ijrr

The International Journal of Robotics Research
30(14) 1695–1708

DOI: 10.1177/0278364911417911
ijr.sagepub.com
SAGE

Optimal path planning for surveillance with temporal-logic constraints*

Stephen L Smith[1], Jana Tůmová[2,3], Calin Belta[2] and Daniela Rus[4]

Abstract

In this paper we present a method for automatically generating optimal robot paths satisfying high-level mission specifications. The motion of the robot in the environment is modeled as a weighted transition system. The mission is specified by an arbitrary linear temporal-logic (LTL) formula over propositions satisfied at the regions of a partitioned environment. The mission specification contains an optimizing *proposition, which must be repeatedly satisfied. The cost function that we seek to minimize is the maximum time between satisfying instances of the optimizing proposition. For every environment model, and for every formula, our method computes a robot path that minimizes the cost function. The problem is motivated by applications in robotic monitoring and data-gathering. In this setting, the optimizing proposition is satisfied at all locations where data can be uploaded, and the LTL formula specifies a complex data-collection mission. Our method utilizes Büchi automata to produce an automaton (which can be thought of as a graph) whose runs satisfy the temporal-logic specification. We then present a graph algorithm that computes a run corresponding to the optimal robot path. We present an implementation for a robot performing data collection in a road-network platform.*

Keywords

Motion planning, optimal path planning, temporal logic

1. Introduction

The goal of this paper is to plan the optimal motion of a robot subject to temporal-logic constraints. This problem arises in many applications where a mobile robot has to perform a sequence of operations subject to external constraints. For example, in a persistent data-gathering task, the robot is required to gather data at several locations and then visit a different set of upload sites to transmit the data. Referring to Figure 1, we would like to enable tasks such as 'Repeatedly gather data at locations g_1, g_2, and g_3. Upload data at either u_1 or u_2 after each data-gather. Follow the road rules, and avoid the road connecting i_4 to i_2'. We wish to determine a robot motion that completes the task and minimizes a cost function, such as the maximum time between data uploads.

Motion and path planning have been studied extensively in the robotics literature (LaValle, 2006). Much of the work has focused on point-to-point navigation, where a mobile robot must travel from a source to a destination, while avoiding obstacles. Many effective solutions have been proposed for this problem, including discretized approaches that utilize graph-search algorithms such as A^* (see, for example, Russell and Norvig, 2003; LaValle, 2006); continuous approaches involving navigation functions and potential fields (Rimon and Koditschek, 1992); and sampling-based methods such as Rapidly-Exploring Random Trees (RRTs) (LaValle and Kuffner, 2001; Tedrake et al., 2010). However, the above approaches do not address more complex planning objectives, where robots must visit multiple locations in an environment, subject to logical or temporal constraints.

Recently there has been an increased interest in using temporal logic to specify mission plans for robots (Antoniotti and Mishra, 1995; Loizou and Kyriakopoulos, 2004; Quottrup et al., 2004; Belta et al., 2005; Fainekos et al., 2009; Kress-Gazit et al., 2009; Wongpiromsarn et al.,

[1]Department of Electrical and Computer Engineering, University of Waterloo, Canada
[2]Department of Mechanical Engineering, Boston University, USA
[3]Faculty of Informatics, Masaryk University, Czech Republic
[4]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA

*Based on Optimal Path Planning Under Temporal Logic Constraints by S. L. Smith, J. Tumova, C. Belta, and D. Rus which appeared in 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). © 2010 IEEE.

Corresponding author:
Stephen L. Smith, Department of Electrical and Computer Engineering, University of Waterloo, 200 University Avenue West, Waterloo ON, N2L 3G1 Canada
Email: stephen.smith@uwaterloo.ca

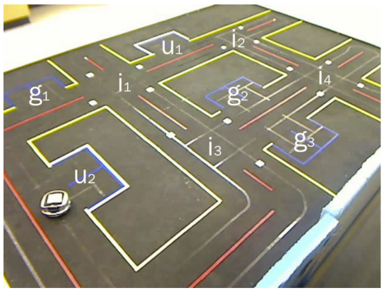

Fig. 1. An environment consisting of roads, intersections, and parking lots. An example mission in the environment is 'Repeatedly gather data at locations g_1, g_2, and g_3. Upload data at either u_1 or u_2 after each data-gather. Follow the road rules, and avoid the road connecting i_4 to i_2.'

2010). Temporal logic is appealing because it provides formal high-level languages in which to describe a complex mission. In addition, tools from model checking (Vardi and Wolper, 1986; Holzmann, 1997; Clarke et al., 1999; Barnat et al., 2009) can be used to verify the existence of a robot path satisfying the specification, and they can produce a satisfying path. However, frequently there are multiple robot paths that satisfy a given specification. In this case, one would like to choose the *optimal* path according to a cost function. The current tools from model checking do not provide a method for doing this. In this paper we consider linear temporal-logic specifications, and a particular form of cost function, and provide a method for computing optimal paths.

In terms of optimizing paths, the most closely related work has been on the vehicle-routing problem (VRP) (Toth and Vigo, 2001). In vehicle routing, the problem is to plan routes for vehicles to optimally service customers. The VRP generalizes the well-known traveling salesman problem (TSP) by considering aspects such as multiple vehicles, vehicles with capacity constraints, and vehicles that must depart and return to specified depot locations. Such aspects can be thought of as specific examples of logical or temporal constraints. While the vehicle-routing problem is NP-hard, many effective heuristics have been developed that provide good solutions to moderately sized problems (Laporte, 2009).

Recent results (Karaman and Frazzoli, 2008*a*,*b*) present extensions of vehicle-routing problems to more general classes of temporal constraints (see also Karaman et al., 2009). In Karaman and Frazzoli (2008*b*), the authors consider vehicle routing with metric temporal-logic specifications. The goal is to minimize a cost function of the vehicle paths (such as total distance traveled). The authors present a method for computing an optimal set of paths by converting the problem to a mixed integer linear program (MILP). While the approach is computationally intensive, it has been used to solve problems of real-world significance. However, their method cannot be applied to the persistent-monitoring and data-gathering applications that are of interest in this paper. This is due to the fact that their method applies only to specifications where the temporal operators are applied directly to atomic propositions. Thus, it does not allow for specifications of the form 'always eventually,' which appear when specifying that a robot should repeatedly perform a task. Because of this, in this paper we take an entirely different approach to optimizing robot motion. The approach that we present leads to an optimization problem on a graph, rather than a MILP.

The contribution of this paper is to present an algorithm that generates optimal robot paths satisfying general linear temporal-logic (LTL) formulas. The cost function that we minimize is motivated by problems in monitoring and data-gathering, and it quantifies the time between satisfying instances of a single *optimizing proposition*. Our solution, summarized in the OPTIMAL-RUN algorithm of Section 4, operates as follows. We represent the motion of the robot in the environment as a weighted transition system. Then, we convert the LTL specification to a Büchi automaton. We synchronize the transition system with the Büchi automaton to create a product automaton. In this automaton, a satisfying run is any run that visits a set of accepting states infinitely often. We show that there exists an optimal run that is in 'prefix–suffix' structure, implying that we can search for runs with a finite transient, followed by a periodic steady state. Thus, we create a polynomial-time graph algorithm based on solutions of bottleneck shortest-path problems to find an optimal cycle containing an accepting state. We implement our solution on the physical testbed shown in Figure 1. A preliminary version of this work appeared as Smith et al. (2010). Here we expand this preliminary version by including technical details, analysis of complexity, and more extensive experiments.

For simplicity of presentation, we assume that the robot moves among the vertices of an environment modeled as a graph. However, by using feedback controllers for facet reachability and invariance in polytopes (Habets and van Schuppen, 2004; Habets et al., 2006; Belta and Habets, 2006), the method developed in this paper can be easily applied for motion planning and control of a robot with 'realistic' continuous dynamics (e.g. a unicycle) traversing an environment partitioned using popular partitioning schemes such as triangulations and rectangular partitions.

The organization of the paper is as follows. In Section 2, we give some temporal-logic preliminaries. In Section 3, we formally state the robot-motion planning problem, and in Section 4 we present our solution. In Section 5 we present results of an experimental case study for a robot performing data-gathering missions in a road-network environment. Finally, in Section 6, we discuss some promising future directions.

2. Preliminaries

In this section we briefly review some aspects of LTL. LTL considers a finite set of variables Π, each of which can be

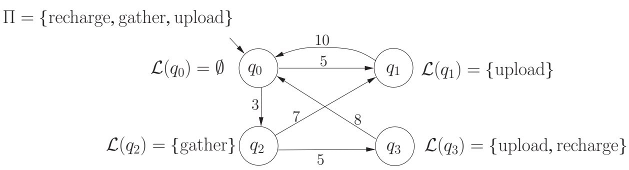

Fig. 2. An example of a weighted transition system. A correct run of the system is for instance $q_0q_2q_1q_0q_2q_3q_0\ldots$, producing the word $\emptyset$\{gather\}\{upload\}$\emptyset$\{gather\}\{upload,recharge\}$\emptyset\ldots$.

either true or false. The variables $\alpha_i \in \Pi$ are called *atomic propositions*. In the context of robots, propositions can capture properties such as 'the robot is located in region 1,' or 'the robot is recharging.'

Given a system model, LTL allows us to express the time evolution of the state of the system. We consider a type of finite model called the *weighted transition system*.

Definition 2.1 (Weighted Transition System). *A weighted transition system is a tuple $\mathcal{T} :=(Q, q_0, R, \Pi, \mathcal{L}, w)$, consisting of (i) a finite set of states Q; (ii) an initial state $q_0 \in Q$; (iii) a transition relation $R \subseteq Q \times Q$; (iv) a set of atomic propositions Π; (v) a labeling function $\mathcal{L} : Q \to 2^{\Pi}$; and (vi) a weight function $w : \mathbb{R} \to \mathbb{R}_{>0}$.*

We assume that the transition system is non-blocking, implying that there is a transition from each state. The transition relation has the expected definition: given that the system is in state $q_1 \in Q$ at time t_1, the system is in state q_2 at time $t_1+w\big((q_1, q_2)\big)$ if and only if $(q_1, q_2) \in R$. The labeling function defines for each state $q \in Q$, the set $\mathcal{L}(q)$ of all atomic propositions valid in q. For example, the proposition 'the robot is recharging' will be valid for all states $q \in Q$ containing recharging stations.

