Robotica (2017) volume 35, pp. 26–49. © Cambridge University Press 2015
doi:10.1017/S0263574715000417

Robot motion planning with task specifications via regular languages

James McMahon†‡ and Erion Plaku†*

†*Department of Electrical Engineering and Computer Science, The Catholic University of America, Washington, DC 20064*
‡*U.S. Naval Research Laboratory, Code 7130, Washington, DC 20375*

(Accepted May 13, 2015. First published online: June 17, 2015)

SUMMARY
This paper presents an efficient approach for planning collision-free and dynamically feasible trajectories that enable a mobile robot to carry out tasks specified as regular languages over workspace regions. A sampling-based tree search is conducted over the feasible motions and over an abstraction obtained by combining the automaton representing the regular language with a workspace decomposition. The abstraction is used to partition the motion tree into equivalence classes and estimate the feasibility of reaching accepting automaton states from these equivalence classes. The partition is continually refined to discover new ways to expand the search. Comparisons to related work show significant speedups.

KEYWORDS: Motion planning; Mobile robots; Robot dynamics; Regular languages.

1. Introduction

Addressing the combined task and motion-planning problem is becoming increasingly important as a growing number of diverse robotics applications in navigation, search-and-rescue, manipulation, and surgical procedures involve reasoning with both discrete actions and continuous motions. In this context, regular languages provide a convenient mathematical model to express robotic tasks. For instance, the task of reaching regions $\mathcal{P}_1, \ldots, \mathcal{P}_n$ in succession, referred to as a sequencing task, can be expressed via the regular expression

$$\phi^n_{\text{seq}} = \pi_1 \pi_2 \ldots \pi_n, \tag{1}$$

where π_i denotes the logical proposition "robot reached $\mathcal{P}_i$," which becomes true only when the robot reaches $\mathcal{P}_i$. When giving the robot the flexibility to visit $\mathcal{P}_1, \ldots, \mathcal{P}_n$ in any order, the task is referred to as a coverage task, which can be written as the regular expression

$$\phi^n_{\text{cov}} = \bigvee_{\langle i_1, \ldots, i_n \rangle \in \text{perm}(1,n)} \pi_{i_1} \pi_{i_2} \ldots \pi_{i_n}, \tag{2}$$

where perm$(1, n)$ denotes all the permutations of $\{1, 2, \ldots, n\}$. Regular languages can also be used to impose partial ordering, e.g., "$\mathcal{P}_1$ and $\mathcal{P}_2$ before $\mathcal{P}_3$ and $\mathcal{P}_4$," which corresponds to the regular expression

$$(\pi_1 \pi_2 \vee \pi_2 \pi_1)(\pi_3 \pi_4 \vee \pi_4 \pi_3). \tag{3}$$

More sophisticated tasks can be constructed by combining simpler ones via union, intersection, complementation, reversal, concatenation, homeomorphism, and many other operations under which

* Corresponding author. E-mail: plaku@cua.edu

regular languages are closed. The tasks can be specified as regular expressions, regular grammars, or deterministic finite automata (DFA), as all these models can be used to represent regular languages.

Regular languages also include a special class of Linear Temporal Logic (LTL), which combines propositions with logical (not, or, and) and temporal (always, next, eventually, until) operators. Specifically, LTL formulas that do not use the "always" operator when written in positive normal form are referred to as syntactically co-safe.[51] Using tools from model checking, any syntactically co-safe LTL formula can be converted into a regular language represented by a DFA.[35,36]

Planning motion trajectories that satisfy task specifications given by regular languages pose unique computational challenges stemming from (i) robot dynamics and collision avoidance, (ii) complexity of the task, and (iii) intertwined dependencies between the feasible motions and the task constraints. Robot dynamics express physical constraints on the feasible motions, such as bounding the velocity and directions of motions, enforcing a minimum turning radius or preventing the wheels from sliding sideways.[41,46] These constraints make it difficult to find control inputs that would drive the robot to the goal. Further challenges arise due to the nonholonomicity of mobile robots which have generally fewer local degrees-of-freedom than globally, requiring careful maneuvering to reach desired positions and orientations. The state space where motion planning takes place is also high dimensional as each state includes information about the robot's position, orientation, steering angle, velocity, and other components related to motion. Moreover, the planned motions must not only be dynamically feasible but also collision-free, often requiring the robot to wiggle its way through narrow passages.

When combining motion planning with regular languages additional challenges arise due to the intertwined dependencies between feasible motions and task constraints. Consider a regular language $\mathcal{L}$ and a word $\sigma = \langle \pi_{i_1} \pi_{i_2} \dots \pi_{i_n} \rangle \in \mathcal{L}$. Due to constraints imposed by obstacles and robot dynamics it may be difficult or impossible to reach the regions of interest in the order specified by σ. As the number of words in $\mathcal{L}$ could increase combinatorially fast with respect to the number of propositions, as in Eq. (2), it becomes computationally challenging to determine which $\sigma \in \mathcal{L}$ is feasible. These intertwined dependencies give rise to a chicken-and-egg problem as the feasibility of σ is determined by generating a motion trajectory that follows σ but planning such trajectory requires σ to be feasible. As motion planning with dynamics offers only probabilistic completeness,[11,37] i.e., probability of finding a solution when it exists approaches one as time tends to infinity, it cannot determine the infeasibility of a word. This is akin to Turing machines which will halt if the word is in the language but could run forever otherwise.

To effectively plan collision-free and dynamically feasible motion trajectories that satisfy task specifications given as regular languages, this paper proposes `Argos` (Automaton- and region-guided motion search). In essence, `Argos` couples the ability of sampling-based motion planning to handle the complexity arising from motion dynamics and collision avoidance with the ability of discrete search to handle discrete abstractions. In particular, `Argos` expands a tree of collision-free and dynamically feasible motions by adding new trajectories as tree branches, which are obtained by sampling input controls and propagating forward the motion dynamics of the robot. A discrete abstraction is imposed by implicitly combining the finite automaton representing the regular language with a workspace decomposition. A key aspect of `Argos` is the use of the discrete abstraction to partition the motion tree into equivalence classes based on the progress made toward accepting automaton states. Heuristic costs based on short abstract paths over the automaton and the workspace decomposition are used to evaluate the feasibility of reaching an accepting automaton state by expanding the motion tree from each equivalence class. Heuristic costs are combined with selection penalties in order to avoid overexploration or becoming stuck while attempting to expand the motion tree from an infeasible equivalence class. The initial workspace decomposition and the partition of the motion tree into equivalence classes are continually refined in order to enable `Argos` to discover new ways to expand the search. The performance of `Argos` is tested in simulation using ground and aerial vehicle models with nonlinear dynamics where the robot operates in complex environments and is required to perform various tasks specified as regular languages. Comparisons to related work show significant computational speedups.

Paper organization: Related work and the contribution offered by `Argos` over related work and its preliminary version[40] are discussed in Section 2. Robot models, task specifications as regular languages, and the problem statement are defined in Section 3. `Argos` is described in Section 4. Experiments and results are discussed in Section 5. The paper concludes in Section 6 with a discussion.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

2. Related Work

Motion planning with regular languages is part of an increasing body of work that seeks to combine task and motion planning. Such approaches can be broadly divided into two categories: those that seek to synthesize controllers from task specifications (Section 2.1) and those that use sampling-based motion planning to find a trajectory that satisfies the task specification (Section 2.2). `Argos` belongs to the second category. The contribution of `Argos` over related work, including its preliminary version,[40] is discussed in Section 2.3.

2.1. Controller synthesis

In controller synthesis, the objective is to design a controller that generates robot motions in accordance with the task specification. Frazzoli, Dahleh, and Feron introduce the maneuver automaton which uses regular languages to define rules for concatenating motion primitives.[24] Dantam and Stilman[16] develop a context-free motion grammar in order to represent and verify the discrete dynamics of hybrid systems.

