ldots)$ is *finite-memory* if $\mu : S \times \mathcal{M} \to R \times \mathcal{M}$, where the finite set $\mathcal{M}$ is called the memory.

For the deterministic transition system models we consider, the run of a transition system implicitly encodes the control policy. An *infinite-memory* run is a run that can be implemented by an infinite-memory policy. Similarly for finite-memory runs.

When possible, we will sometimes abuse notation and refer to the cost $c(t)$ of a state $t \in S$, instead of a transition between states. In this case, we enforce that $c(s, t) = c(t)$ for all transitions $(s, t) \in R$, i.e., the cost of the state is mapped to all incoming transitions. Similar notational simplification is used for weights and should be clear from context.

The cost function c can be viewed concretely as a physical cost of a transition between states, such as time or fuel. This cost can be negative for some transitions, which could, for example, correspond to refueling if the cost is fuel consumption. The weight function w can be viewed as the importance of each transition, which is a flexible design parameter. In the sequel, we will create a run with the minimal weighted average cost. Thus, the designer can give transitions that she thinks are preferable a higher weight than the rest. As an example, consider an autonomous car that is supposed to visit different locations in a city while obeying the rules-of-the-road. In this case, a task specification would encode the locations that should be visited and the rules-of-the-road. Costs might be the time required to traverse different roads. Weights might encode preferences such as visiting certain landmarks. An example scenario is discussed in detail in Section VI.

B. Specification Language

We are interested in using linear temporal logic (LTL) to concisely and unambiguously specify desired system behavior. LTL is a powerful formal language that is relevant to robotics because it allows system behaviors such as response, liveness, safety, stability, priority, and guarantee to be specified.

However, the syntax and semantics of LTL are not relevant for the theory developed in this paper, so we only mention them as needed for the examples in Section VI. The interested reader can find a comprehensive treatment of LTL in [1]. Instead, we follow the automata-based approach of Vardi and Wolper [22], and consider non-deterministic Buchi automata (hereafter called Buchi automata), which accept the class of languages equivalent to ω-regular languages. Thus, our results hold for any property that can be specified as an ω-regular language, which is a regular language extended by infinite repetition (denoted by ω). In particular, LTL is a subset of ω-regular languages, so an equivalent Buchi automaton can be constructed for any LTL formula φ [1].

Definition 2. A Buchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \text{Acc})$ consisting of (i) a finite set of states Q, (ii) a finite alphabet Σ, (iii) a transition relation $\delta \subseteq Q \times \Sigma \times Q$, (iv) a set of initial states $Q_0 \subseteq Q$, (v) and a set of accepting states $\text{Acc} \subseteq Q$.

Let Σ^ω be the set of infinite words over Σ. A run for $\sigma = \mathcal{A}_0\mathcal{A}_1\mathcal{A}_2 \ldots \in \Sigma^\omega$ denotes an infinite sequence $q_0 q_1 q_2 \ldots$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $(q_i, \mathcal{A}_i, q_{i+1}) \in \delta$ for $i \geq 0$. Run $q_0 q_1 q_2 \ldots$ is *accepting (accepted)* if $q_i \in \text{Acc}$ for infinitely many indices $i \in \mathbb{N}$ appearing in the run.

Intuitively, a run is accepted by a Buchi automaton if a state in Acc is visited infinitely often.

We use the definition of an accepting run in a Buchi automaton and the fact that every LTL formula φ can be represented by an equivalent Buchi automaton $\mathcal{A}_\varphi$ to define satisfaction of an LTL formula φ.

Definition 3. Let $\mathcal{A}_\varphi$ be a Buchi automaton corresponding to the LTL formula φ. A run $\sigma = s_0 s_1 s_2 \ldots$ in $\mathcal{T}$ *satisfies* φ, denoted by $\sigma \vDash \varphi$, if the word $L(\sigma)$ is accepted by $\mathcal{A}_\varphi$.

C. Graph Theory

This section lists basic definitions for graphs that will be necessary later. Let $G = (V, E)$ be a directed graph (digraph) with $|V|$ vertices and $|E|$ edges. Let $e = (u, v) \in E$ denote an edge from vertex u to vertex v. A *walk* is a finite edge

sequence $e_0, e_1, \ldots, e_p$, and a *cycle* is a walk in which the initial vertex is equal to the final vertex. A *path* is a walk with no repeated vertices, and a *simple cycle* is a path in which the initial vertex is equal to the final vertex.

A digraph $G = (V, E)$ is *strongly connected* if there exists a path between each pair of vertices $s, t \in V$. A digraph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A digraph $G' \subseteq G$ is a *strongly connected component* if it is a maximal strongly connected subgraph of G.

III. Problem Statement

In this section, we formally state the main problem of the paper and give an overview of our solution approach. Let $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ be a weighted transition system and φ be an LTL specification defined over AP.

Definition 4. Let σ be a run of $\mathcal{T}$ where σ_i is the state at the i-th index of σ. The *weighted average cost* of run σ is

$$J(\sigma) := \limsup_{n \to \infty} \frac{\sum_{i=0}^{n} c(\sigma_i, \sigma_{i+1})}{\sum_{i=0}^{n} w(\sigma_i, \sigma_{i+1})}, \tag{1}$$

where J maps runs of $\mathcal{T}$ to $\mathbb{R} \cup \infty$.