For our transition system we can define a *run* $r_{\mathcal{T}}$ to be an infinite sequence of states $q_0q_1q_2\ldots$ such that q_0 is the initial state, $q_i \in Q$, for all i, and $(q_i, q_{i+1}) \in R$, for all i. A run $r_{\mathcal{T}}$ defines a *word* $\mathcal{L}(q_0)\mathcal{L}(q_1)\mathcal{L}(q_2)\ldots$ consisting of sets of atomic propositions valid at each state. An example of a weighted transition system is given in Figure 2.

Definition 2.2 (Formula of LTL). *An LTL formula ϕ over the atomic propositions Π is defined inductively as follows:*

(i) $\top$ is a formula,
(ii) every atomic proposition $\alpha \in \Pi$ is a formula, and
(iii) if ϕ_1 and ϕ_2 are formulas, then $\phi_1 \vee \phi_2$, $\neg\phi_1$, $\mathbf{X}\,\phi_1$, and $\phi_1\,\mathbf{U}\,\phi_2$ are each formulas,

where $\top$ is a predicate true in each state of a system, $\neg$ (negation) and $\vee$ (disjunction) are standard Boolean connectives, and **X** *and* **U** *are temporal operators.*

LTL formulas are interpreted over infinite runs, as those generated by the transition system $\mathcal{T}$ from Definition 2.1. Informally, $\mathbf{X}\,\alpha$ states that at the next state of a run, proposition α is true (i.e. $\alpha \in \mathcal{L}(q_1)$). In contrast, $\alpha_1\,\mathbf{U}\,\alpha_2$ states that there is a future moment when proposition α_2 is true, and proposition α_1 is true at least until α_2 is true. From these temporal operators we can construct two other useful operators: Eventually (i.e. future), **F** defined as $\mathbf{F}\,\phi := \top\,\mathbf{U}\,\phi$, and Always (i.e. globally), **G**, defined as $\mathbf{G}\,\phi := \neg\,\mathbf{F}\,\neg\,\phi$. The formula $\mathbf{G}\,\alpha$ states that proposition α holds at all states of the run, and $\mathbf{F}\,\alpha$ states that α holds at some future time instance.

An LTL formula can be represented in an automata-theoretic setting as a *Büchi automaton*, defined as follows.

Definition 2.3 (Büchi Automaton). *A Büchi automaton is a tuple $\mathcal{B} :=(S, S_0, \Sigma, \delta, F)$, consisting of*

(i) a finite set of states S;
(ii) a set of initial states $S_0 \subseteq S$;
(iii) an input alphabet Σ;
(iv) a non-deterministic transition relation $\delta \subseteq S \times \Sigma \times S$; and
(v) a set of accepting (final) states $F \subseteq S$.

The semantics of Büchi automata are defined over infinite input words. Setting the input alphabet $\Sigma = 2^{\Pi}$, the semantics are defined over the words consisting of sets of atomic propositions, that is, those produced by a run of the transition system. Let $\omega = \omega_0\omega_1\omega_2\ldots$ be an infinite input word of automaton $\mathcal{B}$, where $\omega_i \in \Sigma$ for each $i \in \mathbb{N}$ (for example, the input $\omega = \mathcal{L}(q_0)\mathcal{L}(q_1)\mathcal{L}(q_2)\ldots$ could be a word produced by a run $q_0q_1q_2\ldots$ of the transition system $\mathcal{T}$).

A *run* of the Büchi automaton *over* an input word $\omega = \omega_0\omega_1\omega_2\ldots$ is a sequence $r_{\mathcal{B}} = s_0s_1s_2\ldots$, such that $s_0 \in S_0$, and $(s_i, \omega_i, s_{i+1}) \in \delta$, for all $i \in \mathbb{N}$.

Definition 2.4 (Büchi acceptance). *A word ω is accepted by the Büchi automaton $\mathcal{B}$ if and only if there exists a run $r_{\mathcal{B}}$ over ω so that $\inf(r_{\mathcal{B}}) \cap F \neq \emptyset$, where $\inf(r_{\mathcal{B}})$ denotes the set of states appearing infinitely often in run $r_{\mathcal{B}}$.*

The Büchi automaton allows us to determine whether or not the word produced by a run of the transition system satisfies an LTL formula. More precisely, for any LTL formula ϕ over a set of atomic propositions Π, there exists a Büchi automaton $\mathcal{B}_{\phi}$ with input alphabet 2^{Π} accepting all and only the infinite words satisfying formula ϕ (Vardi and Wolper, 1986). Translation algorithms were proposed in Vardi and Wolper (1994) and efficient implementations were developed in Gerth et al. (1995) and Gastin and Oddoux (2001). The size of the obtained Büchi automaton is, in general, exponential with respect to the size of the formula. However, many rich behaviors can be described using relatively small LTL formulas, and in these cases the exponential complexity is not prohibitive. An example of a Büchi automaton is given in Figure 3.

3. Problem statement and approach

Consider a single robot in an arbitrary environment, represented as a transition system (as defined in Section 2)

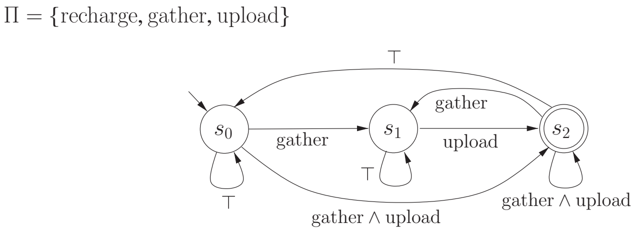

Fig. 3. A Büchi automaton corresponding to LTL formula ($\mathbf{G}\,\mathbf{F}\,$gather $\wedge\,\mathbf{G}\,\mathbf{F}\,$upload) over the alphabet Π. The illustration of the automaton is simplified. In fact, each transition labeled with $\top$ represents $|2^{\Pi}|$ transitions labeled with all different subsets of atomic propositions. Similarly, a transition labeled with gather represents $|2^{\Pi}|/2$ transitions labeled with all subsets of atomic propositions containing the proposition gather, etc.

$\mathcal{T} = (Q, q_0, R, \Pi, \mathcal{L}, w)$. A run in the transition system starting at q_0 defines a corresponding path of the robot in the environment. The time to take transition $(q_1, q_2) \in R$ (i.e. the time for the robot to travel from q_1 to q_2 in the environment) is given by $w(q_1, q_2)$.

To define our problem, we assume that there is an atomic proposition $\pi \in \Pi$, called the *optimizing proposition*. We consider LTL formulas of the form

$$\phi := \varphi \wedge \mathbf{G}\,\mathbf{F}\,\pi. \tag{1}$$

The formula φ can be any LTL formula over Π. The second part of the formula specifies that the proposition π must be satisfied infinitely often, and will simply ensure well-posedness of our optimization.

Let each run of $\mathcal{T}$ start at time $t = 0$, and assume that there is at least one run satisfying LTL formula (1). For each satisfying run $r_{\mathcal{T}} = q_0 q_1 q_2 \ldots$, there is a corresponding word of sets of atomic propositions $\omega = \omega_0\omega_1\omega_2 \ldots$, where $\omega_i = \mathcal{L}(q_i)$. Associated with $r_{\mathcal{T}}$ there is a sequence of time instances $\mathbb{T} := t_0 t_1 t_2 \ldots$, where $t_0 = 0$, and t_i denotes the time at which state q_i is reached $(t_{i+1} = t_i + w((q_i, q_{i+1})))$. From this time sequence we can extract all time instances at which the proposition π is satisfied. We let $\mathbb{T}_\pi$ denote the sequence of satisfying instances of the proposition π.

Our goal is to synthesize an infinite run $r_{\mathcal{T}}$ (i.e. a robot path) satisfying LTL formula (1), and minimizing the cost function

$$\mathcal{C}(r_{\mathcal{T}}) = \limsup_{i \to +\infty} \left(\mathbb{T}_\pi(i+1) - \mathbb{T}_\pi(i)\right), \tag{2}$$

where $\mathbb{T}_\pi(i)$ is the ith satisfying time instance of proposition π. Note that a finite cost in (2) ensures that $\mathbf{G}\,\mathbf{F}\,\pi$ is satisfied. Thus, the specification appears in ϕ merely to ensure that any satisfying run has finite cost. In summary, our goal is the following:

Problem Statement 3.1. *Determine an algorithm that takes as input a weighted transition system $\mathcal{T}$, an LTL formula ϕ over its set of atomic propositions in form* (1), *and an optimizing proposition π, and outputs a run $r_{\mathcal{T}}$ minimizing the cost $\mathcal{C}(r_{\mathcal{T}})$ in* (2).

We now make a few remarks, motivating this problem.