An extensive body of work on controller synthesis has used LTL to specify the robot tasks. Fainekos, Kress-Gazit, Pappas *et al.*,[21] synthesize global controllers for a robot operating in a 2D environment by using model checking to compute a sequence of regions that satisfies the LTL formula and then relying on a local controller to drive the robot from one region to the next as specified in the sequence. Assuming perfect binary sensors for event detection, further work[33] enables the robot to respond to events sensed along the planned path. Controller synthesis has also been used to drive vehicle models in accordance with traffic rules expressed as LTL formulas.[32,34] Vasumathi and Kress-Gazit[47] develop a method based on counterstrategy generation and Boolean satisfiability testing to explain why certain unsynthesizable LTL specification cannot be fulfilled. Filippidis, Dimarogonas, and Kyriakopoulos propose a decentralized approach to control multiple robots from local LTL specifications.[22] Belta, Bicchi, Egerstedt *et al.*,[2] use model checking and motion primitives to synthesize controllers that satisfy LTL specifications. Hierarchical abstractions are introduced in later work[8,31] in order to control a team of robots. Local events are accommodated by a receding horizon technique.[19] Ulusoy, Wongpiromsran, and Belta[53] develop an incremental control-synthesis approach for probabilistic environments with the objective of maximizing the probability of satisfying specifications given by syntactically co-safe LTL formulas.

Controller-synthesis approaches, under some assumptions, are generally correct-by-construction in the sense that the synthesized controller is guaranteed to satisfy the task specification given by the LTL formula. In particular, the environment must be admissible so that bisimulation holds, which requires the availability of a local controller that can guarantee collision-free and dynamically feasible trajectories between any two states from neighboring regions. Designing such controllers, especially for high-dimensional systems with nonlinear dynamics, is a challenging problem and subject of extensive research.[1,10,17,23,27,30]

2.2. Sampling-based motion planning

Sampling-based motion planning leverages the idea of selectively sampling and searching the state space in order to generate a collision-free and dynamically feasible trajectory to a goal region. Sampling-based motion planners can take nonlinear dynamics into account by using a motion tree to conduct the search.[11,37] Starting with the initial state as the root, the motion tree is incrementally expanded by adding new trajectories as branches. Each trajectory is obtained by selecting a vertex from the motion tree, sampling control inputs, and simulating the motion resulting from applying the control inputs for several time steps. Numerous algorithms have been proposed, e.g., `RRT`,[38,39] `EST`,[26] which use nearest neighbors, probability distributions, decompositions, and many other functions to guide the motion-tree expansion.[14,15,18,25,42,45,48,49]

Another class of sampling-based approaches follows a model predictive control paradigm.[7] As in sampling-based motion planning with dynamics, approaches based on model predictive control use sampling of input controls and a model of the robot dynamics to generate dynamically feasible motions. Typically, more than one neighbor is generated each time a vertex is selected for expansion. The vertex selection and expansion strategies often seek to promote expansions toward the goal or optimize some cost function.[12,20] These approaches have focused on computing a feasible trajectory to a goal region. They have not been extended to accomodate task specifications given as regular languages.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

Sampling-based motion planning has also focused on reachability where the objective is to plan a feasible trajectory to a goal region. To account for LTL specifications, `LTLSyclop` coupled sampling-based motion planning with discrete search.[43–45] `LTLSyclop` was shown to effectively guide the motion-tree expansion and handle task specifications given by syntactically co-safe LTL, which can be modeled by DFAs. `LTLSyclop` was enhanced further by starting the discrete search from recently explored abstract states instead of the initial abstract state.[4–6]

Another class of sampling-based motion planners follows a "bottom-up" approach which uses the automaton as an external monitor to keep track of the automaton states associated with the tree trajectories. In this setting, Karaman and Frazzoli[29] use a variant of `RRT` to plan trajectories that satisfy tasks specifications given by deterministic μ-calculus, which includes LTL. This work also shows that optimality can be obtained by rewiring the tree branches in a similar fashion as in `RRT*`.[28] Rewiring, however, similar to controller-synthesis approaches, requires the availability of a controller that can guarantee exact and optimal steering between any two states. Designing such controllers especially for high-dimensional systems with nonlinear dynamics remains challenging.[9,30] Vasile and Belta,[54] under similar assumptions, create a sparse graph with cycles, as a variant of `RRT`, in order to find infinite paths that satisfy a given LTL formula. They also extend this work to combine long-term LTL goals with short-term reactive requirements.[55]

2.3. Contribution over related work

The contribution of `Argos` has several key aspects: (i) partition of the sampling-based motion tree into equivalence classes; (ii) heuristic costs based on short abstract paths to estimate the feasibility of reaching an accepting automaton state from each equivalence class; and (iii) partition refinement in order to promote rapid expansions while discovering new ways to reach accepting automaton states. These make it possible to efficiently plan collision-free and dynamically feasible trajectories that satisfy task specifications given as regular languages.

In contrast, bottom-up approaches, which are based on `RRT` variants and use the automaton as an external monitor, often lack guidance toward accepting automaton states. As a result, they end up wasting significant computational time exploring parts of the space that do not advance the search.

`LTLSyclop` approaches rely on the discrete search to guide the motion-tree expansion. Even though `LTLSyclop` approaches have been shown to solve challenging problems, they still suffer from scalability issues. In fact, `LTLSyclop`, at each iteration, attempts to follow an entire abstract path to an accepting automaton state. As such, `LTLSyclop` often wastes considerable computational time before realizing that, due to constraints imposed by dynamics and obstacles, the current abstract path needs to be abandoned and a new one needs to be computed. As the size of the automaton increases, it becomes increasingly difficult for `LTLSyclop` to find feasible abstract paths that can effectively guide the motion-tree expansion.

A preliminary version of `Argos` appeared as a conference proceeding.[40] `Argos` offers several algorithmic improvements over its preliminary version and extended experimental evaluation with ground and aerial vehicle models. In particular, `Argos` introduces partition refinement which not only reduces its dependence on the initial workspace decomposition but also considerably improves its performance by more effectively identifying equivalence classes which can advance the search toward accepting automaton states. The preliminary version precomputed heuristic costs for every possible abstract state. As such, it could not be scaled to large automata. In contrast, `Argos` computes the heuristic cost of an abstract state only when it is reached by the motion tree. Moreover, `Argos` improves the discrete search by using internal heuristics based on distances to accepting automaton states and leveraging shortest paths in the decomposition to the regions of interest. The experiments in `Argos` include subdivision decompositions and aerial vehicle models in addition to the triangulation decompositions and ground vehicle models used in the preliminary version.

3. Mathematical Framework

This section defines the robot models and motion trajectories from a motion-planning perspective, the task specifications as regular languages over workspace propositions, the automata representations, the semantics of task specifications over motion trajectories, and the problem statement.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

3.1. Robot models and motion trajectories

From a motion-planning perspective, a robot model is defined by its state and motions resulting from applying controls. A robot state $s \in \mathcal{S}$, where $\mathcal{S}$ denotes the state space, defines the position, orientation, steering angle, linear and angular velocities, and other components that change as a result of robot motion. A state $s \in \mathcal{S}$ is considered valid, as computed by a function $\text{VALID} : \mathcal{S} \to \{\top, \bot\}$, when position, steering angle, velocity, and other components of s are within desired bounds, and the robot is not in collision with obstacles when placed according to the position and orientation specified in s.

The robot is controlled by applying external inputs. As an example, a robotic vehicle is often controlled by setting the acceleration and steering wheel. A control function $u : [0, T] \to \mathcal{U}$ indicates the control input that is applied to the robot at each time step $t \in [0, T]$, where $\mathcal{U}$ is the control space. As a result of applying the control inputs, the robot state changes according to its underlying motion dynamics, giving rise to a motion trajectory $\zeta : [0, T] \to \mathcal{S}$.

Motion dynamics are often specified as a set of differential equations $f_{\text{MotionEqs}} : \mathcal{S} \times \mathcal{U} \to \dot{\mathcal{S}}$. Since $\mathcal{S}$ often includes velocities, $f_{\text{MotionEqs}}$ is generally nonlinear. Moreover, this work allows for nonholonomic constraints, which are essential to model motion dynamics associated with robotic vehicles and other systems, since the controllable degrees of freedom are less than the total degrees of freedom. In addition to differential equations, physics-based engines such as Bullet[13] and ODE[52] are used to provide an increased level of realism by also modeling friction, gravity, nonflat terrains, and other interactions of the robot with the world, which cannot be easily described analytically. Examples of a physics-based ground vehicle, a snake-like robot, and an aerial vehicle model are provided in Section 5.1.