Since LTL specifications are typically defined over infinite sequences of states, we consider the (weighted) average cost function in (1) to ensure that the cost function is bounded. This cost function is well-defined when (i) $c(\sigma_i, \sigma_{i+1}) < \infty$ for all $i \geq 0$, and (ii) there exists a $j \in \mathbb{N}$ such that $w(\sigma_i, \sigma_{i+1}) > 0$ for infinitely many $i \geq j$, which we assume is true for the sequel. Assumption (ii) enforces that a run does not eventually visit only states with zero weights.

To better understand the weighted average cost function J, consider the case where $w(s, t) = 1$ for all transitions $(s, t) \in R$. Let a cost $c(s, t)$ be arbitrarily fixed for each transition $(s, t) \in R$. Then, $J(\sigma)$ is the average cost per transition between states (or average cost per stage). If $w(s, t) = 1$ for states in $s, t \in S' \subset S$ and $w(s, t) = 0$ for states in $S - S'$, then $J(\sigma)$ is the mean time per transition between states in S'.

As an example, consider $\sigma = (s_0 s_1)^\omega$ where $s_0 s_1$ repeats indefinitely. Let $c(s_0, s_1) = 1$, $c(s_1, s_0) = 2$, $w(s_0, s_1) = 1$, and $w(s_1, s_0) = 1$. Then, $J(\sigma) = 1.5$ is the average cost per transition. Now, let $w(s_1, s_0) = 0$. Then, $J(\sigma) = 3$ is the average cost per transition from s_0 to s_1.

The weighted average cost function is more natural than the minimax cost function of Smith et al. [20] in some application contexts. For example, consider an autonomous vehicle repeatedly picking up people and delivering them to a destination. It takes a certain amount of fuel to travel between discrete states, and each discrete state has a fixed number of people that need to be picked up. A natural problem formulation is to minimize the fuel consumption per person picked up, which is a weighted average cost where fuel is the cost and the number of people is the weight. The cost function in [20] cannot adequately capture this task.

Definition 5. An *optimal satisfying finite-memory run* of $\mathcal{T}$ is a run σ^* such that

$$J(\sigma^*) = \inf \{ J(\sigma) \mid \sigma \text{ is finite-memory run of } \mathcal{T},\ \sigma \vDash \varphi \}, \tag{2}$$

i.e., run σ^* achieves the infimum in (2).

An *optimal satisfying infinite-memory run* is defined similarly for infinite-memory runs of $\mathcal{T}$.

Although we show in Section V-B that infinite-memory runs are generally necessary to achieve the infimum in (2), we focus on finite-memory runs, as these are more practical than their infinite-memory counterparts. However, finding an optimal satisfying finite-memory run is potentially ill-posed, as the infimum might not be achieved due to the constraint that the run must also satisfy φ. This happens when it is possible to reduce the cost of a satisfying run by including an arbitrarily long, low weighted average cost subsequence. For instance, consider the run $\sigma = (s_0 s_0 s_1)^\omega$. Let $c(s_0, s_0) = 1$, $c(s_1, s_0) = c(s_0, s_1) = 2$ and the weights equal 1 for each transition. Assume that a specification is satisfied if s_1 is visited infinitely often. Then, $J(\sigma)$ can be reduced by including an arbitrarily large number of self transitions from s_0 to s_0 in σ, even though these do not affect satisfaction of the specification. Intuitively, one should restrict these repetitions to make finding an optimal satisfying finite-memory run well-posed. We will show that one can always compute an ϵ-suboptimal finite-memory run by restricting the length of these repetitions. We defer the details to Section IV, when we will have developed the necessary technical machinery.

Problem 1. Given a weighted transition system $\mathcal{T}$ and an LTL specification φ, compute an optimal satisfying finite-memory run σ^* of $\mathcal{T}$ if one exists.

Remark 1. For completeness, we show how to compute optimal satisfying infinite-memory runs in Section V-B. These runs achieve the minimal weighted average cost, but do so by adding arbitrarily long progressions of states that do not change whether or not the specification is satisfied.

IV. Reformulation of the problem

We solve Problem 1 by first creating a product automaton that represents runs that are allowed by the transition system $\mathcal{T}$ and also satisfy the LTL specification φ. We can limit our search for finite-memory runs, without loss of generality, to runs in the product automaton that are of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$. Here σ_{pre} is a finite walk and σ_{suf} is a finite cycle that is repeated infinitely often. Runs with this structure are said to be in *prefix-suffix* form. We observe that the weighted average cost only depends on σ_{suf}, which reduces the problem to searching for a cycle σ_{suf} in the product automaton. This search can be done using dynamic programming techniques for finite-memory runs. The optimal accepting run $\sigma^*_{\mathcal{P}}$ is then projected back on $\mathcal{T}$ as σ^*, which solves Problem 1.

A. Product Automaton

We use the standard product automaton construction, due to Vardi and Wolper [22], to represent runs that are allowed by the transition system and satisfy the LTL specification.