Remarks 3.2 (Comments on Problem Statement)**.** Cost function form: *The transition system produces infinite runs and the cost function* (2) *evaluates the steady-state time between satisfying instances of π. This form of the cost is motivated by persistent-monitoring tasks, where we seek to optimize the long-term behavior. In the upcoming sections we design an algorithm that minimizes the time to reach the optimal steady state: Thus, the runs produced will achieve the cost in* (2) *in finite time. In addition, in Remark 4.12 we discuss how we can optimize alternative cost functions that consider both transient and steady-state behavior.*

Expressivity of LTL formula (1): *Many interesting LTL specifications can be cast in the form of* (1). *For example, suppose that we want to minimize the time between satisfying instances of a disjunction of propositions $\vee_i \alpha_i$. We can write this in the form of formula* (1) *by defining a new proposition π that is satisfied at each state in which an α_i is satisfied.*

In addition, the LTL formula φ in (1) allows us to specify various rich robot-motion requirements. An example of such is global absence *($\mathbf{G}\,\neg\psi$, globally keep avoiding ψ),* response *($\mathbf{G}\,(\psi_1 \Rightarrow \mathbf{F}\,\psi_2)$, whenever ψ_1 holds true, ψ_2 will happen in the future),* reactivity *($\mathbf{G}\,\mathbf{F}\,\psi_1 \Rightarrow \mathbf{G}\,\mathbf{F}\,\psi_2$, if ψ_1 holds in the future for any time point, ψ_2 has to happen in the future for that time point as well),* sequencing *($\psi_1\,\mathbf{U}\,\psi_2\,\mathbf{U}\,\psi_3$, ψ_1 holds until ψ_2 happens, which holds until ψ_3 happens), and many others. For concrete examples, see Section 5.* □

4. Problem solution

In this section we describe our solution to Problem 3.1. We leverage ideas from the automata-theoretic approach to model checking.

4.1. The product automaton

Consider the weighted transition system $\mathcal{T} = (Q, q_0, R, \Pi, \mathcal{L}, w)$, and a proposition $\pi \in \Pi$. In addition, consider an LTL formula $\phi = \varphi \wedge \mathbf{G}\,\mathbf{F}\,\pi$ over Π in form (1), translated into a Büchi automaton $\mathcal{B}_\phi = (S, S_0, 2^{\Pi}, \delta, F)$. With these two components, we define a new object, which we call the *product automaton*, that is suitably defined for our problem.

Definition 4.1 (Product automaton)**.** *The product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{B}_\phi$ between the transition system $\mathcal{T}$ and the Büchi automaton $\mathcal{B}_\phi$ is defined as the tuple $\mathcal{P} := (S_{\mathcal{P}}, S_{\mathcal{P},0}, \delta_{\mathcal{P}}, F_{\mathcal{P}}, w_{\mathcal{P}}, S_{\mathcal{P},\pi})$, consisting of*

(i) a finite set of states $S_{\mathcal{P}} = Q \times S$,

(ii) a set of initial states $S_{\mathcal{P},0} = \{q_0\} \times S_0$,

(iii) a transition relation $\delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$, where $((q,s),(\bar{q},\bar{s})) \in \delta_{\mathcal{P}}$ if and only if $(q,\bar{q}) \in R$ and $(s, \mathcal{L}(q), \bar{s}) \in \delta$,

(iv) a set of accepting (final) states $F_{\mathcal{P}} = Q \times F$,
(v) a weight function $w_{\mathcal{P}} : \delta_{\mathcal{P}} \to \mathbb{R}_{>0}$, where $w_{\mathcal{P}}\big(((q,s),(\bar{q},\bar{s}))\big) = w\big((q,\bar{q})\big)$, for all $\big((q,s),(\bar{q},\bar{s})\big) \in \delta_{\mathcal{P}}$,
(vi) a set of states $S_{\mathcal{P},\pi} \subseteq S_{\mathcal{P}}$ in which the proposition π holds true. Thus, $(q,s) \in S_{\mathcal{P},\pi}$ if and only if $\pi \in \mathcal{L}(q)$.

The product automaton (as defined above) can be seen as a Büchi automaton with a trivial input alphabet. Since the alphabet is trivial, we omit it. Thus, we say that a run $r_{\mathcal{P}}$ in product automaton $\mathcal{P}$ is accepting if $\inf(r_{\mathcal{P}}) \cap F_{\mathcal{P}} \neq \emptyset$. An example product automaton is illustrated in Figure 4.

As in the transition system, we associate with each run $r_{\mathcal{P}} = p_0p_1p_2\ldots$, a sequence of time instances $\mathbb{T}_{\mathcal{P}} := t_0t_1t_2\ldots$, where $t_0 = 0$, and t_i denotes the time at which the ith vertex in the run is reached [$t_{i+1} = t_i + w_{\mathcal{P}}(p_i, p_{i+1})$]. From this time sequence we can extract a sequence $\mathbb{T}_{\mathcal{P},\pi}$, containing time instances t_i, where $p_i \in S_{\mathcal{P},\pi}$ (i.e. $\mathbb{T}_{\mathcal{P},\pi}$ is a sequence of satisfying instances of the optimizing proposition π in $\mathcal{T}$). The cost of a run $r_{\mathcal{P}}$ on the product automaton $\mathcal{P}$ [which corresponds to cost function (2) on transition system $\mathcal{T}$] is

$$\mathcal{C}_{\mathcal{P}}(r_{\mathcal{P}}) = \limsup_{i\to+\infty} \big(\mathbb{T}_{\mathcal{P},\pi}(i+1) - \mathbb{T}_{\mathcal{P},\pi}(i)\big). \qquad (3)$$

The product automaton can also be viewed as a weighted graph, where the states define vertices of the graph and the transitions define the edges. Thus, we at times refer to runs of the product automaton as *paths*. A *finite path* is then a finite fragment of an infinite path.

Each accepting run of the product automaton can be projected to a run of the transition system satisfying the LTL formula. Formally, we have the following.

Proposition 4.2 (Product Run Projection, Vardi and Wolper (1986))**.** *For any accepting run $r_{\mathcal{P}} = (q_0,s_0)(q_1,s_1)(q_2,s_2)\ldots$ of the product automaton $\mathcal{P}$, the sequence $r_{\mathcal{T}} = q_0q_1q_2\ldots$ is a run of $\mathcal{T}$ satisfying ϕ. Furthermore, the values of cost functions $\mathcal{C}_{\mathcal{P}}$ and $\mathcal{C}$ are equal for runs $r_{\mathcal{P}}$ and $r_{\mathcal{T}}$, respectively.*

Similarly, if $r_{\mathcal{T}} = q_0q_1q_2\ldots$ is a run of $\mathcal{T}$ satisfying ϕ, then there exists an accepting run $r_{\mathcal{P}} = (q_0,s_0)(q_1,s_1)(q_2,s_2)\ldots$ of the product automaton $\mathcal{P}$, such that the values of cost functions $\mathcal{C}$ and $\mathcal{C}_{\mathcal{P}}$ are equal.

Finally, we need to discuss the structure of an accepting run of a product automaton $\mathcal{P}$.

Definition 4.3 (Prefix–Suffix Structure)**.** *A* prefix *of an accepting run is a finite path from an initial state to an accepting state $f \in F_{\mathcal{P}}$ containing no other occurrence of f. A* periodic suffix *is an infinite run originating at the accepting state f reached by the prefix, and periodically repeating a finite path originating and ending at f, and containing no other occurrence of f (but possibly containing other vertices in $F_{\mathcal{P}}$). An accepting run is in* prefix–suffix *structure if it consists of a prefix followed by a periodic suffix.*

Intuitively, the prefix can be thought of as the transient, while the suffix is the steady-state periodic behavior.

Lemma 4.4 (Prefix–Suffix Structure)**.** *At least one of the accepting runs $r_{\mathcal{P}}$ of $\mathcal{P}$ that minimizes cost function $\mathcal{C}_{\mathcal{P}}(r_{\mathcal{P}})$ is in prefix–suffix structure.*

Proof: Let $r_{\mathcal{P}}$ be an accepting run that minimizes cost function $\mathcal{C}_{\mathcal{P}}(r_{\mathcal{P}})$ and is not in prefix–suffix structure. We will prove the existence of an accepting run $\rho_{\mathcal{P}}$ in prefix–suffix structure, such that $\mathcal{C}_{\mathcal{P}}(\rho_{\mathcal{P}}) \leq \mathcal{C}_{\mathcal{P}}(r_{\mathcal{P}})$. The idea behind the proof is that an accepting state must occur infinitely many times on $r_{\mathcal{P}}$. We then show that we can extract a finite path starting and ending at this accepting state, which can be repeated to form a periodic suffix whose cost is no larger than $\mathcal{C}_{\mathcal{P}}(r_{\mathcal{P}})$.

To begin, there exists a state $f \in F_{\mathcal{P}}$ occurring on $r_{\mathcal{P}}$ infinitely many times. Run $r_{\mathcal{P}}$ consists of a prefix $r_{\mathcal{P}}^{\text{fin}}$ ending at state f followed by an infinite, non-periodic suffix $r_{\mathcal{P}}^{\text{suf}}$ originating at the state f reached by the prefix. The suffix $r_{\mathcal{P}}^{\text{suf}}$ can be viewed as an infinite number of finite paths of form $fp_1p_2\ldots p_nf$, where $p_i \neq f$ for any $i \in \{1,\ldots,n\}$. Let $\mathcal{R}$ denote the set of all finite paths of the mentioned form occurring on the suffix $r_{\mathcal{P}}^{\text{suf}}$.

Note, that each path in the set $\mathcal{R}$ has to contain at least one occurrence of a state from $S_{\mathcal{P},\pi}$. To see this, assume by way of contradiction that there is a path $fp_1p_2\ldots p_nf$ that does not contain any state from $S_{\mathcal{P},\pi}$. The prefix $r_{\mathcal{P}}^{\text{fin}}$ followed by infinitely many repetitions of this path is indeed an accepting run of $\mathcal{P}$. However, if projected into run of $\mathcal{T}$, formula $\mathbf{G}\,\mathbf{F}\,\pi$ and thus also formula ϕ is violated, contradicting Proposition 4.2.

Similarly as for infinite paths, we associate with each finite path of length n a sequence of time instances $\mathbb{T}_{\mathcal{P}} := t_0t_1t_2\ldots t_n$, where $t_0 = 0$, and t_i denotes the time at which the ith vertex in the run is reached [$t_{i+1} = t_i + w_{\mathcal{P}}(p_i, p_{i+1})$]. From this time sequence we can extract a sequence $\mathbb{T}_{\mathcal{P},\pi}$, containing time instances t_i, where $p_i \in S_{\mathcal{P},\pi}$.