From a motion-planning perspective, the motions resulting from applying input controls are encapsulated by a function

$$s_{\text{new}} \leftarrow \text{MOTION}(s, u, dt), \tag{4}$$

where the new state s_{new} is obtained by applying the control u to the state s for one time step dt. When the motion dynamics are described as a set of differential equations $f_{\text{MotionEqs}}$, the simulator relies on numerical integration to compute s_{new}. To ensure accuracy, as advocated in the literature, Runge–Kutta methods with an adaptive step are used for the integration. For increased realism, the proposed planner, `Argos`, can also work in conjunction with physics-based engines which model general rigid body dynamics by computing the forces acting on the bodies and the motions resulting from applying these forces.

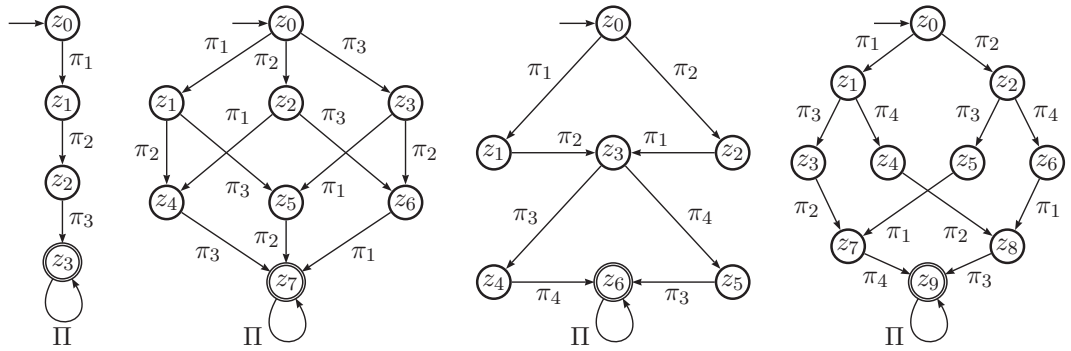
3.2. Task specifications as regular languages over workspace propositions

The world in which the robot operates, referred to as the workspace and denoted by $\mathcal{W}$, contains several obstacles $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_m\}$ and several regions of interest $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, which do not intersect with each other or any of the obstacles. Each $\mathcal{P}_i \in \mathcal{P}$ is labeled with a proposition π_i. The set of propositions is then defined as $\Pi = \{\pi_1, \ldots, \pi_n\}$. A function $\text{PROP} : \mathcal{W} \to \Pi \cup \{\pi_\bot\}$ maps each point $p \in \mathcal{W}$ to the corresponding proposition in Π if p is inside some $\mathcal{P}_i \in \mathcal{P}$ or to the special symbol $\pi_\bot$ if p is not inside any of the regions of interest, i.e.,

$$\text{PROP}(p) = \begin{cases} \pi_i, & \text{if } p \in \mathcal{P}_i \text{ for some } \mathcal{P}_i \in \mathcal{P}, \\ \pi_\bot, & \text{if } p \notin \cup_{i=1}^{n} \mathcal{P}_i. \end{cases} \tag{5}$$

The task that the robot is required to accomplish is specified as a regular language over Π. As an example, the task of visiting $\mathcal{P}_1$ or $\mathcal{P}_2$ before $\mathcal{P}_3$ can be expressed as the regular expression $(\pi_1 \vee \pi_2)\pi_3$. Equation (1) provides an example of a sequencing task which requires the robot to visit $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ in succession. Equation (2) provides an example of a coverage task where the robot can visit $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ in any order.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press



(a) ϕ^3_{seq} (b) ϕ^3_{cov} (c) ϕ^4_{po} (d) $\phi^4_{\text{zig-zag}}$

Fig. 1. Examples of DFAs for instances of the tasks described in Section 3.2. To simplify the figures, if $\delta_{\mathcal{A}}(z, \pi_i)$ is not shown, it means that the DFA enters the reject state z_{rej} and stays in z_{rej} for all the inputs.

As another example, in the partial-order task, the robot is required to first visit $\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil \frac{n}{2} \rceil}$ in any order and then visit $\mathcal{P}_{\lceil \frac{n}{2} \rceil + 1}, \ldots, \mathcal{P}_n$ in any order, i.e.,

$$\phi^n_{\text{po}} = \left(\bigvee_{\substack{\langle i_1, \ldots, i_{\lceil \frac{n}{2} \rceil} \rangle \in \\ \text{perm}(1, \lceil \frac{n}{2} \rceil)}} \pi_{i_1} \ldots \pi_{i_{\lceil \frac{n}{2} \rceil}} \right) \left(\bigvee_{\substack{\langle i_{1+\lceil \frac{n}{2} \rceil}, \ldots, i_n \rangle \in \\ \text{perm}(1+\lceil \frac{n}{2} \rceil, n)}} \pi_{i_{1+\lceil \frac{n}{2} \rceil}} \ldots \pi_n \right). \tag{6}$$

Experiments were also conducted with a zig-zag task which partitions the regions of interest into two sets: $A = \{\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil \frac{n}{2} \rceil}\}$ and $B = \mathcal{P} \setminus A$. The robot is then required to visit regions by alternating between A and B, i.e., A, B, A, B, ..., until it visits all the regions in $\mathcal{P}$, ensuring that it never visits the same region twice. The zig-zag task can be expressed as

$$\phi^n_{\text{zig-zag}} = \bigvee_{\substack{\langle i_1, i_3, \ldots, i_{2\lceil \frac{n}{2} \rceil - 1} \rangle \in \\ \text{perm}(1, \lceil \frac{n}{2} \rceil)}} \left(\bigvee_{\substack{\langle i_2, i_4, \ldots, i_{2\lfloor \frac{n}{2} \rfloor} \rangle \in \\ \text{perm}(1+\lceil \frac{n}{2} \rceil, n)}} \pi_{i_1} \pi_{i_2} \ldots \pi_{i_n} \right). \tag{7}$$

3.3. DFA representation and semantics over motion trajectories

The proposed planner, `Argos`, takes as input a DFA corresponding to the regular language describing the desired task. Examples of DFAs for the instances of the tasks described in Section 3.2 are shown in Fig. 1. A DFA over propositions $\Pi = \{\pi_1, \ldots, \pi_n\}$ is defined formally as follows.

Definition 1. A DFA is a tuple $\mathcal{A} = (Z, \Pi, \delta, z_{\text{init}}, \text{Accept})$, where Z is a finite set of states, Π is the input alphabet, $\delta : Z \times \Pi \to Z$ is the transition function, $z_{\text{init}} \in Z$ is the initial state, and Accept $\subseteq Z$ is the set of accepting states. Let $\sqcup$ denote the empty string. Let $\Pi^* = \cup_{k=0}^{\infty} \Pi^k$, where Π^k denotes the strings of length k formed by concatenating the symbols in Π (with $\Pi^0 = \{\sqcup\}$). The extended transition function $\hat{\delta} : Z \times \Pi^* \to Z$ is then defined in the usual way, i.e.,

$$\forall z \in Z, \sigma \in \Pi^*, \pi_i \in \Pi : \quad \hat{\delta}(z, \sqcup) = z \quad \text{and} \quad \hat{\delta}(z, \sigma \pi_i) = \delta(\hat{\delta}(z, \sigma), \pi_i). \tag{8}$$

$\mathcal{A}$ accepts $\sigma \in \Pi^*$ if and only if $\hat{\delta}(z_{\text{init}}, \sigma) \in \text{Accept}$.

The robot is considered to have reached a region of interest $\mathcal{P}_i$ if the position component of its state is inside $\mathcal{P}_i$. More formally, let PROJ : $\mathcal{S} \to \mathcal{W}$ provide a mapping from states to workspace points. Given that the robot state s often includes $s = \langle s_{\text{position}}, s_{\text{rotation}}, s_{\text{velocity}}, \ldots \rangle$, the projection function

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

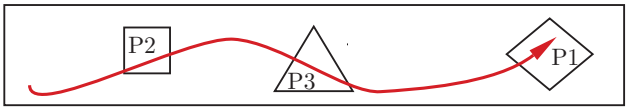


Fig. 2. The sequence of regions reached by ζ is $\mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_1$. Hence, $\text{WORD}(\zeta) = \langle \pi_2 \pi_3 \pi_1 \rangle$.