Definition 6. Let $\mathcal{T} = (S, R, s_0, AP, L, c, w)$ be a weighted transition system and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, \text{Acc})$ be a Buchi automaton. The *product automaton* $\mathcal{P} = \mathcal{T} \times \mathcal{A}$ is the tuple $\mathcal{P} := (S_\mathcal{P}, \delta_\mathcal{P}, \text{Acc}_\mathcal{P}, s_{\mathcal{P},0}, AP_\mathcal{P}, L_\mathcal{P}, c_\mathcal{P}, w_\mathcal{P})$, consisting of

(i) a finite set of states $S_\mathcal{P} = S \times Q$,
(ii) a transition relation $\delta_\mathcal{P} \subseteq S_\mathcal{P} \times S_\mathcal{P}$, where $((s,q),(s',q')) \in \delta_\mathcal{P}$ if and only if $(s,s') \in R$ and $(q, L(s), q') \in \delta$,
(iii) a set of accepting states $\text{Acc}_\mathcal{P} = S \times \text{Acc}$,
(iv) a set of initial states $S_{\mathcal{P},0}$, with $(s_0, q_0) \in S_{\mathcal{P},0}$ if $q_0 \in Q_0$,
(v) a set of atomic propositions $AP_\mathcal{P} = Q$,
(vi) a labeling function $L_\mathcal{P} : S \times Q \to 2^Q$,
(vii) a cost function $c_\mathcal{P} : \delta_\mathcal{P} \to \mathbb{R}$, where $c_\mathcal{P}((s,q),(s',q')) = c(s,s')$ for all $((s,q),(s',q')) \in \delta_\mathcal{P}$, and
(viii) a weight function $w_\mathcal{P} : \delta_\mathcal{P} \to \mathbb{R}_{\geq 0}$, where $w_\mathcal{P}((s,q),(s',q')) = w(s,s')$ for all $((s,q),(s',q')) \in \delta_\mathcal{P}$.

A run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ is *accepting* if $(s_i,q_i) \in \text{Acc}_\mathcal{P}$ for infinitely many indices $i \in \mathbb{N}$.

The *projection* of a run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ in the product automaton $\mathcal{P}$ is the run $\sigma = s_0 s_1 \ldots$ in the transition system. The projection of a finite-memory run in $\mathcal{P}$ is a finite-memory run in $\mathcal{T}$ [1].

The following proposition relates accepting runs in $\mathcal{T}$ and $\mathcal{P}$ and is due to Vardi and Wolper [22].

Proposition 1. *([22]) Let $\mathcal{A}_\varphi$ be a Buchi automaton corresponding to the LTL formula φ. For any accepting run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ in the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\varphi$, its projection $\sigma = s_0 s_1 \ldots$ in the transition system $\mathcal{T}$ satisfies φ. Conversely, for any run $\sigma = s_0 s_1 \ldots$ in $\mathcal{T}$ that satisfies φ, there exists an accepting run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ in the product automaton.*

Lemma 1. *For any accepting run $\sigma_\mathcal{P}$ in $\mathcal{P}$ and its projection σ in $\mathcal{T}$, $J(\sigma_\mathcal{P}) = J(\sigma)$. Conversely, for any σ in $\mathcal{T}$ that satisfies φ, there exists an accepting run $\sigma_\mathcal{P}$ in $\mathcal{P}$ with $J(\sigma_\mathcal{P}) = J(\sigma)$.*

Proof: Consider a run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ in $\mathcal{P}$. By definition, for states $(s_i,q_i),(s_{i+1},q_{i+1}) \in S_\mathcal{P}$ and $s_i, s_{i+1} \in S_\mathcal{T}$, the cost $c_\mathcal{P}((s_i,q_i),(s_{i+1},q_{i+1})) = c(s_i,s_{i+1})$ and the weight $w_\mathcal{P}((s_i,q_i),(s_{i+1},q_{i+1})) = w(s_i,s_{i+1})$ for all $i \geq 0$, so $J(\sigma_\mathcal{P}) = J(\sigma)$. Now consider a run $\sigma = s_0 s_1 \ldots$ in $\mathcal{T}$ that satisfies φ. Proposition 1 gives the existence of an accepting run $\sigma_\mathcal{P} = (s_0,q_0)(s_1,q_1)\ldots$ in $\mathcal{P}$, and so $J(\sigma_\mathcal{P}) = J(\sigma)$. ■

By Lemma 1, an accepting run $\sigma^*_\mathcal{P}$ with minimal weighted average cost in the product automaton has a projection in the transition system σ^* that is a satisfying run with minimal weighted average cost.

B. Prefix-Suffix Form

We show that Problem 1 is equivalent to finding a run of the form $\sigma_\mathcal{P} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$, in the product automaton $\mathcal{P}$ that minimizes the weighted average cost function (1). We equivalently treat the product automaton as a graph when convenient. Our analysis and notation in this section is similar to that of [20]; we optimize a different cost function on the Vardi and Wolper [22] product automaton construction.

Definition 7. Let σ_{pre} be a finite walk in $\mathcal{P}$ and σ_{suf} be a finite cycle in $\mathcal{P}$. A run $\sigma_\mathcal{P}$ is in *prefix-suffix* form if it is of the form $\sigma_\mathcal{P} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$.

It is well-known that if there exists an accepting run in $\mathcal{P}$ for an LTL formula φ, then there exists an accepting run in prefix-suffix form for φ [1]. This can be seen since the product automaton $\mathcal{P}$ is finite, but an accepting run is infinite and visits an accepting state infinitely often. Thus, at least one accepting state must be visited infinitely often, and this can correspond to a repeated cycle including the accepting state. For an accepting run $\sigma_\mathcal{P}$, the suffix σ_{suf} is a cycle in the product automaton $\mathcal{P}$ that satisfies the acceptance condition, i.e., it includes an accepting state. The prefix σ_{pre} is a finite run from an initial state $s_{\mathcal{P},0}$ to a state on an accepting cycle.

The following lemma shows that a minimum weighted average cost run can be found searching over finite-memory runs of the form $\sigma_\mathcal{P} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$.