For each finite path $r \in \mathcal{R}$ with n states and k occurrences of a state from $S_{\mathcal{P},\pi}$ we define the following three costs

- $c^{f\rightsquigarrow}(r) = \mathbb{T}_{\mathcal{P},\pi}(0) - \mathbb{T}_{\mathcal{P}}(0)$
- $c(r) = \max_{i\in\{0,\ldots,k-1\}}\big(\mathbb{T}_{\mathcal{P},\pi}(i+1) - \mathbb{T}_{\mathcal{P},\pi}(i)\big)$
- $c^{\rightsquigarrow f}(r) = \mathbb{T}_{\mathcal{P}}(n) - \mathbb{T}_{\mathcal{P},\pi}(k)$.

Further, we define an equivalence relation $\sim$ over $\mathcal{R}$ as follows. Let $r_1, r_2 \in \mathcal{R}$. $r_1 \sim r_2$ if and only if

- $c^{f\rightsquigarrow}(r_1) = c^{f\rightsquigarrow}(r_2)$,
- $c(r_1) = c(r_2)$, and
- $c^{\rightsquigarrow f}(r_1) = c^{\rightsquigarrow f}(r_2)$.

Costs $c^{f\rightsquigarrow}$, c, and $c^{\rightsquigarrow f}$ can be extended to $c_{\sim}^{f\rightsquigarrow}$, $c_{\sim}$, and $c_{\sim}^{\rightsquigarrow f}$ in a natural way. For example, we define $c_{\sim}^{f\rightsquigarrow}([r]_{\sim}) = c^{f\rightsquigarrow}(r)$, where $r \in [r]_{\sim}$. The other two costs are defined analogously.

Let us extract a set $\mathcal{R}^{\inf}/_{\sim}$ from the set of equivalence classes $\mathcal{R}/_{\sim}$ such that each class in $\mathcal{R}^{\inf}/_{\sim}$ is infinite or contains a finite path that is repeated in $r_{\mathcal{P}}$ infinitely many

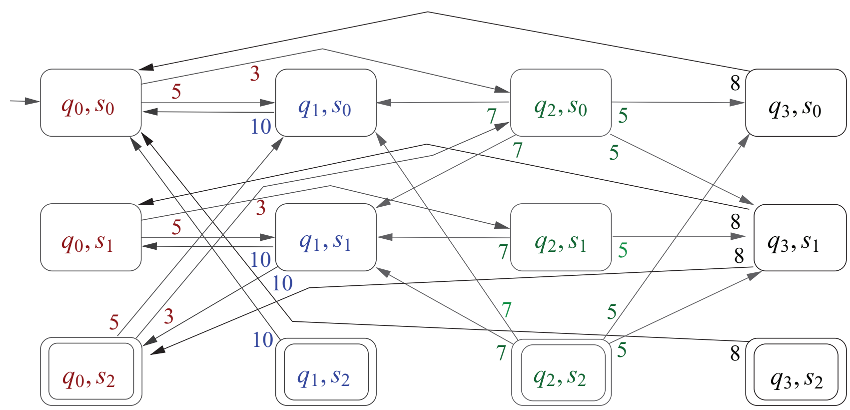

Fig. 4. Product automaton between the transition system in Figure 2 and the Büchi automaton in Figure 3.

times. As a consequence, for each class $[r]_\sim$ in $\mathcal{R}^{\inf}/_\sim$, it holds that $c_\sim([r]_\sim) \leq \mathcal{C}_\mathcal{P}(r_\mathcal{P})$. The set $\mathcal{R}/_\sim$ is finite, because there is only a finite number of different values of costs. Furthermore, accepting run $r_\mathcal{P}$ is infinite and thus $\mathcal{R}^{\inf}/_\sim$ is non-empty.

Let $[\rho]_\sim \in \mathcal{R}^{\inf}/_\sim$ now be a class such that $c_\sim^{f\rightsquigarrow}([\rho]_\sim)$ is minimal among the classes from $\mathcal{R}^{\inf}/_\sim$.

Each time a finite path in $[\rho]_\sim$ appears in $r_\mathcal{P}$, it is followed by another finite path. So, infinitely many times the 'following' path comes from a class $([r]_\sim) \in \mathcal{R}^{\inf}/_\sim$. Then, we must have $c^{\rightsquigarrow f}([\rho]_\sim) + c^{f\rightsquigarrow}([r]_\sim) \leq \mathcal{C}_\mathcal{P}(r_\mathcal{P})$. But, $c^{f\rightsquigarrow}([r]_\sim) \geq c^{\rightsquigarrow f}([\rho]_\sim)$, and thus $c^{\rightsquigarrow f}([\rho]_\sim) + c^{f\rightsquigarrow}([\rho]_\sim) \leq \mathcal{C}_\mathcal{P}(r_\mathcal{P})$.

Thus we can build the run $\rho_\mathcal{P}$ as the prefix $r_\mathcal{P}^{\text{fin}}$ followed by a periodic suffix $\rho_\mathcal{P}^{\text{suf}}$, which is obtained by infinitely many repetitions of an arbitrary path $\rho \in [\rho]_\sim$. $\rho_\mathcal{P}$ is in prefix–suffix structure and for its suffix $\rho_\mathcal{P}^{\text{suf}}$ it also holds that $\mathcal{C}_\mathcal{P}(\rho_\mathcal{P}) = \max_{i\in\mathbb{N}} \left(\mathbb{T}_{\mathcal{P},\pi}(i+1) - \mathbb{T}_{\mathcal{P},\pi}(i+1)\right) = \max\left(c(\rho), c^{f\rightsquigarrow}(\rho) + c^{\rightsquigarrow f}(\rho)\right) \leq \mathcal{C}_\mathcal{P}(r_\mathcal{P})$. ■

Definition 4.5 (Suffix Cost)**.** *The cost of the suffix $p_0 p_1 \ldots p_n p_0 p_1 \ldots$ of a run $r_\mathcal{P}$ is defined as follows. Let $t_{0,0}, t_{0,1}, \ldots, t_{0,n}, t_{1,0}, t_{1,1} \ldots$ be the sequence of times at which the vertices of the suffix are reached on run $r_\mathcal{P}$. Extract the sub-sequence $\mathbb{T}_\mathcal{P}^{\text{suf}}$ of times $t_{i,j}$, where $p_j \in S_{\mathcal{P},\pi}$ (i.e. the satisfying instances of proposition π in transition system $\mathcal{T}$). Then, the cost of the suffix is*

$$\mathcal{C}_\mathcal{P}^{\text{suf}}(r_\mathcal{P}) = \max_{i\in\mathbb{N}}(\mathbb{T}_\mathcal{P}^{\text{suf}}(i+1) - \mathbb{T}_\mathcal{P}^{\text{suf}}(i)).$$

From the definition of the product automaton cost $\mathcal{C}_\mathcal{P}$ and the suffix cost $\mathcal{C}_\mathcal{P}^{\text{suf}}$ we obtain the following result.

Lemma 4.6 (Cost of a Run)**.** *Given a run $r_\mathcal{P}$ in prefix–suffix structure and its suffix $p_0 p_1 p_2 \ldots p_n p_0 p_1 \ldots$, the value of the cost function $\mathcal{C}_\mathcal{P}(r_\mathcal{P})$ is equal to the cost of the suffix $\mathcal{C}_\mathcal{P}^{\text{suf}}(r_\mathcal{P})$.*

Our aim is to synthesize a run $r_\mathcal{T}$ of $\mathcal{T}$ minimizing the cost function $\mathcal{C}(r_\mathcal{T})$ and ensuring that the word produced by this run will be accepted by $\mathcal{B}$. This goal now translates to generating a run $r_\mathcal{P}$ of $\mathcal{P}$, such that the run satisfies the Büchi condition $F_\mathcal{P}$ and minimizes cost function $\mathcal{C}_\mathcal{P}(r_\mathcal{P})$. Furthermore, to find a satisfying run $r_\mathcal{P}$ that minimizes $\mathcal{C}_\mathcal{P}(r_\mathcal{P})$, it is enough to consider runs in prefix–suffix structure (see Lemma 4.4). From Lemma 4.6 it follows that the whole problem reduces to finding a periodic suffix $r_\mathcal{P}^{\text{suf}} = f p_1 p_2 \ldots p_n f p_1 \ldots$ in $\mathcal{P}$, such that:

(i) f is reachable from an initial state in $S_{\mathcal{P},0}$,
(ii) $f \in F_\mathcal{P}$ (i.e. f is an accepting state), and
(iii) the cost of the suffix $r_\mathcal{P}^{\text{suf}}$ is a minimum among all the suffixes satisfying (i) and (ii).

Finally, we can find the shortest prefix in $\mathcal{P}$ that starts at an initial state in $S_{\mathcal{P},0}$ and ends at the state f in the suffix $r_\mathcal{P}^{\text{suf}}$. By concatenating the prefix and suffix, we obtain an optimal run in $\mathcal{P}$. By projecting the optimal run to $\mathcal{T}$, via Proposition 4.2, we obtain a solution to our stated problem.

4.2. Graph algorithm for shortest-bottleneck cycles

We now focus on finding an optimal suffix in the product automaton. We cast this problem as a path optimization on a graph. To do this, let us define some terminology.

A graph $G = (V, E, w)$ consists of a vertex set V, an edge set $E \subseteq V \times V$, and a weight function $w : E \to \mathbb{R}_{>0}$. A *cycle* in G is a vertex sequence $v_1 v_2 \ldots v_k v_{k+1}$, such that $(v_i, v_{i+1}) \in E$ for each $i \in \{1, \ldots, k\}$, and $v_1 = v_{k+1}$. Given a vertex set $S \subseteq V$, consider a cycle $c = v_1 \ldots v_k v_{k+1}$ containing at least one vertex in S. Let $(i_1, i_2, \ldots, i_s)$ be the ordered set of vertices in c that are elements of S (i.e. indices with order $i_1 < i_2 < \cdots < i_m$, such that $v_j \in S$ if and only if $j \in \{i_1, i_2, \ldots, i_s\}$). Then, the *S-bottleneck length* is

$$\max_{\ell\in\{1,\ldots,s\}} \sum_{j=i_\ell}^{i_{\ell+1}-1} w(e_j),$$

where $i_{s+1} = i_1$. In words, the S-bottleneck distance is defined as follows.