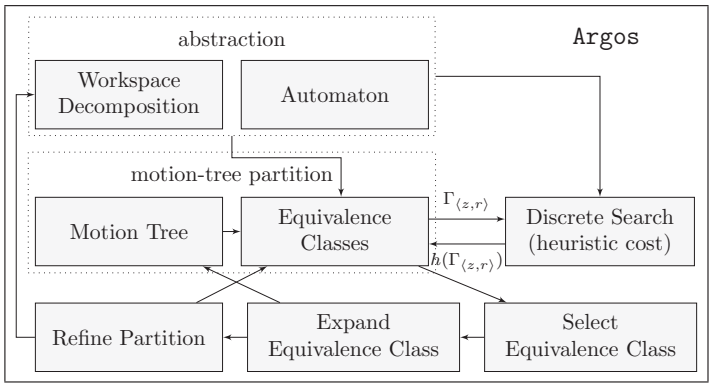


Fig. 3. Schematic representation of the interactions among the main components of `Argos`.

can be defined as $\text{PROJ}(s) = s_{\text{position}}$. The proposition satisfied by the robot when at state s is then given as $\text{PROP}(\text{PROJ}(s))$. For convenience, the notation $\text{PROP}(s)$ is used as shorthand for $\text{PROP}(\text{PROJ}(s))$.

A trajectory $\zeta : [0, T] \to \mathcal{S}$ is considered to have reached $\mathcal{P}_i$ at time $t \in [0, T]$ if the robot's position is inside $\mathcal{P}_i$, i.e., $\text{PROJ}(\zeta(t)) \in \mathcal{P}_i$. Let $\mathcal{P}_{i_1}$ denote the first region reached by ζ. Let $\mathcal{P}_{i_2}$ denote the next region reached by ζ, where $\mathcal{P}_{i_1} \neq \mathcal{P}_{i_2}$. Continuing in this manner, ζ will reach a sequence of regions $\mathcal{P}_{i_1}, \mathcal{P}_{i_2}, \ldots, \mathcal{P}_{i_k}$, where $\mathcal{P}_{i_j} \neq \mathcal{P}_{i_{j+1}}$ for all $1 \leq j < k$. The sequence of propositions $\langle \pi_{i_1} \pi_{i_2} \ldots \pi_{i_\ell} \rangle \in \Pi^*$ satisfied by ζ, where π_{i_j} is the proposition associated with $\mathcal{P}_{i_j}$, is referred to as the word generated by ζ and is denoted by $\text{WORD}(\zeta)$. Figure 2 shows an example.

As a result of this mapping, ζ satisfies the given task if and only if an accepting automaton state is reached when running $\mathcal{A}$ with $\text{WORD}(\zeta)$ as the input, i.e.,

$$\hat{\delta}_{\mathcal{A}}(z_{\text{init}}, \text{WORD}(\zeta)) \in \text{Accept}_{\mathcal{A}}. \tag{9}$$

3.4. Problem statement

The motion-planning problem with task-specifications via regular languages over workspace propositions can be stated as follows.

Definition 2. Given

- the workspace $\mathcal{W}$ with obstacles $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_m\}$ and regions of interest $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$,
- a task specified as a DFA $\mathcal{A} = \langle Z, \Pi, z_{\text{init}}, \delta, \text{Accept} \rangle$ over propositions $\Pi = \{\pi_1, \ldots, \pi_n\}$, where π_i is associated with the region of interest $\mathcal{P}_i$,
- a robot model $\langle \mathcal{S}, \mathcal{U}, \text{MOTION} \rangle$ to simulate its motion dynamics,
- a function $\text{VALID} : \mathcal{S} \to \{\top, \bot\}$ where $\text{VALID}(s)$ checks whether or not the robot is in collision and state values are within desired bounds, and
- an initial state $s_{\text{init}} \in \mathcal{S}$

compute a control function $\hat{u} : [0, T] \to \mathcal{U}$ such that the trajectory $\zeta : [0, T] \to \mathcal{S}$ obtained by applying $\hat{u}$ starting at s_{init} avoids collisions, i.e., $\forall t \in [0, T] : \text{VALID}(\zeta(t)) = \top$, and satisfies the task, i.e., $\hat{\delta}_{\mathcal{A}}(z_{\text{init}}, \text{WORD}(\zeta)) \in \text{Accept}_{\mathcal{A}}$.

4. Method

Before presenting the algorithmic details, this section provides an overview of `Argos` focusing on its main components and their interplay. A schematic illustration is provided in Fig. 3.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

`Argos` conducts the search by expanding a tree of collision-free and dynamically feasible motion trajectories. `Argos` uses a discrete abstraction based on an implicit combination of the automaton representing the regular language with a workspace decomposition to partition the motion tree into equivalence classes. In particular, each vertex v in the motion tree is associated with an automaton state depending on the sequence of regions of interest reached by the trajectory connecting the root of the tree to v. Each vertex v is also associated with the region in the workspace that contains it. Vertices that are associated with the same automaton state and the same decomposition region are said to belong to the same equivalence class.

`Argos` proceeds iteratively by (i) selecting an equivalence class, (ii) expanding the motion tree from vertices associated with the selected equivalence class, and (iii) refining the partition. The selection of an equivalence class relies on heuristic costs which estimate the feasibility of reaching an accepting automaton state. Such heuristic costs are computed based on short abstract paths over the discrete abstraction composed of the automaton and the workspace decomposition. Once selected, the equivalence class is expanded by adding a collision-free and dynamically feasible trajectory from one of its vertices. The objective is to expand the motion tree along regions associated with the abstract path to an accepting automaton state. As a result of the expansion, new vertices are added to the motion tree. These vertices are associated with existing equivalence classes or new equivalence classes are created when they reach new automaton states and decomposition regions. After the expansion, a penalty is applied to the selected equivalence class in order to avoid overexploration or becoming stuck when expansion attempts fail to make progress. Moreover, the partition is further refined in order to enable `Argos` to discover new ways to reach an accepting automaton state. `Argos` continues in this manner by selecting an equivalence class, expanding it, and refining the partition until it finds a solution or exceeds the maximum allowed running time.

The rest of the section is organized as follows. The workspace decomposition, partition of the motion-tree into equivalence classes, and the computation of the heuristic costs are described in Sections 4.1–4.3. The overall search is described in Section 4.4.

4.1. Workspace decomposition

The workspace $\mathcal{W}$ is decomposed into a number of nonoverlapping (except at the boundary) regions. The physical adjacency among the regions in the decomposition is expressed as a graph $D = (R, E)$, where R denotes the regions and $E = \{(r_i, r_j) : r_i, r_j \in R \text{ and } r_i, r_j \text{ are adjacent}\}$ denotes the edges. The cost of an edge $(r_i, r_j) \in E$ is defined as the Euclidean distance between the centroids of r_i and r_j, i.e., $\text{cost}(r_i, r_j) = ||\text{centroid}(r_i), \text{centroid}(r_j)||$.

`Argos` can work with any workspace decomposition. This work uses both triangulations and subdivisions. Triangulations are applicable only to 2D workspaces, while subdivisions can be used for both 2D and 3D workspaces. In a triangulation, the unoccupied workspace area, $(\mathcal{W} \setminus \mathcal{O}) \setminus \cup_{i=1}^{n} \mathcal{P}_i$, is decomposed into a number of triangles $\text{tri}_1, \ldots, \text{tri}_\ell$. The set of regions R includes these triangles as well as the polygons $P_1, \ldots, P_n$ associated with the propositions of interest $\pi_1, \ldots, \pi_n$, i.e., $R = \{\text{tri}_1, \ldots, \text{tri}_\ell, \mathcal{P}_1, \ldots, \mathcal{P}_n\}$. Regions $r_1, r_2 \in R$ are considered adjacent if and only if they share a side. The Triangle package[50] is used for the computation of the triangulation. Examples are shown in Figs. 4 and 5.

In a subdivision decomposition, the bounding box of $\mathcal{W}$ is recursively split into half along the largest dimension until it is free of obstacles, it is entirely occupied by obstacles, or its volume becomes smaller than a predefined threshold. Each leaf cell in the subdivision tree is labeled as "free," "mixed," or "blocked" depending on whether it is free of obstacles, partially occupied by obstacles or entirely occupied by obstacles. The set of regions R includes all the leaf cells marked as "free." It also includes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ since they do not necessarily correspond to subdivision cells (each $\mathcal{P}_i$ can be any polygon in 2D and any polyhedron in 3D). A subdivision cell is considered to be adjacent to $\mathcal{P}_i$ if they intersect or one is inside the other. Two subdivision cells are considered to be adjacent if they share a boundary (part of an edge in 2D or part of a face in 3D). Figure 5 shows a subdivision of $\mathcal{W}$.