Lemma 2. *There exists at least one accepting finite-memory run $\sigma_\mathcal{P}$ of $\mathcal{P}$ that minimizes J and is in prefix-suffix form, i.e., $\sigma_\mathcal{P} = \sigma_{pre}(\sigma_{suf})^\omega$.*

Proof: Let σ_{gen} be an accepting finite-memory run in $\mathcal{P}$ that is not in prefix-suffix form and has weighted average cost $J(\sigma_{\text{gen}})$. Since σ_{gen} is accepting, it must visit an accepting state $s_{\text{acc}} \in S_\mathcal{P}$ infinitely often. Let the finite walk σ_{pre} be from an initial state $s_{\mathcal{P},0}$ to the first visit of s_{acc}. Now consider the set of walks between successive visits to s_{acc}. Each walk starts and ends at s_{acc} (so it is a cycle), is finite with bounded length, and has a weighted average cost associated with it. For each cycle τ, compute the weighted average cost $J(\tau^\omega)$. Let σ_{suf} be the finite cycle with the minimum weighted average cost over all τ. Then, $J(\sigma_\mathcal{P}) = J(\sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega) \leq J(\sigma_{\text{gen}})$. Since σ_{gen} was arbitrary, the claim follows. ■

The next proposition shows that the weighted average cost of a run does not depend on any finite prefix of the run.

Proposition 2. *Let $\sigma = s_0 s_1 \ldots$ be a run (in $\mathcal{T}$ or $\mathcal{P}$) and $\sigma_{k:\infty} = s_k s_{k+1} \ldots$ be the run σ starting at index $k \in \mathbb{N}$. Then, their weighted average costs are equal, i.e., $J(\sigma) = J(\sigma_{k:\infty})$.*

Proof: From Definition 1, costs and weights depend only on the transition—not the index. Also, from the assumptions that directly follow equation (1), transitions with positive weight occur infinitely often. Thus,

$$\begin{aligned} J(\sigma) &:= \limsup_{n\to\infty} \frac{\sum_{i=0}^{n} c(\sigma_i,\sigma_{i+1})}{\sum_{i=0}^{n} w(\sigma_i,\sigma_{i+1})} \\ &= \limsup_{n\to\infty} \frac{\sum_{i=0}^{k-1} c(\sigma_i,\sigma_{i+1}) + \sum_{i=k}^{n} c(\sigma_i,\sigma_{i+1})}{\sum_{i=0}^{k-1} w(\sigma_i,\sigma_{i+1}) + \sum_{i=k}^{n} w(\sigma_i,\sigma_{i+1})} \\ &= \limsup_{n\to\infty} \frac{\sum_{i=k}^{n} c(\sigma_i,\sigma_{i+1})}{\sum_{i=k}^{n} w(\sigma_i,\sigma_{i+1})} = J(\sigma_{k:\infty}). \end{aligned}$$

■

From Proposition 2, finite prefixes do not contribute to the weighted average cost function, so

$J(\sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega) = J((\sigma_{\text{suf}})^\omega)$. Thus, one can optimize over the suffix σ_{suf}, which corresponds to an accepting cycle in the product automaton. Given an optimal accepting cycle σ^*_{suf}, one then computes a walk from an initial state to σ^*_{suf}.

We now define a weighted average cost function for finite walks in the product automaton that is analogous to (1).

Definition 8. The *weighted average cost* of a finite walk $\sigma_{\mathcal{P}} = (s_0, q_0)(s_1, q_1)\dots(s_m, q_m)$ in the product automaton is

$$\tilde{J}(\sigma_{\mathcal{P}}) := \frac{\sum_{i=0}^{m} c_{\mathcal{P}}(\sigma_i, \sigma_{i+1})}{\sum_{i=0}^{m} w_{\mathcal{P}}(\sigma_i, \sigma_{i+1})}, \tag{3}$$

with similar assumptions on c and w as for equation (1).

Problem 2. Let $acc(\mathcal{P})$ be the set of all accepting cycles in the product automaton $\mathcal{P}$ reachable from an initial state. Find a suffix σ^*_{suf} where $\tilde{J}(\sigma^*_{\text{suf}}) = \inf_{\sigma_{\mathcal{P}} \in acc(\mathcal{P})} \tilde{J}(\sigma_{\mathcal{P}})$ if it exists.

Proposition 3. *Let $\sigma^*_{\mathcal{P}} = \sigma_{pre}(\sigma^*_{suf})^\omega$ be a solution to Problem 2. The projection to the transition system of any optimal accepting run $\sigma^*_{\mathcal{P}}$ is a solution to Problem 1.*

Proof: From Lemma 2, there exists an accepting run $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$ that minimizes J. From Proposition 2 and equation (5), $J(\sigma_{\mathcal{P}}) = J((\sigma_{\text{suf}})^\omega) = \tilde{J}(\sigma_{\text{suf}})$. ■

We now pause to give a high-level overview of our approach to solving Problem 1, using its reformulation as Problem 2. The major steps are outlined in Algorithm 1. First, a Buchi automaton $\mathcal{A}_\varphi$ corresponding to the LTL formula φ is created. Then, we create the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\varphi$. Reachability analysis on $\mathcal{P}$ determines, in linear time in the size of $\mathcal{P}$, all states that can be reached from an initial state, and thus guarantees existence of a finite prefix σ_{pre} to all remaining states. Next, we compute the strongly connected components (scc) of $\mathcal{P}$, since two states can be on the same cycle only if they are in the same strongly connected component. This partitions the original product automaton into sub-graphs, each of which can be searched independently for optimal cycles.