Definition 4.7 (S-Bottleneck Length)**.** *Given a graph $G = (V, E, w)$, and a vertex set $S \subseteq V$, the S-bottleneck length*

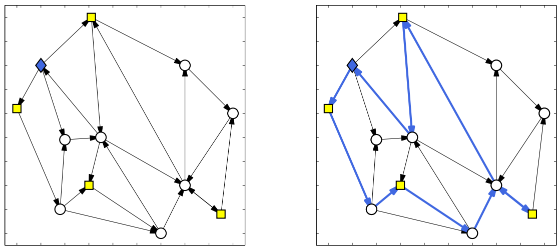

Fig. 5. The left figure shows a possible input to the MIN-BOTTLENECK-CYCLE algorithm. In the directed graph, the edge weights are given by the Euclidean distance. The set F is a singleton given by the diamond. The vertices in S are drawn as yellow squares. The right figure shows an optimal cycle with a minimum S-bottleneck length using thick edges.

of a cycle in G is the maximum distance between successive appearances of an element of S on the cycle.[1]

The *bottleneck length* of a cycle is defined as the maximum length edge on the cycle (Korte and Vygen, 2007). In contrast, the S-bottleneck length measures distances between vertices in S. With the terminology in place, our goal is to solve the following *constrained S-bottleneck problem*.

Problem Statement 4.8. *Given a graph $G = (V, E, w)$, and two vertex sets $F, S \subseteq V$, find a cycle in G containing at least one vertex in F, with minimum S-bottleneck length.*

Our solution, shown in Algorithm 1, is called the MIN-BOTTLENECK-CYCLE algorithm. It utilizes Dijkstra's algorithm (Korte and Vygen, 2007) for computing shortest paths between pairs of vertices (called SHORTEST-PATH), and a slight variation of Dijkstra's algorithm for computing shortest-bottleneck paths between pairs of vertices (called SHORTEST-BOT-PATH).

SHORTEST-PATH takes as inputs a graph $G = (V, E, w)$, a set of source vertices $A \subseteq V$, and a set of destination vertices $B \subseteq V$. It outputs a distance matrix $D \in \mathbb{R}^{|A| \times |B|}$, where the entry $D(i, j)$ gives the shortest-path distance from A_i to B_j. It also outputs a predecessor matrix $P \in V^{|A| \times |V|}$, where $P(i, j)$ is the predecessor of j on a shortest path from A_i to V_j. For a vertex $v \in V$, the shortest path from v to v is defined as the shortest cycle containing v. If there does not exist a path between vertices, then the distance is $+\infty$. SHORTEST-BOT-PATH has the same inputs as SHORTEST-PATH, but it outputs paths that minimize the maximum edge length, rather than the sum of edge lengths.

Figure 5 (left) shows an example input to the algorithm. The graph contains 12 vertices, with one vertex (diamond) in F, and four vertices (square) in S. Figure 5 (right) shows the optimal solution as produced by the algorithm. The bottleneck occurs between the square vertices immediately before and after the diamond vertex.

Algorithm 1: MIN-BOTTLENECK-CYCLE(G, S, F)

Input: A directed graph G, and vertex subsets F and S
Output: A cycle in G that contains at least one vertex in F and minimizes the S-bottleneck distance.

1 Compute shortest paths between vertices in S:

$$(D, P) \leftarrow \text{SHORTEST-PATH}(G, S, S).$$

2 Define a graph G_S with vertices S and adjacency matrix D.

3 Compute shortest S-bottleneck paths between vertices in S:

$$(D_{\text{bot}}, P_{\text{bot}}) \leftarrow \text{SHORTEST-BOT-PATH}(G_S, S, S).$$

4 Compute shortest paths from each vertex in F to each vertex in S, and from each vertex in S to each vertex in F:

$$(D_{F\to S}, P_{F\to S}) \leftarrow \text{SHORTEST-PATH}(G, F, S)$$
$$(D_{S\to F}, P_{S\to F}) \leftarrow \text{SHORTEST-PATH}(G, S, F).$$

Set $D_{F\to S}(i, j) = 0$ and $D_{S\to F}(j, i) = 0$ for all i, j such that $F_i = S_j$.

5 For each triple $(f, s_1, s_2) \in F \times S \times S$, set

$$C(f, s_1, s_2) := \begin{cases} D_{F\to S}(f, s_1) \\ \quad + D_{S\to F}(s_2, f) & \text{if } f \neq s_1 = s_2 \\ \max\{D_{F\to S}(f, s_1) \\ \quad + D_{S\to F}(s_2, f), D_{\text{bot}}(s_1, s_2)\}, & \text{otherwise.} \end{cases}$$

6 Find the triple (f^*, s_1^*, s_2^*) that minimizes $C(f, s_1, s_2)$.

7 If the minimum cost is $+\infty$, then output 'no cycle exists.' Else, output cycle by extracting the path from f^* to s_1^* using $P_{F\to S}$, the path from s_1^* to s_2^* using P_{bot} and P, and the path from s_2^* to f^* using $P_{S\to F}$.

In the algorithm, one has to take special care that cycle lengths are computed properly when $f = s_1$, $s_1 = s_2$, or $f = s_2$. This is done by setting some entries of $D_{F\to S}$ and $D_{S\to F}$ to zero in step 4, and by defining the cost differently when $f \neq s_1 = s_2$ in step 5. In the following theorem we show the correctness of the algorithm.

Theorem 4.9 (MIN-BOTTLENECK-CYCLE Optimality). *The* MIN-BOTTLENECK-CYCLE *algorithm solves the constrained S-bottleneck problem (Problem 4.8).*

Proof: Every valid cycle must contain at least one element from F and at least one element from S. Let $c := v_1 v_2 \ldots v_k v_1$, be a valid cycle, and without loss of generality let $v_1 \in F$. From this cycle we can extract the triple $(v_1, v_a, v_b) \in F \times S \times S$, where $v_a, v_b \in S$, and $v_i \notin S$ for all $i < a$ and for all $i > b$. (Note that, $a = b = 1$ is possible.)

Consider the cycle c with corresponding triple $(f, s_1, s_2) := (v_1, v_a, v_b)$ as defined above, and let $L(c)$ denote its S-bottleneck length. It is straightforward to verify, using the definition of the S-bottleneck length, that $L(c) \geq C(f, s_1, s_2)$.

The cycle computed for the triple (f, s_1, s_2) in step 5 (as given by the four predecessor matrices) takes the shortest path from f to s_1, the shortest S-bottleneck path from s_1 to s_2, and the shortest path from s_2 to f. However, the shortest path from f to s_1 (and from s_2 to f) may contain other vertices from S. Thus, the S-bottleneck length of this cycle, denoted $L(f, s_1, s_2)$, satisfies

$$L(f, s_1, s_2) \leq C(f, s_1, s_2) \leq L(c), \qquad (4)$$

implying that $C(f, s_1, s_2)$ upper bounds the length of the computed cycle. However, if we take c to be a cycle with minimum length, then necessarily $L(c) \leq L(f, s_1, s_2)$. Hence, equation (4) implies that for an optimal cycle, $L(f, s_1, s_2) = C(f, s_1, s_2) = L(c)$. Thus, by minimizing the cost function in step 5 we compute the minimum length cycle. ■

Computational complexity. Finally, we characterize the computational complexity of the MIN-BOTTLENECK-CYCLE algorithm. Let n, m, n_S, and n_F be the number of vertices (edges) in the sets V, E, S, and F, respectively. Dijkstra's algorithm can be implemented to compute the shortest paths from a source vertex $v \in V$, to all other vertices in V in $O(n \log n + m)$ run time. Thus, for sparse graphs (which includes many transition systems), the run time is $O(n \log n)$.

Proposition 4.10 (MIN-BOTTLENECK-CYCLE Run Time)**.** *The run time of the* MIN-BOTTLENECK-CYCLE *algorithm is* $O\big((n_S + n_F)(n \log n + m + n_S^2)\big)$. *Thus, in the worst-case, the run time is* $O(n^3)$. *For sparse graphs with* $n_S, n_F \ll n$, *the run time is* $O\big((n_S + n_F)\, n \log n\big)$.

Proof: We simply look at the run time of each step in the algorithm. Step 1 requires n_S calls to Dijkstra's algorithm, and has run time $O(n_S(n \log(n) + m))$. Step 3 requires n_S calls to Dijkstra's algorithm on a smaller graph $G_S = (S, E_S, w_S)$, and has run time $O(n_S(n_S \log(n_S) + |E_S|))$. Step 4 has run time $O(n_F(n \log(n) + m))$. Finally, steps 5 and 6 require searching over all $n_F \cdot n_S^2$ possibilities, and they have run times $O(n_F n_S^2)$. Since $|E_S| \leq n_S^2$, the run time in general is given by $O\big((n_S + n_F)(n \log n + m + n_S^2)\big)$. ■

4.3. *The* OPTIMAL-RUN *Algorithm*

We are now ready to combine the results from the previous section to present a solution to Problem 3.1. The solution, the OPTIMAL-RUN algorithm, is summarized in Algorithm 2.

The correctness of the OPTIMAL-RUN algorithm follows directly from Lemma 4.4, Theorem 4.9, and Proposition 4.2.

Algorithm 2: OPTIMAL-RUN($\mathcal{T}, \phi$)

Input: A weighted transition system $\mathcal{T}$, and temporal-logic specification ϕ in form (1).
Output: A run in $\mathcal{T}$ that satisfies ϕ and minimizes (2).

1 Convert ϕ to a Büchi automaton $\mathcal{B}_\phi$.
2 Compute the product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{B}_\phi$.
3 Compute the cycle MIN-BOTTLENECK-CYCLE($G, S_{\mathcal{P},\pi}, F_{\mathcal{P}}$), where $G = (S_{\mathcal{P}}, \delta_{\mathcal{P}}, w_{\mathcal{P}})$.
4 Compute a shortest path from $S_{\mathcal{P},0}$ to the cycle.
5 Project the complete run (path and cycle) to a run on $\mathcal{T}$ using Proposition 4.2.