`Argos` also relies on a function LocateRegion : $\mathcal{W} \to R \cup \{r_\perp\}$, which maps each workspace point p to the corresponding region in R or to the special symbol $r_\perp$ if p is not inside any of the regions. In the case of subdivision decomposition, a point p could belong both to some $\mathcal{P}_i$ and some subdivision cell. For this reason, LocateRegion determines first if p is inside any of the regions $\mathcal{P}_1, \ldots, \mathcal{P}_n$. If not, it searches among the subdivision cells. LocateRegion can run in

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

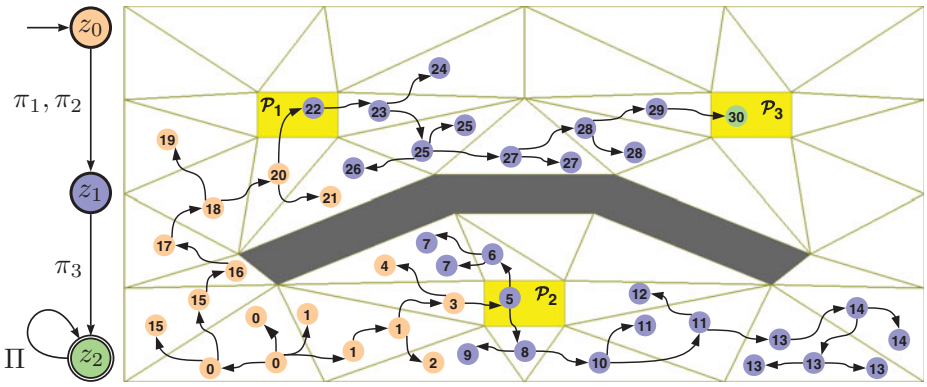


Fig. 4. Partition of the motion tree into equivalence classes as induced by the automaton and the workspace decomposition. Task specified by the automaton requires visiting $\mathcal{P}_1$ or $\mathcal{P}_2$ and then $\mathcal{P}_3$, i.e., $(\pi_1 \vee \pi_2)\pi_3$. Tree vertices are colored according to the corresponding automaton state. Tree vertices belonging to the same equivalence class, i.e., the same automaton state and the same decomposition region, are labeled with the same number.

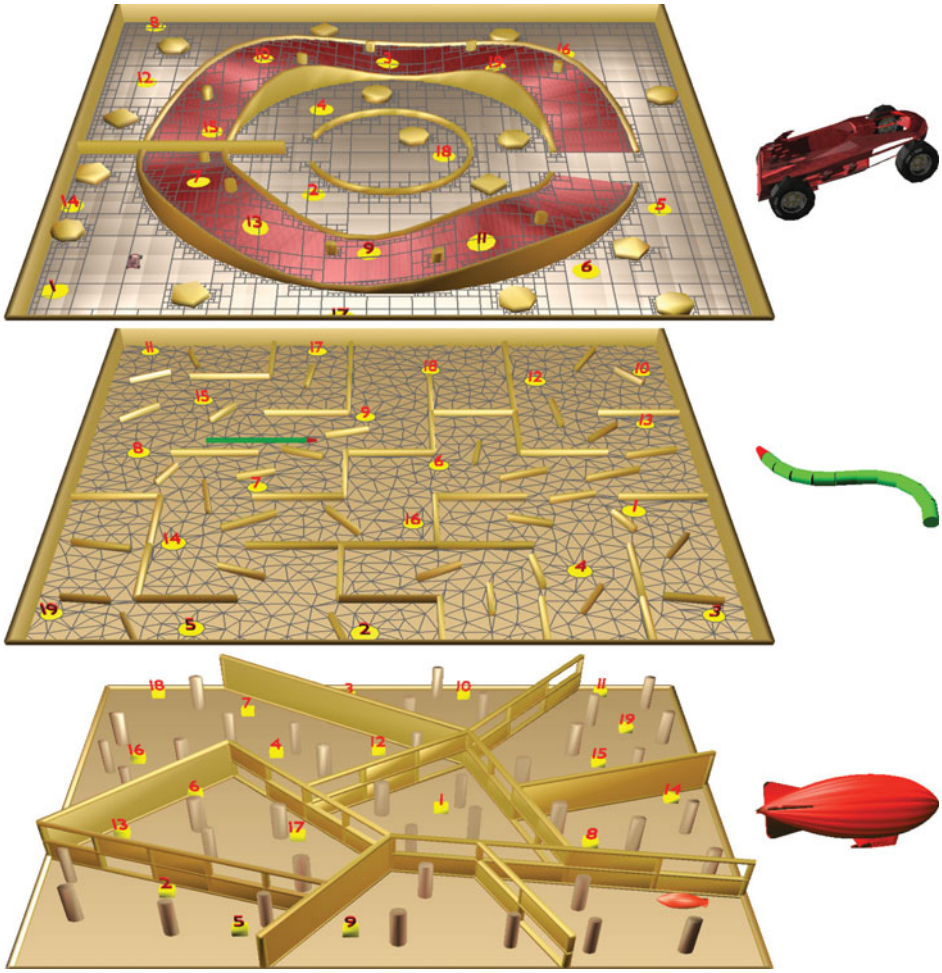


Fig. 5. Scenes and robots. Each figure shows an example of a problem instance with 19 regions of interest. A subdivision workspace decomposition is shown for the first scene and a triangulation decomposition is shown for the second scene (scene 3 uses a subdivision decomposition but it is not shown here as it clutters the figure). Figures viewed better in color and on screen. Video attachment shows solutions obtained by `Argos`.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

polylogarithmic time with respect to the number of triangles[3] or in logarithmic time in the case of subdivision decompositions. To facilitate presentation, the shorthand notation LOCATEREGION(s) is used to denote running the function with the position component of the state s as the input point, i.e., LOCATEREGION(s) $\stackrel{\text{def}}{=}$ LOCATEREGION(PROJ(s)).

4.2. Partition of the motion tree into equivalence classes

The search for a collision-free and dynamically feasible trajectory that satisfies the task specification is conducted by expanding a motion tree $\mathcal{T}$ in the state space $\mathcal{S}$. The motion tree $\mathcal{T}$ is maintained as a directed acyclic graph consisting of vertices and edges. Each vertex $v \in \mathcal{T}$ is associated with a collision-free state in $\mathcal{S}$, denoted by $v.s$. Each edge $(v, v') \in \mathcal{T}$ is associated with a collision-free and dynamically feasible trajectory from $v.s$ to $v'.s$. Starting from the initial state s_{init} as the root, $\mathcal{T}$ is expanded by adding new vertices and edges. The details of the motion-tree expansion are provided later in Section 4.4.2. For now, consider a vertex $v \in \mathcal{T}$. Let TRAJ$_{\mathcal{T}}(v)$ denote the trajectory from s_{init} to v, which is obtained by concatenating the trajectories associated with the edges in $\mathcal{T}$ that connect the root to v. A solution is obtained when a vertex v is added to $\mathcal{T}$ such that WORD(TRAJ$_{\mathcal{T}}(v)$) is accepted by the automaton $\mathcal{A}$, i.e.,

$$\hat{\delta}_{\mathcal{A}}(z_{\text{init}}, \text{WORD}(\text{TRAJ}_{\mathcal{T}}(v))) \in \text{Accept}_{\mathcal{A}}. \tag{10}$$

We make an important observation that the automaton and the workspace decomposition induce a partition of the vertices in $\mathcal{T}$ into equivalence classes. Informally, vertices v_i and v_j belong to the same equivalence class if and only if TRAJ$_{\mathcal{T}}(v_i)$ and TRAJ$_{\mathcal{T}}(v_j)$ end up in the same decomposition region and their corresponding words end up on the same automaton state. That is, from a discrete perspective, these trajectories provide the same information. More formally, $v.s$ is defined as the last state of TRAJ$_{\mathcal{T}}(v)$, i.e.,

$$v.s = \text{LASTSTATE}(\text{TRAJ}_{\mathcal{T}}(v)). \tag{11}$$

Let $v.r$ denote the decomposition region in R associated with v, i.e.,

$$v.r = \text{LOCATEREGION}(v.s). \tag{12}$$

Let $v.z$ denote the automaton state obtained by running WORD(TRAJ$_{\mathcal{T}}(v)$) on the automaton $\mathcal{A}$, i.e.,

$$v.z = \hat{\delta}_{\mathcal{A}}(z_{\text{init}}, \text{WORD}(\text{TRAJ}_{\mathcal{T}}(v))). \tag{13}$$

Consider an abstract state $\langle z, r \rangle$ with $z \in Z_{\mathcal{A}}$ and $r \in R$. Then, the equivalence class $\Gamma_{\langle z,r \rangle}$ groups together all the tree vertices mapped to $\langle z, r \rangle$, i.e.,

$$\Gamma_{\langle z,r \rangle} = \{v : v \in \mathcal{T} \wedge v.z = z \wedge v.r = r\}. \tag{14}$$

Let Γ denote the partition of the motion tree $\mathcal{T}$ into these equivalence classes, i.e.,

$$\Gamma = \{\Gamma_{\langle z,r \rangle} : |\Gamma_{\langle z,r \rangle}| > 0\}. \tag{15}$$

An illustration is provided in Fig. 4. The partition, as described next, is used to effectively guide the expansion of the motion tree $\mathcal{T}$.