For each strongly connected component of $\mathcal{P}$, we compute the cycle σ_{suf} with the minimum weighted average cost, regardless of whether or not it is accepting (see Section V-B). This is the infimum of the minimum weighted average cost over all accepting cycles. If this cycle is accepting, then the infimum is achieved by a finite-memory run. If not, then the infimum is not achieved by a finite-memory run and thus we must further constrain the form of the suffix σ_{suf} to make the optimization well-posed.

A natural choice is finite-memory policies, which correspond to bounding the length of σ_{suf}. We can solve for the optimal accepting σ_{suf} subject to this additional constraint using dynamic programming techniques. The optimal accepting σ_{suf} over all strongly connected components is σ^*_{suf}. Given σ^*_{suf}, we compute a finite walk σ_{pre} from an initial state to any state on σ^*_{suf}. The finite walk σ_{pre} is guaranteed to exist due to the initial reachability computation. The optimal run in the product automaton is then $\sigma^*_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma^*_{\text{suf}})^\omega$. The projection of $\sigma^*_{\mathcal{P}}$ to the transition system as σ^* solves Problem 1, given the additional constraint that σ_{suf} has bounded length.

Algorithm 1 Overview of solution approach

Input: Weighted transition system $\mathcal{T}$ and LTL formula φ
Output: Run σ^*, a solution to Problem 1

```
Create Buchi automaton A_φ
Create product automaton P = T × A_φ
Compute states in P reachable from an initial state
Calculate strongly connected components (scc) of P
for scc ∈ P do
    Let σ*_suf = arg inf { J̃(σ) | σ is cycle in P }
    if σ*_suf is an accepting cycle then
        break {finite-memory run achieves infimum}
    end if
    Find best bounded-length accepting σ*_suf over all s_acc ∈ scc
    (Section V)
end for
Take optimal σ*_suf over all sccs
Compute finite prefix σ_pre from initial state to σ*_suf
Project run σ*_P = σ_pre(σ*_suf)^ω to T as σ*
```

Remark 2. In Section V, we treat the product automaton as a graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$, with the natural bijections between states $S_{\mathcal{P}}$ and vertices $V_{\mathcal{P}}$ and between edges $(u, v) \in E_{\mathcal{P}}$ and transitions in δ_P. We further assume that a reachability computation has been done, so that $G_{\mathcal{P}}$ only includes states reachable from an initial state $s_{\mathcal{P},0}$. We assume that $\mathcal{G}_{\mathcal{P}}$ is strongly connected. If not, the strongly connected components of the $\mathcal{P}$ can be found in $O(|V_{\mathcal{P}}| + |E_{\mathcal{P}}|)$ time with Tarjan's algorithm [6]. To compute the optimal cycle for the entire graph, one finds the optimal cycle in each strongly connected component and then selects the optimal over all strongly connected components. We denote each strongly connected component of $G_{\mathcal{P}}$ by $G = (V, E)$, where $n = |V|$ and $m = |E|$.

V. Solution Approach

In this section, we give algorithms for computing optimal finite-memory and infinite-memory runs. We assume that $G = (V, E)$ is a strongly connected component of the product automaton $\mathcal{P}$ and has at least one accepting state. The techniques we adapt were originally developed for the minimum cost-to-time ratio problem [7, 8, 10, 13, 14].

A. Computing finite-memory runs

We present two related algorithms that find an optimal accepting cycle σ^*_{suf} in increasing levels of generality. While the algorithm in Section V-A2 subsumes the first algorithm, the first one is more intuitive and computationally efficient when the weight function is a constant function.

1) Minimum mean cycle: We first investigate the case where $w(e) = 1$ for all $e \in E$, so the total weight of a walk is equivalent to the number of transitions. This is similar to the problem investigated by Karp [13], with the additional constraint that the cycle must be accepting. This additional constraint prevents a direct application of Karp's theorem [13], but our approach is similar. The intuition is that, conditional

on the weight of a walk, the minimum cost walk gives the minimum average cost walk.

Let $s \in V$ be an accepting vertex (i.e., accepting state). For every $v \in V$, let $F_k(v)$ be the minimum cost of a walk of length $k \in \mathbb{N}$ from s to v. Thus, $F_k(s)$ is the minimum cost cycle of length k, which we note is accepting by construction. We compute $F_k(v)$ for all $v \in V$ and $k = 1, \ldots, n$ by the recurrence

$$F_k(v) = \min_{(u,v)\in E} \left[F_{k-1}(u) + c(u,v)\right], \tag{4}$$

where $F_0(s) = 0$ and $F_0(v) = \infty$ for $v \neq s$.

It follows from equation (4) that $F_k(v)$ can be computed for all $v \in V$ in $O(|V||E|)$ operations. To find the minimum mean cycle cost with fewer than M transitions (i.e., bounded-length suffix), we simply compute $\min F_k(s)/k$ for all $k = 1, \ldots, M$. If there are multiple cycles with the optimal cost, pick the cycle corresponding to the minimum k.

We repeat the above procedure for each accepting vertex $s \in V$. The minimum mean cycle value is the minimum of these values. We record the optimal vertex s^* and the corresponding integer k^*. To determine the optimal cycle corresponding to s^* and k^*, we simply determine the corresponding transitions from (4) for $F_{k^*}(s^*)$ from vertex s^*. The repeated application of recurrence (4) takes $O(n_a|V||E|)$ operations, where n_a is the number of accepting vertices, which is typically significantly smaller than $|V|$.