Theorem 4.11 (Correctness of OPTIMAL-RUN)**.** *The* OPTIMAL-RUN *algorithm solves Problem 3.1.*

Remark 4.12 (Alternative Cost for Optimizing Prefix)**.** *The* OPTIMAL-RUN *algorithm optimizes the cost of the repeated suffix. For the prefix, we simply find the shortest path from an initial state to the suffix. However, the cost of the prefix is not optimized. This is due to the fact that the cost function* $\mathcal{C}$ *was chosen with persistent-monitoring tasks in mind, where the long-term behavior is of interest. However, in some applications, the transient behavior may be of interest. In this case we can define an alternative cost function* $\mathcal{C}'$*:*

$$\mathcal{C}'(r_{\mathcal{T}}) = \sup_{i \in \mathbb{N}} \left(\mathbb{T}_\pi(i+1) - \mathbb{T}_\pi(i) \right). \qquad (5)$$

Then, we can consider two alternative problems: (i) Find a run $r_{\mathcal{T}}$ *minimizing the cost* $\mathcal{C}'(r_{\mathcal{T}})$*; or (ii) Find a run* $r_{\mathcal{T}}$ *that minimizes the cost* $\mathcal{C}'(r_{\mathcal{T}})$ *among all the runs minimizing the cost* $\mathcal{C}(r_{\mathcal{T}})$*. Both problems can be solved by slightly modifying the* MIN-BOTTLENECK-CYCLE *algorithm.*

We can extend the proof of Lemma 4.4 to show that there is a run in prefix–suffix form that minimizes $\mathcal{C}'$*. By appropriately defining the cost of the prefix, we can also show that the cost* C' *is equal to the maximum of the prefix cost and the suffix cost. Then, to solve problems (i) and (ii) we add a step to the* MIN-BOTTLENECK-CYCLE *algorithm in which we compute the shortest-bottleneck path from each initial state* $v_0 \in V$ *to each state* $s \in S$*. We record the cost of the path from* v_0 *to* s *as* $C_p(v_0, s)$*. For problem (i) we alter step 6 to find the tuple* $(v_0^*, f^*, s_1^*, s_2^*)$ *that minimizes* $\max\{C_p(v_0, s_1), C(f, s_1, s_2)\}$*. For problem (ii) we alter step 6 to find the tuple* $(v_0^*, f^*, s_1^*, s_2^*)$ *that minimizes* $C_p(v_0, s_1)$ *among the tuples that minimize* $C(f, s_1, s_2)$*. Finally, we remove step 4 from the* OPTIMAL-RUN *algorithm.* □

Computational Complexity of Optimal-Run. The worst-case computational complexity of the OPTIMAL-RUN algorithm can be characterized as follows. Any LTL formula ϕ can be translated into a Büchi automaton in time $2^{O(|\phi|)}$ computation time (Baier et al., 2008).[2] The worst-case size of the Büchi automaton (i.e. the number of states) is also

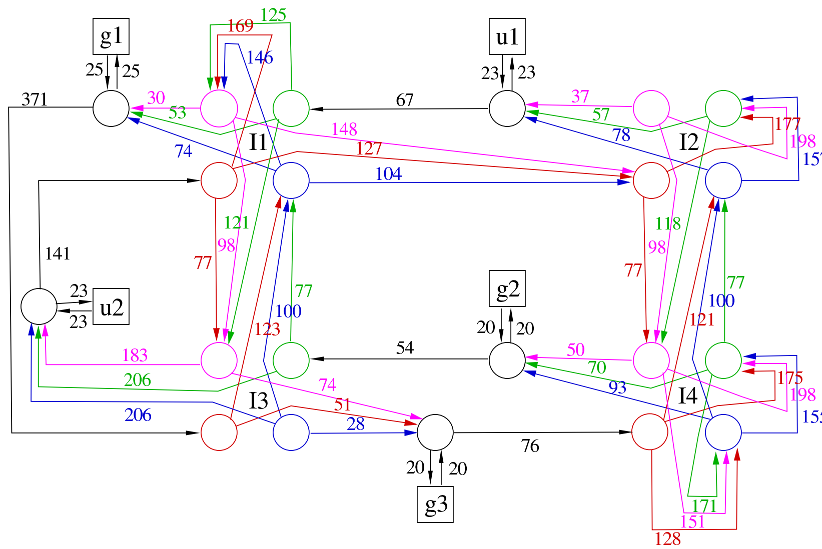

Fig. 6. The weighted transition system for the road network in Figure 1.

$2^{O(|\phi|)}$. The size of the product obtained in step 2 of the OPTIMAL-RUN algorithm is therefore $O(|T| \cdot 2^{O(|\phi|)})$, where $|T|$ is the number of states in the transition system. Then, from Proposition 4.10, the worst-case complexity of the OPTIMAL-RUN algorithm is $O(|T|^3 \cdot 2^{O(|\phi|)})$.

Thus, the worst-case complexity is quite restrictive, being exponential in the size of the LTL formula. However, many rich robot behaviors can be described using relatively small LTL formulas. In addition, the time required to compute the Büchi automaton, and the size of the Büchi automaton, are frequently much smaller than the worst-case bound. In the following section we show that the proposed approach can be used to generate robot-motion plans that satisfy rich requirements in complex environments.

5. Case studies and experiments

In this section, we present an implementation of the OPTIMAL-RUN algorithm on a physical testbed. We focus on a data-gathering mission in which a robot must repeatedly gather data at interesting locations, and then upload it at designated sites. We also present a case study that outlines several different robot missions, and how they can be expressed in LTL. The purpose of this section is to (i) demonstrate the utility of the proposed approach in generating complex motion plans; (ii) illustrate the expressivity of LTL and the class of optimizations considered in this paper; (iii) highlight the subtleties and challenges that arise when expressing a desired behavior in LTL; and, (iv) provide numerical data on the complexity and computation time of our proposed approach.

5.1. The road-network testbed

We implemented the OPTIMAL-RUN algorithm on the road network shown in Figure 1. This network is a collection of roads, intersections, and parking lots (which serve as data-gather and upload locations), connected by a simple set of rules (e.g. a road connects two, not necessarily different, intersections and the parking lots can only be located on the side of a road). The city is easily reconfigurable through re-taping. The robot used is a Khepera III miniature car. The car can sense its entry into an intersection from a road, its entry into a road from an intersection, when it passes in front of a parking lot, when it is correctly parked in a parking space, and when an obstacle is dangerously close. The car is programmed with motion and communication primitives allowing it to safely drive on a road, turn in an intersection, and park. The car can communicate through Wi-Fi with a desktop computer, which is used as an interface to the user (i.e. to enter the specification) and to perform all the computation necessary to generate the control strategy. Once computed, this is sent to the car, which executes the task autonomously by interacting with the environment.

A model of the motion of the car in the road network using a weighted transition system (Definition 2.1) is depicted in Figure 6 and proceeds as follows. The set of states Q is the set of labels assigned to the intersections, parking lots, and branching points between the roads and parking lots. The transition relation R shows how the regions are connected and the transitions' labels give distances between them (measured in inches). In our testbed the robot moves at constant speed v, and thus the distances and travel times are equivalent. For these experiments, the robot can only move on the right-hand lane of a road and it cannot make a U-turn at an intersection. To capture this, we model each intersection as four different states. Note that, in reality, each state in Q has an associated set of motion primitives, and the selection of a motion primitive (e.g. *go_straight*, *turn_right*) determines the transition to one unique next state. This motivates our assumption that the weighted transition system from Definition 2.1 is deterministic, and therefore its inputs can be removed.

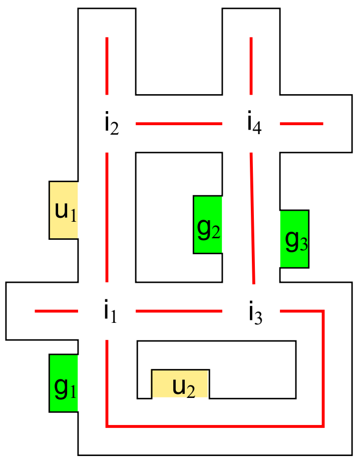

Fig. 7. Schematic illustration of the road network. For each road, the median is shown as a red line. The robot must drive on the right-hand side of the road (i.e. the right-hand side of the median). Intersections are labelled i_1 through i_4. Data-gather locations, labeled g_1, g_2, and g_3, are shaded green (dark). Data upload locations, labeled u_1 and u_2, are shaded yellow (light).

5.2. *An experimental case study of the data-gathering missions*

In our experiments, we have consider data-gathering missions of the following form. Parking lots u_1 and u_2 in Figure 1 are data-upload locations (light-shaded regions in Figure 7) and parking lots g_1, g_2, and g_3 are data-gather locations (dark-shaded regions in Figure 7). The optimizing proposition π in LTL formula (1) is

$$\pi := u_1 \vee u_2, \tag{6}$$

that is we want to minimize the time between data uploads. Assuming infinite runs of the robot in the environment, we are able to describe the motion requirements as LTL formulas, where atomic propositions are simply names of the parking lots.

In this section we describe seven different data-gathering cases. Each case describes a data-gathering mission, and the cases are roughly ordered in increasing complexity. For each case we have computed the optimal run according to the OPTIMAL-RUN algorithm, and we have implemented the run on our testbed. In Table 1 we summarize the key statistics for each case. The summary data consists of (i) the maximum distance between uploads on the optimal path, (ii) the maximum time between uploads observed in the robot experiment, (iii) the number of states in the Büchi automaton, (iv) the number of states in the product automaton, (v) the time to translate the LTL formula into a Büchi automaton, and (vi) the time to compute the optimal path in the product automaton. The computations were performed on a desktop computer with a 2.8 GHz quad core processor and 8 GB of RAM. We utilized the LTL2BA software by Gastin and Oddoux (2001) to translate an LTL formula to a Büchi automaton.