4.3. Heuristic costs over the discrete abstraction

Consider an equivalence class $\Gamma_{\langle z,r \rangle}$. Let $\sigma = \langle \pi_{i_1} \dots \pi_{i_k} \rangle \in \Pi^*$ denote a word that leads to an accepting automaton state when starting from z, i.e., $\hat{\delta}_{\mathcal{A}}(z, \sigma) \in \text{Accept}_{\mathcal{A}}$. Let Λ denote a sequence of regions in the decomposition $D = (R, E)$ that starts at r and reaches $\mathcal{P}_{i_1}, \dots, \mathcal{P}_{i_k}$ in succession. If the motion tree $\mathcal{T}$ can be expanded from $\Gamma_{\langle z,r \rangle}$ so that it follows Λ, then the resulting trajectory will satisfy the task specification. Of course, constraints imposed by dynamics and obstacles may make it difficult or even impossible to move along certain regions of Λ. Nevertheless, Λ can serve as a heuristic to guide the motion-tree expansion.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

For this purpose, `Argos` searches over both $\mathcal{A}$ and $D = (R, E)$ to compute a discrete path from the abstract state $\langle z, r\rangle$ to an accepting automaton state. The length of this path serves as the heuristic cost for $\Gamma_{\langle z,r\rangle}$. During the discrete search, the outgoing edges of an abstract state $\langle z', r'\rangle$ with $z' \in Z_{\mathcal{A}}$ and $r' \in R$ (recall that R includes $\mathcal{P}_1, \ldots, \mathcal{P}_n$) are computed on-the-fly as

$$\text{EDGES}(\langle z', r'\rangle) = \{\langle z'', \mathcal{P}_i\rangle : \pi_i \in \Pi \wedge z'' = \delta_{\mathcal{A}}(z', \pi_i)\}. \tag{16}$$

In other words, there is an edge from $\langle z', r'\rangle$ to $\langle z'', \mathcal{P}_i\rangle$ if the automaton $\mathcal{A}$ transitions from z' to z'' with input π_i. In this way, reaching $\mathcal{P}_i$ from r' would enable a transition from z' to z''. The cost of an abstract edge $(\langle z', r'\rangle, \langle z'', \mathcal{P}_i\rangle)$ is defined as the length of the shortest path in $D = (R, E)$ from r' to $\mathcal{P}_i$, i.e.,

$$\text{cost}(\langle z', r'\rangle, \langle z'', \mathcal{P}_i\rangle) = \text{MINCOSTPATH}_D(r', \mathcal{P}_i). \tag{17}$$

Let $\langle z, r\rangle, \langle z_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \langle z_{i_k}, \mathcal{P}_{i_k}\rangle$ denote the abstract path computed by the discrete search. Then, the heuristic cost $h(\Gamma_{\langle z,r\rangle})$ of $\Gamma_{\langle z,r\rangle}$ is defined as

$$h(\Gamma_{\langle z,r\rangle}) = \text{MINCOSTPATH}_D(r, \mathcal{P}_{i_1}) + \sum_{j=1}^{k-1} \text{MINCOSTPATH}_D(\mathcal{P}_{i_j}, \mathcal{P}_{i_{j+1}}). \tag{18}$$

`Argos` uses best-first search to conduct the discrete search. To make the search efficient, `Argos` precomputes for each $\mathcal{P}_i$ the shortest path to each region $r' \in R$. More specifically, the shortest paths in $D = (R, E)$ from $\mathcal{P}_i$ to each $r' \in R$ are computed with a single call to Dijkstra's single-source shortest-path algorithm using $\mathcal{P}_i$ as the source. As a result of this precomputation, each call to $\text{MINCOSTPATH}_D(r', \mathcal{P}_i)$ is carried out in constant time by returning the precomputed result.

As the size of the automaton $\mathcal{A}$ can grow exponentially with respect to the number of propositions, the best-first search uses an evaluation function $f_{\text{BestFirst}}$ in order to effectively guide the search toward an accepting automaton state. For an abstract state $\langle z, r\rangle$, it is defined as

$$f_{\text{BestFirst}}(\langle z, r\rangle) = d_{\mathcal{A}}(z) \cdot d(\mathcal{P}_1, \ldots, \mathcal{P}_n). \tag{19}$$

The term $d_{\mathcal{A}}(z)$ denotes the minimum number of transitions in $\mathcal{A}$ from z to an accepting state. These values are precomputed for each $z \in Z_{\mathcal{A}}$ by a single call to Dijkstra's shortest-path algorithm using a special state connected to all the accepting automaton states as the source and reversing the edges of $\mathcal{A}$. The term $d(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ denotes the minimum pairwise shortest-path distance among $\mathcal{P}_1, \ldots, \mathcal{P}_n$ in $D = (R, E)$, i.e.,

$$d(\mathcal{P}_1, \ldots, \mathcal{P}_n) = \min_{\substack{1 \le i,j \le n \\ i \ne j}} \text{MINCOSTPATH}_D(\mathcal{P}_i, \mathcal{P}_j). \tag{20}$$

The best-first search is implemented using a minimum heap data structure. As is common, each node keeps track of the best cost found so far to reach it and the parent it came from. The search terminates when a node labeled with $\langle z', r'\rangle$ where $z' \in \text{Accept}_{\mathcal{A}}$ is added to the heap. In the heap comparison, when two abstract states $\langle z', r'\rangle$ and $\langle z'', r''\rangle$ have the same value according to $f_{\text{BestFirst}}$, the tie is broken by comparing the best costs found so far to reach $\langle z', r'\rangle$ and $\langle z'', r''\rangle$. As shown by the experimental results, the discrete search efficiently computes short abstract paths that effectively guide the motion-tree expansion.

4.4. Overall automaton- and region-guided expansion of the motion tree

Pseudocode for `Argos` is given in Algorithm 1. `Argos` starts by computing a workspace decomposition (Algorithm 1:1) as described in Section 4.1. `Argos` then invokes Dijkstra's algorithm for each $\mathcal{P}_i \in \mathcal{P}$ as the source to compute the shortest paths from $\mathcal{P}_i$ to each region $r \in R$ (Algorithm 1:2). The results are stored with each $r \in R$ so that $\text{MINCOSTPATH}_D(r, \mathcal{P}_i)$ can return the precomputed result in constant time when invoked by `Argos` during the computation of the heuristic costs associated with the equivalence classes in Γ. To effectively guide the discrete search over $\mathcal{A}$ and

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

$D = (R, E)$, as discussed in Section 4.3, `Argos` also precomputes for each automaton state $z \in Z_{\mathcal{A}}$ the minimum number of transitions to reach an accepting state (Algorithm 1:3).