2) Minimum cycle ratio: We now discuss a more general case, which subsumes the discussion in Section V-A1. Our approach is based on that of Hartmann and Orlin [10], who consider the unconstrained case.

Let the possible weights be in the integer set Val $= \{1, \ldots, w_{max}\}$, where w_{max} is a positive integer. Let $E' \subseteq E$, and define weights as

$$w(e) = \begin{cases} x \in \text{Val} & \text{if } e \in E' \\ 0 & \text{if } e \in E - E'. \end{cases}$$

The setup in Section V-A1 is when $E' = E$ and Val $= \{1\}$.

Let $T_u := \max_{(u,v)\in E} w(u,v)$ for each vertex $u \in V$. Then, $T := \sum_{u\in V} T_u$ is the maximum weight of a path.

Let $s \in V$ be an accepting state. For each $v \in V$, let $G_k(v)$ be the minimum cost walk from s to v that has total weight equal to k. This definition is similar to $F_k(v)$ in Section V-A1 except now k is the total weight w of the edges, which is no longer simply the number of edges. Let $G'_k(v)$ be the minimum cost walk from s to v that has total weight equal to k and with the last edge of the walk in E'. Finally, let $d(u,v)$ be the minimum cost of a path from u to v in G consisting solely of edges of $E - E'$. The costs $d(u,v)$ are pre-computed using an all-pairs shortest paths algorithm, which assumes there are no negative-cost cycles in $E - E'$ [6].

The values $G_k(v)$ can be computed for all $v \in V$ and $k = 1, \ldots, T$ by the recurrence

$$\begin{aligned} G'_k(v) &= \min_{(u,v)\in E'} \left[G_{k-w(u,v)}(u) + c(u,v)\right] \qquad (5) \\ G_k(v) &= \min_{u\in V} \left[G'_k(u) + d(u,v)\right] \end{aligned}$$

where $G_0(v) = d(s,v)$.

The optimal cycle cost and the corresponding cycle are recovered in a similar manner as described in Section V-A1, and are accepting by construction. The recurrence in (5) requires $O(n_a T|V|^2)$ operations, where n_a is the number of accepting vertices. This algorithm runs in pseudo-polynomial time, as T is an integer, and so its binary description length is $O(\log(T))$. The recurrence for G_k can be computed more efficiently if the edge costs are assumed to be non-negative, as explained in [10]. This improves the overall complexity of the recurrence to $O(n_a T(|E| + |V|\log|V|))$ time [10].

Remark 3. Consider the special case where weights are restricted to be 0 or 1. Then, the total weight T is $O(|V|)$ and the above algorithm has polynomial time complexity $O(n_a|V|^3)$ or $O(n_a|V|(|E| + |V|\log|V|))$ if edge costs are assumed to be non-negative.

Remark 4. Although finite prefixes do not affect the cost (cf. Proposition 2), it may be desired to create a "good" finite prefix. The techniques described in Section V-A2 (and similarly Section V-A1) can be adapted to create these finite prefixes. After computing the optimal accepting cycle C^*, one can compute the values $G_k(v)$ and corresponding walk defined with respect to the initial state s_0 for all states $v \in C^*$.

B. Computing infinite-memory runs

Infinite-memory runs achieve the minimum weighted average cost J^*. However, their practical use is limited, as they achieve J^* by increasingly visiting states that do not affect whether or not the specification is satisfied. This is unlikely the designer's intent, so we only briefly discuss these runs for completeness. A related discussion on infinite-memory runs, but in the adversarial environment context, appears in [5].

Let σ_{opt} be the (possibly non-accepting) cycle with the minimum weighted average cost $J(\sigma^\omega_{\text{opt}})$ over all cycles in G. Clearly, the restriction that a cycle is accepting can only increase the weighted average cost. Let σ_{acc} be a cycle that contains both an accepting state and a state in σ_{opt}. Let $\sigma_{\text{acc},i}$ denote the ith state in σ_{acc}. For symbols α and β, let $(\alpha\beta^k)^\omega$ for $k = 1, 2, \ldots$ denote the sequence $\alpha\beta\alpha\beta\beta\alpha\beta\beta\beta\ldots$.

Proposition 4. *Let $\sigma_{\mathcal{P}} = (\sigma_{acc}\sigma^k_{opt})^\omega$, where $k = 1, 2, \ldots$. Then, $\sigma_{\mathcal{P}}$ is accepting and achieves the minimum weighted average cost* (1).

Proof: Run $\sigma_{\mathcal{P}}$ is accepting because it repeats σ_{acc} infinitely often. Let $\alpha_c = \sum_{i=0}^{p} c(\sigma_{\text{acc},i}, \sigma_{\text{acc},i+1})$ and $\alpha_w = \sum_{i=0}^{p} w(\sigma_{\text{acc},i}, \sigma_{\text{acc},i+1})$, where integer p is the length of σ_{acc}. Define β_c and β_w similarly for σ_{opt}. Then,

$$\begin{aligned} J(\sigma) &:= \limsup_{n\to\infty} \frac{\sum_{k=1}^{n}(\alpha_c + k\beta_c)}{\sum_{k=1}^{n}(\alpha_w + k\beta_w)} \\ &= \limsup_{n\to\infty} \frac{(n+1)\alpha_c + \beta_c \sum_{k=1}^{n} k}{(n+1)\alpha_w + \beta_w \sum_{k=1}^{n} k} \\ &= \limsup_{n\to\infty} \frac{\beta_c}{\beta_w} = J((\sigma_{\text{opt}})^\omega). \end{aligned}$$

■

A direct application of a minimum cost-to-time ratio algorithm, e.g., [10], can be used to compute σ_{opt} since there is no constraint that it must include an accepting state. Also, given σ_{opt}, σ_{acc} always exists as there is an accepting state in the same strongly connected component as σ_{opt} by construction.