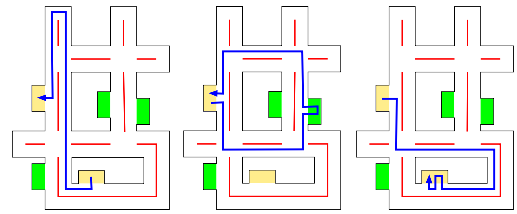

Fig. 8. The robot path (shown as lines with arrows) for Case A. Green (dark shaded) areas are data-gathering locations, and yellow (light shaded) areas are upload locations. The robot periodically follows the path, which is composed of the illustrated fragments as seen from left to right.

Case A. To begin, let us consider the following mission. Repeatedly visit data-gather locations (g_1, g_2, or g_3) to gather data and repeatedly visit upload locations (u_1 or u_2) to upload data. The objective is to minimize the time between visits to data upload locations, and therefore the optimizing proposition π is given by the LTL formula from Equation (6). We can specify this behavior as the following LTL formula:

$$\phi_A := \mathbf{G}\,\mathbf{F}\,(g_1 \vee g_2 \vee g_3) \wedge \mathbf{G}\,\mathbf{F}\,\pi.$$

Using the OPTIMAL-RUN algorithm, we compute the robot path shown in Figure 8. This figure is interpreted as follows. The figure consists of a sequence of environment snapshots, read from left to right. Each snapshot shows a robot path as a line which starts and ends at a data-upload location. The starting point of the robot path on the $(i+1)$th snapshot is given by endpoint of the path on the ith snapshot. The endpoint of the final snapshot connects with the starting point of the first snapshot. Thus, the infinite robot path is obtained by cycling through these snapshots.

The time to run the algorithm and the value of the cost function are summarized in Table 1.

Case B. Looking at the results of Case A, we see that the robot does not always gather new data before visiting an upload location (in Figure 8 the robot visits two upload locations in a row). To eliminate this behavior, we should specify that the robot can only visit a data-upload location if it has just gathered data. This can be specified as follows:

$$\phi_B := \phi_A \wedge \mathbf{G}\big((u_1 \vee u_2) \Rightarrow \mathbf{X}((\neg u_1 \wedge \neg u_2)\ \mathbf{U}\,(g_1 \vee g_2 \vee g_3))\big).$$

The corresponding robot path is shown in Figure 9.

Case C. In some situations the data-gather locations g_1, g_2, and g_3 may contain different information, and thus it is beneficial to periodically visit each of them. To specify this, we can build on Case B and write the following formula:

$$\phi_C := \mathbf{G}\,\mathbf{F}\,g_1 \wedge \mathbf{G}\,\mathbf{F}\,g_2 \wedge \mathbf{G}\,\mathbf{F}\,g_3 \wedge \mathbf{G}\,\mathbf{F}\,\pi \wedge \mathbf{G}\big((u_1 \vee u_2) \Rightarrow \mathbf{X}((\neg u_1 \wedge \neg u_2)\ \mathbf{U}\,(g_1 \vee g_2 \vee g_3))\big).$$

Table 1. A summary of the seven data-gathering cases.

Case	Length m	Time (min) travel	# of states Büchi	# of states product	Time (sec) LTL to Büchi	Time (sec) computation
A	6.23	2.5	3	78	∼1	∼1
B	6.23	2.5	7	182	∼1	∼1
C	9.13	3.6	11	286	∼1	∼1
D	9.13	3.6	17	442	∼1	∼1
E	9.13	3.6	49	1274	∼1	∼8
F	10.48	4.1	34	884	∼1	∼2
G	9.50	3.7	34	884	∼1	∼2

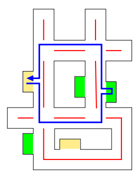

Fig. 9. The robot path for Case B. Note that the robot does not visit two upload locations without visiting a download location in between. The value of the optimization function is the same as in Case A (6.23 m).

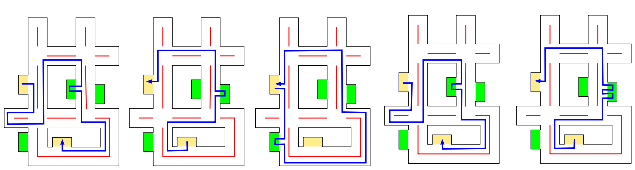

Fig. 10. The robot path for Case C. Note that the robot visits all three download locations and only visits an upload location if it has just gathered data. Extension 1 shows the robot executing the first two snapshots.

Using the Optimal-Run algorithm, the computed path of the robot is shown in Figure 10. Extension 1 shows the robot's execution of this path. The video ends at the completion of the second snapshot in Figure 10. The time to run the algorithm and the value of the cost function are summarized in Table 1. Note that this more restrictive formula results in a larger cost function value than in Cases A or B.

Case D. Notice that in the last snapshot of Figure 10, the robot visits data-gather location g_3 twice in a row. Such behavior does not increase the value of the cost function, but may not be desirable in some circumstances. We can eliminate this behavior by specifying that the robot must visit an upload location after gathering data:

$$\phi_D := \phi_C \wedge \mathbf{G}\,((\mathsf{g}_1 \vee \mathsf{g}_2 \vee \mathsf{g}_3) \Rightarrow \mathbf{X}\,(\neg(\mathsf{g}_1 \vee \mathsf{g}_2 \vee \mathsf{g}_3)\ \mathbf{U}\,(\mathsf{u}_1 \vee \mathsf{u}_2)))\,.$$

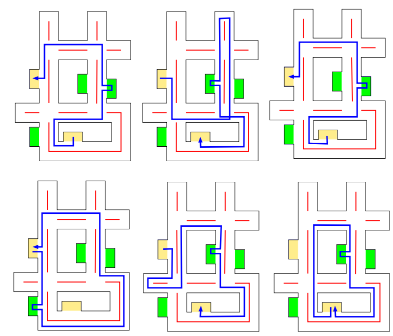

Fig. 11. The robot path for case D. One may observe that snapshots 1, 2, and 6 are redundant and it would be sufficient to periodically repeat snapshots 3, 4, and 5 to satisfy the formula. Such 'aesthetic' changes do not improve the value of cost function.

The new path of the robot is shown in Figure 11. Note from Table 1 that the maximum distance between uploads does not change from Case C to Case D.

Case E. Now, suppose that we would like to require an equal number of visits to each data-gather location. We can observe that in Case D, some of the gather locations are visited more frequently than the others. To formalize this idea of equality, we can specify an order in which the data-gather locations should be visited: g_3, g_1, and g_2, in this order. The syntax for specifying this order is somewhat complicated, and involves nested 'until' operators. The specification becomes

$$\begin{aligned}\phi_E :=& (\neg\mathsf{g}_1 \wedge \neg\mathsf{g}_2)\ \mathbf{U}\,\mathsf{g}_3 \wedge\\ & \mathbf{G}\,(\mathsf{g}_3 \Rightarrow \mathbf{X}((\neg\mathsf{g}_2 \wedge \neg\mathsf{g}_3)\ \mathbf{U}\,(\mathsf{g}_1 \wedge \mathbf{X}\,((\neg\mathsf{g}_1 \wedge \neg\mathsf{g}_3)\\ & \ \mathbf{U}\,(\mathsf{g}_2 \wedge \mathbf{X}\,((\neg\mathsf{g}_1 \wedge \neg\mathsf{g}_2)\ \mathbf{U}\,\mathsf{g}_3)))))) \wedge\\ & \mathbf{G}\,((\mathsf{u}_1 \vee \mathsf{u}_2) \Rightarrow \mathbf{X}\,((\neg\mathsf{u}_1 \wedge \neg\mathsf{u}_2)\ \mathbf{U}\,(\mathsf{g}_1 \vee \mathsf{g}_2 \vee \mathsf{g}_3))) \wedge\\ & \mathbf{G}\,((\mathsf{g}_1 \vee \mathsf{g}_2 \vee \mathsf{g}_3) \Rightarrow \mathbf{X}\,(\neg(\mathsf{g}_1 \vee \mathsf{g}_2 \vee \mathsf{g}_3)\ \mathbf{U}\,(\mathsf{u}_1 \vee \mathsf{u}_2))) \wedge\\ & \mathbf{G}\,\mathbf{F}\,\pi\,.\end{aligned}$$

The robot path for this case is shown in Figure 12.

Case F. We can also specify 'safety' constraints for the robot. For example, consider the objective of Case D with the additional constraint that the road connecting i_4 to i_2 (illustrated in pink in Figure 13) should be avoided. In this case, the specification becomes

$$\phi_F := \phi_D \wedge \mathbf{G}\,\neg(\mathsf{i}_4 \wedge \mathbf{X}\,\mathsf{i}_2)\,.$$

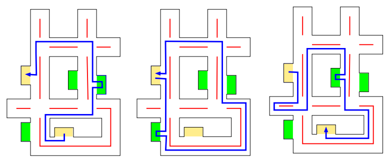

Fig. 12. The robot path for Case E. The robot visits g_1, then g_2, and then g_3, periodically. The value of the optimization function is 9.13 m, which is the same as in Case D.

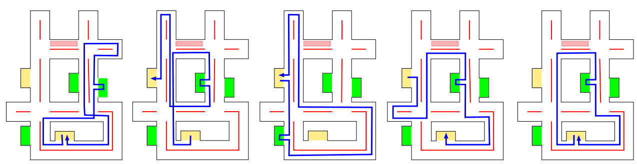

Fig. 13. The robot path for Case F. The robot never uses the road connecting i_4 to i_2. The value of the optimization function is 10.48 m, which is more than in Case D.