`Argos`$(\mathcal{W}, \mathcal{O}, \mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{U}, \text{MOTION}, \text{VALID}, s_{\text{init}})$
1: $D = (R, E) \leftarrow$ WORKSPACEDECOMPOSITION$(\mathcal{W}, \mathcal{O}, \mathcal{P})$
2: **for** $\mathcal{P}_i \in \mathcal{P}$ **do** SINGLESOURCEALLSHORTESTPATHS$(D, \mathcal{P}_i)$
3: $\langle d_{\mathcal{A}}(z_1), \ldots, d_{\mathcal{A}}(z_k)\rangle \leftarrow$ MINNRTRANSITIONSTOACCEPT$(\mathcal{A})$
4: $\mathcal{T} \leftarrow \emptyset$; $\Gamma \leftarrow \emptyset$; $\Gamma_{\text{EmptySplits}} \leftarrow \emptyset$; ADDVERTEX$(\mathcal{T}, \Gamma,$ `noParent`$, s_{\text{init}},$ `noControl`$)$
5: **while** TIME$() < t_{\max}$
6: $\quad\Gamma_{\langle z,r\rangle} \leftarrow$ SELECTEQUIVALENCECLASS(Γ)
7: $\quad v \leftarrow$ EXPANDMOTIONTREE$(\mathcal{T}, \Gamma, \Gamma_{\langle z,r\rangle})$
8: $\quad$**if** $v.z \in \text{Accept}_{\mathcal{A}}$ **then return** TRAJ$(\mathcal{T}, v)$
9: $\quad$**if** VOL$(r) > \text{vol}_{\min}$ **and** $r \notin \mathcal{P}$ **then** REFINEPARTITION$(\Gamma, r, \Gamma_{\text{EmptySplits}})$
10: **return** `null`

Algorithm 1: Pseudocode for `Argos`.

The motion tree $\mathcal{T}$ is initialized by adding the vertex v_{init} as the root and setting $v_{\text{init}}.s$ to s_{init} (Algorithm 1:4). This creates the first equivalence class, $\Gamma = \{\Gamma_{\langle z_{\text{init}}, r_{\text{init}}\rangle}\}$, where $r_{\text{init}} =$ LOCATEREGION(s_{init}) and $v_{\text{init}} \in \Gamma_{\langle z_{\text{init}}, r_{\text{init}}\rangle}$. After the initialization, `Argos` iteratively expands $\mathcal{T}$ by adding new vertices and new edges until a solution is found or an upper bound on running time is exceeded (Algorithm 1:5–10). The overall process is driven by the following functions:

- SELECTEQUIVALENCECLASS(Γ) to select an equivalence class $\Gamma_{\langle z,r\rangle} \in \Gamma$ from which to expand $\mathcal{T}$ using selection penalties and heuristic costs defined over $\mathcal{A}$ and $D = (R, E)$ as the basis for the selection;
- EXPANDMOTIONTREE$(\mathcal{T}, \Gamma, \Gamma_{\langle z,r\rangle})$ to expand $\mathcal{T}$ by adding a new collision-free and dynamically feasible trajectory from a vertex associated with $\Gamma_{\langle z,r\rangle}$; and
- REFINEPARTITION$(\Gamma, r, \Gamma_{\text{EmptySplits}})$ to refine the decomposition D and the equivalence classes in Γ by partitioning r and redistributing the vertices associated with each $\Gamma_{\langle z',r\rangle} \in \Gamma$ that has r as its region.

The rest of the section describes these functions in more detail.

4.4.1. Selecting an equivalence class for the motion-tree expansion. The selection of an equivalence class $\Gamma_{\langle z,r\rangle}$ from Γ seeks to promote expansions of the motion tree toward an accepting automaton state. A weight $w(\Gamma_{\langle z,r\rangle})$ is associated with each $\Gamma_{\langle z,r\rangle}$ defined as

$$w(\Gamma_{\langle z,r\rangle}) = 2^{-d_{\mathcal{A}}(z)} \cdot (\hat{h}(\Gamma_{\langle z,r\rangle}))^{\alpha} \cdot \beta^{\text{NrSel}(\Gamma_{\langle z,r\rangle})}. \tag{21}$$

The equivalence class with the maximum weight is then selected for expansion, i.e.,

$$\text{SELECTEQUIVALENCECLASS}(\Gamma) \stackrel{\text{def}}{=} \text{argmax}_{\Gamma_{\langle z,r\rangle} \in \Gamma} w(\Gamma_{\langle z,r\rangle}). \tag{22}$$

Recall that $d_{\mathcal{A}}(z)$ denotes the minimum number of transitions in $\mathcal{A}$ to reach an accepting state from z. As such, $w(\Gamma_{\langle z,r\rangle})$ increases exponentially as $d_{\mathcal{A}}(z)$ decreases. This has the desired effect of promoting expansions from equivalence classes that get closer and closer to accepting automaton states.

The exponent $\alpha \geq 1$ serves as a parameter to tune the strength of the heuristic. To counter the greedy aspect of the heuristic, the term $\beta^{\text{NrSel}(\Gamma_{\langle z,r\rangle})}$ with $0 < \beta < 1$ is used to penalize each selection of $\Gamma_{\langle z,r\rangle}$, where $\text{NrSel}(\Gamma_{\langle z,r\rangle})$ denotes the number of times $\Gamma_{\langle z,r\rangle}$ has been selected for expansion. The penalization is necessary in order to avoid overexploration of $\Gamma_{\langle z,r\rangle}$ or become stuck when expansions from $\Gamma_{\langle z,r\rangle}$ are infeasible due to constraints imposed on the feasible motions by the underlying robot dynamics and the requirement to avoid collisions.

The term $\hat{h}(\Gamma_{\langle z,r\rangle})$ denotes the value of the heuristic cost $h(\Gamma_{\langle z,r\rangle})$ normalized in the range $[\epsilon, 1]$ as

$$\hat{h}(\Gamma_{\langle z,r\rangle}) = \max\{\epsilon, 1 - h(\Gamma_{\langle z,r\rangle})/h_{\text{up}}\}. \tag{23}$$

The normalization seeks to provide a uniform measure across different workspaces and different task specifications (since $h(\Gamma_{\langle z,r\rangle})$ depends on the physical dimensions of the workspace and the distances

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

among the regions of interest that must be visited). In the normalization, small values of $h(\Gamma_{\langle z,r\rangle})$ are mapped to values close to 1. The term h_{up} serves as an upper bound. The equivalence class $\Gamma_{\langle z,r\rangle}$ is considered to be too far away from an accepting automaton state when $h(\Gamma_{\langle z,r\rangle}) \geq h_{\text{up}}(1-\epsilon)$. In such cases, $\hat{h}(\Gamma_{\langle z,r\rangle})$ is set to ϵ. A small value $\epsilon > 0$ rather than zero is used in order to ensure that each $\Gamma_{\langle z,r\rangle}$ has a nonzero probability of being selected for future expansions. One possibility is to define $h_{\text{up}} = \max_{z' \in Z_{\mathcal{A}}, r' \in R} h(\Gamma_{\langle z',r'\rangle})$. Such definition would require precomputing all the heuristic values and is computationally feasible only when considering small automata. As an alternative, in this work, h_{up} is defined as

$$h_{\text{up}} = h(\Gamma_{\langle z_{\text{init}}, r_{\text{init}}\rangle}) + \max_{r' \in R} \text{MINCOSTPATH}_D(r', r_{\text{init}}), \tag{24}$$

where $r_{\text{init}} = \text{LOCATEREGION}(s_{\text{init}})$. In this way, h_{up} adds to $h(\Gamma_{\langle z_{\text{init}}, r_{\text{init}}\rangle})$ the maximum among the lengths of the shortest paths in the decomposition graph $D = (R, E)$ to reach r_{init} from any region in R. This maximum is computed with a single call to Dijkstra's shortest-path algorithm using r_{init} as the source. The maximum is initially set to $-\infty$ and is updated each time a node is extracted from the heap data structure associated with Dijkstra's algorithm.