The next proposition shows that finite-memory runs can be arbitrarily close to the optimal weighted average cost J^*.

Proposition 5. *Given any $\epsilon > 0$, a finite-memory run $\sigma_{\mathcal{P}}$ exists with $J((\sigma_{\mathcal{P}})^\omega) < J((\sigma_{opt})^\omega) + \epsilon = J^* + \epsilon$.*

Proof: Construct a finite-memory run of the form $\sigma_{\mathcal{P}} = \sigma_{\text{pre}}(\sigma_{\text{suf}})^\omega$, where σ_{suf} has fixed length. In particular, let $\sigma_{\text{suf}} = \sigma_{\text{acc}}(\sigma_{\text{opt}})^M$ for a large (fixed) integer M. By picking M large enough, the error between $J((\sigma_{\text{suf}})^\omega)$ and $J((\sigma_{\text{opt}})^\omega)$ can be made arbitrarily small. ■

Thus, finite-memory runs can approximate the performance of infinite-memory runs arbitrarily closely. This allows a designer to tradeoff between runs with low weighted average cost and runs with short lengths.

C. Complexity

We now discuss the complexity of the entire procedure, i.e., Algorithm 1. The number of states and transitions in the transition system is $n_{\mathcal{T}}$ and $m_{\mathcal{T}}$, respectively. The ω-regular specification is given by a Buchi automaton $\mathcal{A}_\varphi$. The product automaton has $n_{\mathcal{P}} = n_{\mathcal{T}} \times |\mathcal{A}_\varphi|$ states and $m_{\mathcal{P}}$ edges. For finite-memory runs, the dynamic programming algorithms described in Section V-A take $O(n_a n_{\mathcal{P}} m_{\mathcal{P}})$ and $O(n_a T(m_{\mathcal{P}} + n_{\mathcal{P}} \log n_{\mathcal{P}}))$ operations, assuming non-negative edge weights for the latter bound. Here, n_a is the number of accepting states in the product automaton. Usually, n_a is significantly smaller than $n_{\mathcal{P}}$. For infinite-memory runs, there is no accepting state constraint for the cycles, so standard techniques [10, 13] can be used that take $O(n_{\mathcal{P}} m_{\mathcal{P}})$ and $O(T(m_{\mathcal{P}} + n_{\mathcal{P}} \log n_{\mathcal{P}}))$ operations, again assuming non-negative edge weights for the latter bound. The algorithms in Section V are easily parallelizable, both between strongly connected components of $\mathcal{P}$ and for each accepting state.

In practice, an LTL formula φ is used to automatically generate a Buchi automaton $\mathcal{A}_\varphi$. The length of an LTL formula φ is the number of symbols. A corresponding Buchi automaton $\mathcal{A}_\varphi$ has size $2^{O(|\varphi|)}$ in the worst-case, but this behavior is rarely encountered in practice.

VI. Examples

The following examples demonstrate the techniques developed in Section V in the context of autonomous driving and surveillance. Each cell in Figures (1) and (2) corresponds to a state, and each state has transitions to its four neighbors. We specify costs and weights over states, as discussed in Section II. Tasks are formally specified by LTL formulas and informally in English. We use the following LTL symbols without definition: negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), always ($\Box$), and eventually ($\Diamond$) [1].

The first example is motivated by autonomous driving. The weighted transition system represents an abstracted car that can transition between neighboring cells in the grid (Figure 1). The car's task is to repeatedly visit the states labeled a, b, and c while always avoiding states labeled x. Formally, $\varphi = \Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c \wedge \Box\neg x$. Costs are defined over states to reward driving in the proper lane (the outer boundary) and penalize leaving it. Weights are zero for all states except states labeled a, b, and c, which each have weight of one.

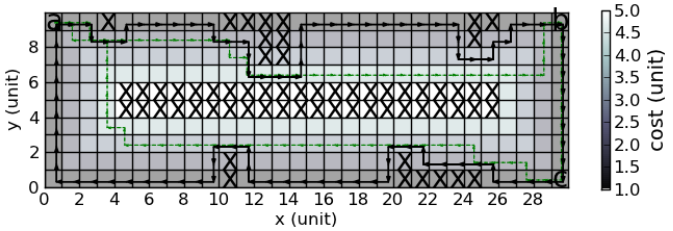

Fig. 1. Driving task, with optimal run (black) and feasible run (green).

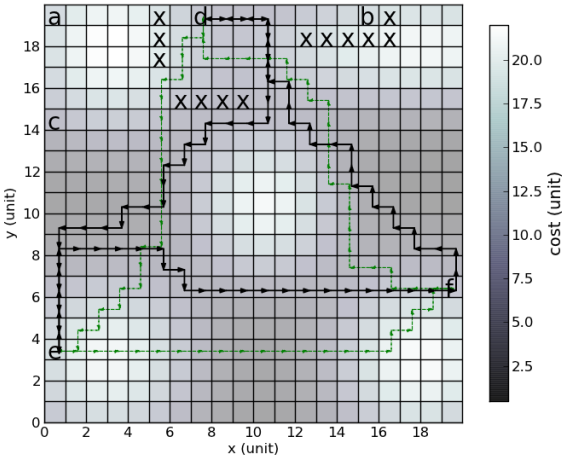

Fig. 2. Surveillance task, with optimal run (black) and feasible run (green).