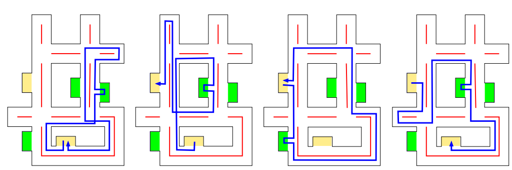

Fig. 14. The robot path for Case G. The robot uploads the data in u_2 after gathering them in g_3. The value of the optimization function is 9.5 m, which again exceeds that of Case D.

The robot path for this case is shown in Figure 13.

Case G. Another type of constraint may be that data from location g_3 must be uploaded at location u_2. The specification from Case D can easily be extended to incorporate this constraint:

$$\phi_G := \phi_D \wedge \mathbf{G}\left(\mathsf{g}_3 \Rightarrow \left(\neg \mathsf{u}_1\, \mathbf{U}\, \mathsf{u}_2\right)\right).$$

The robot path for this case is shown in Figure 14. Note that from Table 1, the cost function value for this case lies between that from Case D and from Case F.

Remark 5.1 (Modeling Robot Navigation Errors)**.** *In implementing the robot paths on our testbed, there were instances in which the robot failed to make the proper transition. This occurred when the robot was following a road, turning at intersections, or entering/exiting data-gather and upload locations. For example, in 50 trials of each motion primitive, we observed three failures when performing left turns, one failure when performing right turns, three failures when entering a gather/upload location, and one failure when exiting a gather/upload location. When such failures occur, the robot enters a different state than expected. Our current method does not allow the robot to recover in these situations.*

Such failures can be modeled and dealt with formally by allowing for non-determinism or probabilistic transitions. For example, if in our experimental setup we observe that applying a right-turn motion primitive at an intersection may result in the robot going straight through it, then we associate both going straight and right-turn outcomes with this motion primitive. The transition system describing the motion of the robot in the environment then becomes non-deterministic. If, in addition, we could quantify the success and failure rates of the motion primitives at different locations in the environment, then we could model the motion of the robot as a Markov decision process (MDP). While there are recent results for temporal-logic control of both of these types of system (Ding et al., 2011; Tůmová et al., 2010; Lahijanian et al., 2011), the connection with optimality is still an open problem and it is a future direction for our research. □

6. Conclusions and future directions

In this paper we presented a method for planning the optimal motion of a robot subject to temporal-logic constraints. Temporal logic provides a rich language in which to describe complex robot missions. Motivated by persistent-monitoring and data-gathering applications, we considered temporal-logic specifications that contain a single *optimizing proposition* that must be repeatedly satisfied. We developed an algorithm for computing the optimal robot path that minimizes the maximum time between satisfying instances of the optimizing proposition. Experimental results show the applicability of this approach for a robot moving in a city-like environment.

There are many promising directions for future work. First, as discussed in Remark 5.1, since robot actions are imprecise, we would like to extend the optimization in this paper to MDPs. This would allow us to model actuator failures, imprecise robot motion, and probabilistic propositions. We are also interested in the case of multiple robots. The difficulty in this problem appears to be capturing the relative positions of robots during their motion. It does not appear that such information can be captured in the transition-system model of this paper. A solution may be to move to timed automata, which are rich enough to capture the full configuration of a group of robots. The apparent drawback of this approach is in the increased computational complexity. Finally, it would be interesting to identify other types of optimization problems that could be solved using this approach. This paper focused on the min-max cost-function formulation since it gives a hard guarantee for the time between satisfying instances. However, there are other relevant costs, such as the average time between satisfying instances. It seems likely that the approach used in this paper could be extended to solve these alternate cost functions, and in our future work we will explore this direction.

Notes

1 If the cycle does not contain an element of S, then its S-bottleneck length is defined as $+\infty$.
2 The notation $|\phi|$ denotes the size of the LTL formula, and is measured in terms of the number of operators (temporal and Boolean) that appear in the formula.

Funding

This material is based upon work supported in part by ONR-MURI (award N00014-09-1-1051), ARO (award W911NF-09-1-0088), and Masaryk University (grant numbers LH11065 and GD102/09/H042), and other funding sources (AFOSR YIP FA9550-09-1-0209, NSF CNS-1035588, NSF CNS-0834260).

Acknowledgement

The work by S. L. Smith was performed while at MIT. We thank Yushan Chen and Samuel Birch at Boston University for their work on the road-network platform and Alphan Ulusoy at Boston University for his work on the implementation.

References

Antoniotti M and Mishra B (1995) Discrete event models + temporal logic = supervisory controller: Automatic synthesis of locomotion controllers. In: *IEEE International Conference on Robotics and Automation*, Nagoya, Japan, pp. 1441–1446.

Baier C, Katoen J-P and Larsen KG (2008) *Principles of Model Checking*. Cambridge, MA: MIT Press.

Barnat J, Brim L and Ročkai P (2009) DiVinE 2.0: High-performance model checking. In: *High Performance Computational Systems Biology*. Los Alamitos, CA: IEEE Computer Society Press, pp. 31–32.

Belta C and Habets LCGJM (2006) Control of a class of nonlinear systems on rectangles. *IEEE Transactions on Automatic Control* 51: 1749–1759.

Belta C, Isler V and Pappas GJ (2005) Discrete abstractions for robot motion planning and control in polygonal environment. *IEEE Transactions on Robotics* 21: 864–875.

Clarke EM, Peled D and Grumberg O (1999) *Model Checking*. Cambridge, MA: MIT Press.

Ding XC, Smith SL, Belta C and Rus D (2011) LTL control with probabilistic satisfaction guarantees. In: *IFAC World Congress*, Milan, Italy, to appear.

Fainekos GE, Girard A, Kress-Gazit H and Pappas GJ (2009) Temporal logic motion planning for dynamic robots. *Automatica* 45: 343–352.

Gastin P and Oddoux D (2001) Fast LTL to Büchi automata translation. In: *Conference on Computer Aided Verification* (*Lecture Notes in Computer Science*, Vol. 2102). Berlin: Springer, pp. 53–65.

Gerth R, Peled D, Vardi M and Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic. In: *Protocol Specification, Testing and Verification*. London: Chapman & Hall, pp. 3–18.

Habets LCGJM and van Schuppen JH (2004) A control problem for affine dynamical systems on a full-dimensional polytope. *Automatica* 40: 21–35.

Habets LCGJM, Collins PJ and van Schuppen JH (2006) Reachability and control synthesis for piecewise-affine hybrid systems on simplices. *IEEE Transactions on Automatic Control* 51: 938–948.

Holzmann G (1997) The model checker SPIN. *IEEE Transactions on Software Engineering* 25: 279–295.

Karaman S and Frazzoli E (2008*a*) Complex mission optimization for multiple-uavs using linear temporal logic. In: *American Control Conference*, Seattle, WA, pp. 2003–2009.

Karaman S and Frazzoli E (2008*b*) Vehicle routing problem with metric temporal logic specifications. In: *IEEE Conference on Decision and Control,* Cancún, México, pp. 3953–3958.

Karaman S, Rasmussen S, Kingston D and Frazzoli E (2009) Specification and planning of uav missions: a process algebra approach. In: *American Control Conference*, St Louis, MO, pp. 1442–1447.

Korte B and Vygen J (2007) *Combinatorial Optimization: Theory and Algorithms*, 4th edn (*Algorithmics and Combinatorics*, Vol. 21). Berlin: Springer.

Kress-Gazit H, Fainekos GE and Pappas GJ (2009) Temporal logic-based reactive mission and motion planning. *IEEE Transactions on Robotics* 25: 1370–1381.

Lahijanian M, Andersson SB and Belta C (2011) Temporal logic control for Markov decision processes. In: *American Control Conference*, San Francisco, CA, to appear.

Laporte G (2009) Fifty years of vehicle routing. *Transportation Science* 43: 408–416.

LaValle SM (2006) *Planning Algorithms*. Cambridge: Cambridge University Press.

LaValle SM and Kuffner JJ (2001) Randomized kinodynamic planning. *International Journal of Robotics Research* 20: 378–400.

Loizou SG and Kyriakopoulos KJ (2004) Automatic synthesis of multiagent motion tasks based on LTL specifications. In: *IEEE Conference on Decision and Control*, Paradise Island, Bahamas, pp. 153–158.

Quottrup MM, Bak T and Izadi-Zamanabadi R (2004) Multi-robot motion planning: A timed automata approach. In: *IEEE International Conference on Robotics and Automation*, New Orleans, LA, pp. 4417–4422.

Rimon E and Koditschek DE (1992) Exact robot navigation using artificial potential functions. *IEEE Transactions on Robotics and Automation* 8: 501–518.

Russell S and Norvig P (2003) *Artificial Intelligence: A Modern Approach*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Smith SL, Tůmová J, Belta C and Rus D (2010) Optimal path planning under temporal logic constraints. In: *IEEE/RSJ International Conference on Intelligent Robots & Systems*, Taipei, Taiwan, pp. 3288–3293.

Tedrake R, Manchester IR, Tobenkin MM and Roberts JW (2010) LQR-trees: Feedback motion planning via sums of squares verification. *International Journal of Robotics Research* 29: 1038–1052.

Toth P and Vigo D, eds (2001) The vehicle routing problem. In: *Monographs on Discrete Mathematics and Applications*. Philadelphia, PA: SIAM.

Tůmová J, Yordanov B, Belta C, Černá I and Barnat J (2010) A symbolic approach to controlling piecewise affine systems. In: *IEEE Conference on Decision and Control*, Atlanta, GA, pp. 4230–4235.

Vardi MY and Wolper P (1986) An automata-theoretic approach to automatic program verification. In: *Logic in Computer Science*, pp. 322–331.

Vardi MY and Wolper P (1994) Reasoning about infinite computations. *Information and Computation* 115: 1–37.

Wongpiromsarn T, Topcu U and Murray RM (2010) Receding horizon control for temporal logic specifications. In: *Hybrid Systems: Computation and Control*, Stockholm, Sweden, pp. 101–110.

Appendix: Index to Multimedia Extensions

The multimedia extension page is found at http://www.ijrr.org

Table of Multimedia Extensions

Extension	Type	Description
1	Video	Robot implementation of data-gathering for case study C