The second example, Figure 2, is motivated by surveillance. The robot's task is to repeatedly visit states labeled either a, b, c or d, e, f. States labeled x should always be avoided. Formally, $\varphi = ((\Box\Diamond a \wedge \Box\Diamond b \wedge \Box\Diamond c) \vee (\Box\Diamond d \wedge \Box\Diamond e \wedge \Box\Diamond f)) \wedge \Box\neg x$. Costs vary with the state as described in Figure 2, and might describe the time to navigate different terrain. The weight is zero at each state, except states a and f, where the weight is one.

Numerical results are in Table I. Computation times for optimal and feasible runs are given by t_{opt} and t_{feas} respectively. All computations were done using Python on a Linux desktop with a dual-core processor and 2 GB of memory. The feasible satisfying runs were generated with depth-first search. The optimal satisfying runs were generated with the algorithm from Section V-A2. Since it was possible to decrease the weighted average cost by increasing the length of the cycle (i.e., the infimum was not achieved by a finite-memory satisfying run), we used the shortest cycle such that $J(\sigma_{\text{opt}}) < \infty$. Thus, the optimal values $J(\sigma_{\text{opt}})$ are conservative. The improvement of the optimal runs over a feasible run is evident from Figures 1 and 2. In Figure 1, the optimal run immediately heads back to its lane to reduce costs, while the feasible run does not. In Figure 2, the optimal run avoids visiting high-cost regions.

TABLE I
NUMERICAL RESULTS

Example	$\mathcal{T}$ (nodes/edges)	$\mathcal{A}_\varphi$	$\mathcal{P}$	$\mathcal{P}$ (reachable)	# SCC	# acc. states	J_{opt} (units)	J_{feas} (units)	t_{opt} (sec)	t_{feas} (sec)
Driving	300 / 1120	4 / 13	1200 / 3516	709 / 2396	1	1	49.3	71.3	2.49	0.68
Surveillance	400 / 1520	9 / 34	3600 / 14917	2355 / 8835	2	2	340.9	566.3	21.9	1.94

VII. CONCLUSIONS

We created optimal runs of a weighted transition system that minimized a weighted average cost function subject to ω-regular language constraints. These constraints include the well-studied linear temporal logic as a subset. We showed that optimal system runs correspond to cycles in a lifted product space, which includes behaviors that both are valid for the system and satisfy the temporal logic specification. Dynamic programming techniques were used to solve for an optimal cycle in this product space.

Future directions include investigating notions of optimality with both non-deterministic transition systems and adversarial environments. Additionally, better computational complexity may be achievable for fragments of ω-regular languages.

ACKNOWLEDGEMENTS

The authors thank Pavithra Prabhakar for helpful discussions. This work was supported in part by a NDSEG Fellowship, the Boeing Corporation, and NSF Grant CNS-0911041.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] C. Belta and L.C.G.J.M. Habets. Control of a class of non-linear systems on rectangles. *IEEE Transaction on Automatic Control*, 51:1749–1759, 2006.

[3] C. Belta, V. Isler, and G. Pappas. Discrete abstractions for robot motion planning and control in polygonal environments. *IEEE Transactions on Robotics*, 21:864–874, 2004.

[4] D. P. Bertsekas. *Dynamic Programming and Optimal Control (Vol. I and II)*. Athena Scientific, 2001.

[5] K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *Annual Symposium on Logic in Computer Science (LICS)*, 2005.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms: 2nd ed*. MIT Press, 2001.

[7] G. B. Dantzig, W. O. Blattner, and M. R. Rao. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 77–84. Dunod, Paris and Gordon and Breach, New York, 1967.

[8] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1998.

[9] L. Habets, P.J. Collins, and J.H. van Schuppen. Reachability and control synthesis for piecewise-affine hybrid systems on simplices. *IEEE Transaction on Automatic Control*, 51:938–948, 2006.

[10] M. Hartmann and J. B. Orlin. Finding mimimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567–574, 1993.

[11] Sertac Karaman and Emilio Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *Proc. of IEEE Conference on Decision and Control*, 2009.

[12] Sertac Karaman, Ricardo G. Sanfelice, and Emilio Frazzoli. Optimal control of mixed logical dynamical systems with linear temporal logic specifications. In *Proc. of IEEE Conference on Decision and Control*, pages 2117–2122, 2008. doi: 10.1109/CDC.2008.4739370.

[13] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[14] R. M. Karp and J. B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete and Applied Mathematics*, 3:37–45, 1981.

[15] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[16] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transaction on Automatic Control*, 53(1):287–297, 2008.

[17] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25:1370–1381, 2009.

[18] S. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20:378–400, 2001.

[19] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. *IEEE Transactions on Robotics*, 26: 469–482, 2010.

[20] S. L. Smith, J. Tumova, C. Belta, and D. Rus. Optimal path planning for surveillance with temporal-logic constraints. *The International Journal of Robotics Research*, 30:1695–1708, 2011.

[21] P. Toth and D. Vigo, editors. *The Vehicle Routing Problem*. Philadelphia, PA: SIAM, 2001.

[22] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science*, pages 322–331, 1986.

[23] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Proc. of the 13th International Conference on Hybrid Systems: Computation and Control*, 2010.