INITEQUIVALENCECLASS$(\Gamma, \Gamma_{\langle z,r\rangle})$

1: **if** h_{up} is not defined **then**
2: $\quad h_{\text{up}} \leftarrow \text{DISCRETESEARCH}(\mathcal{A}, D, \langle z_{\text{init}}, r_{\text{init}}\rangle) + \max_{r' \in R} \text{MINCOSTPATH}_D(r', r_{\text{init}})$
3: $h \leftarrow \text{DISCRETESEARCH}(\mathcal{A}, D, \langle z, r\rangle)$; $\quad \hat{h} \leftarrow \max\{\epsilon, 1 - h/h_{\text{up}}\}$
4: $d \leftarrow \text{MINNREDGESTOACCEPT}(\mathcal{A}, z)$
5: **if** $\text{FIND}(\Gamma_{\text{emptySplits}}, \langle z, r\rangle) = \texttt{null}$ **then** $\text{NrSel}(\Gamma_{\langle z,r\rangle}) \leftarrow 0$
6: **else** $\text{NrSel}(\Gamma_{\langle z,r\rangle}) \leftarrow \text{VALUE}(\Gamma_{\text{EmptySplits}}, \langle z, r\rangle)$; $\text{REMOVE}(\Gamma_{\text{EmptySplits}}, \langle z, r\rangle)$
7: $w(\Gamma_{\langle z,r\rangle}) \leftarrow 2^{-d} \cdot \hat{h}^{\beta} \cdot \gamma^{\text{NrSel}(\Gamma_{\langle z,r\rangle})}$; $\quad \text{INSERT}(\Gamma, \Gamma_{\langle z,r\rangle})$

SELECTEQUIVALENCECLASS(Γ)

return $\text{argmax}_{\Gamma_{\langle z,r\rangle} \in \Gamma} w(\Gamma_{\langle z,r\rangle})$

Algorithm 2: Pseudocode for computing $w(\Gamma_{\langle z,r\rangle})$ when $\Gamma_{\langle z,r\rangle}$ is first created and for selecting an equivalence class from which to expand the motion tree.

As an implementation note, Γ is maintained as a maximum heap data structure. The selection penalty β is applied to $w(\Gamma_{\langle z,r\rangle})$ each time $\Gamma_{\langle z,r\rangle}$ is selected for expansion, i.e., $w(\Gamma_{\langle z,r\rangle}) \leftarrow \beta \cdot w(\Gamma_{\langle z,r\rangle})$, and $\text{NrSel}(\Gamma_{\langle z,r\rangle})$ is increased by one. Pseudocode for computing $w(\Gamma_{\langle z,r\rangle})$ when $\Gamma_{\langle z,r\rangle}$ is first created is given in Algorithm 2. The value h_{up} used for the normalization of the heuristic cost is computed only once and used thereafter (Algorithm 2:1–2). The discrete search described in Section 4.3 is invoked to compute $h(\Gamma_{\langle z,r\rangle})$ (Algorithm 2:3). The value $d_{\mathcal{A}}(z)$ is retrieved in constant time from the vector storing these values (Algorithm 2:4), which were computed earlier by `Argos` (Algorithm 1:3). The number of selections for $\Gamma_{\langle z,r\rangle}$ is set to zero when it is first created unless $\Gamma_{\langle z,r\rangle}$ resulted from the refinement process (Algorithm 2:5–6). In that case, as described in more detail in Section 4.4.3, $\Gamma_{\langle z,r\rangle}$ inherits the number of selections from its parent. After computing $w(\Gamma_{\langle z,r\rangle})$, $\Gamma_{\langle z,r\rangle}$ is inserted into Γ so that it can be used for future expansions of the motion tree $\mathcal{T}$ (Algorithm 2:7).

4.4.2. Expanding the motion tree. After selecting an equivalence class $\Gamma_{\langle z,r\rangle}$, the motion tree $\mathcal{T}$ is expanded from $\Gamma_{\langle z,r\rangle}$ by selecting a vertex from $\Gamma_{\langle z,r\rangle}$ and then extending a collision-free and dynamically feasible trajectory from $v.s$. Pseudocode is provided in Algorithm 3.

The selection of a vertex in $\Gamma_{\langle z,r\rangle}$ from which to expand the motion tree $\mathcal{T}$ seeks to promote expansions along the abstract path associated with $\Gamma_{\langle z,r\rangle}$. Let $\mathcal{P}_{i_1}$ denote the first region along the abstract path $\langle z, r\rangle, \langle z_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \langle z_{i_k}, \mathcal{P}_{i_k}\rangle$ from $\langle z, r\rangle$ to an accepting automaton state as determined by the discrete search during the computation of the heuristic cost $h(\Gamma_{\langle z,r\rangle})$ (see Section 4.3). The target point p is generated by sampling a random point inside one of the regions along the shortest path in the decomposition $D = (R, E)$ from r to $\mathcal{P}_{i_1}$. Recall that such shortest paths have already been precomputed by running Dijkstra's single source shortest-path algorithm using each $\mathcal{P}_i$ as the source (Algorithm 1:2). As a result, target sampling is done in constant time by first selecting a region along the shortest path from r to $\mathcal{P}_{i_1}$ uniformly at random and then generating a random point inside that

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

ExpandMotionTree($\mathcal{T}, \Gamma, \Gamma_{\langle z,r\rangle}$)
1: $p \leftarrow$ SelectTarget($\Gamma_{\langle z,r\rangle}$); $v \leftarrow$ SelectVertex($\Gamma_{\langle z,r\rangle}, p$)
2: $u \leftarrow$ SelectControl($v.s, p$)
3: **for** several steps **do**
4: AdjustControl($v.s, p, u$); $s_{\text{new}} \leftarrow$ motion($v.s, u, dt$)
5: **if** valid(s_{new}) $= \perp$ **then return** v
6: $v_{\text{new}} \leftarrow$ AddVertex($\mathcal{T}, \Gamma, v, s_{\text{new}}, u$)
7: **if** $v_{\text{new}}.z \in \text{Accept}_{\mathcal{A}}$ **then return** v_{new}
8: $v \leftarrow v_{\text{new}}$
9: **return** v

Algorithm 3: Pseudocode for expanding the motion tree from the selected equivalence class $\Gamma_{\langle z,r\rangle}$.

region. Expansions toward such target point will promote progress along the abstract path associated with $\Gamma_{\langle z,r\rangle}$.

After the target point p has been generated, SelectVertex($\Gamma_{\langle z,r\rangle}$, p) selects for expansion the vertex $v \in \Gamma_{\langle z,r\rangle}$ that is closest to p according to the distance $||\text{proj}(v.s), p||$. A trajectory is generated from $v.s$ by applying control inputs for several steps. A common strategy in sampling-based motion planning is to select the control input uniformly at random in order to promote expansions along different directions.[11] When available, PID controllers that seek to steer $v.s$ toward p can also be used. For example, a PID controller for a robotic vehicle would adjust the steering so that the vehicle would turn and move toward p. Note that `Argos` does not place any requirements on these controllers, so exact steering is not needed. The function motion($v.s$, u, dt) is used to simulate the motion dynamics and determine the new state s_{new} obtained by applying the input control u to $v.s$ for one time step dt (Algorithm 3:4). If s_{new} is in collision or some state value exceeds the defined bounds, as determined by valid(s_{new}), the tree expansion terminates early (Algorithm 3:5). Otherwise, a new vertex v_{new} is added to $\mathcal{T}$ with s_{new} as its state and v as its parent (Algorithm 3:6). If $\text{traj}_{\mathcal{T}}(v_{\text{new}})$ reaches an accepting automaton state, i.e., $\hat{\delta}_{\mathcal{A}}(z_{\text{init}}, \text{word}(\text{traj}_{\mathcal{T}}(v_{\text{new}}))) \in \text{Accept}_{\mathcal{A}}$, the expansion terminates successfully with $\text{traj}_{\mathcal{T}}(v_{\text{new}})$ as the solution (Algorithm 3:7). Otherwise, the tree expansion continues from the new vertex (Algorithm 3:8).

AddVertex($\mathcal{T}, \Gamma, v, s_{\text{new}}, u$)
1: $v_{\text{new}} \leftarrow$ NewVertex($\mathcal{T}$); v_{new}.parent $\leftarrow v$; $v_{\text{new}}.s \leftarrow s_{\text{new}}$; $v_{\text{new}}.u \leftarrow u$
2: $v_{\text{new}}.r \leftarrow$ LocateRegion(s_{new}); v_{new}.prop $\leftarrow$ PropLabel($v_{\text{new}}.r$)
3: **if** v_{new}.prop $\notin \{\pi_1, \ldots, \pi_n\}$ **or** v_{new}.prop $= v$.prop **then** $v_{\text{new}}.z \leftarrow v.z$
4: **else** $v_{\text{new}}.z \leftarrow \delta_{\mathcal{A}}(v.z, v_{\text{new}}.\text{prop})$
5: $\langle z, r\rangle \leftarrow \langle v_{\text{new}}.z, v_{\text{new}}.r\rangle$
6: **if** $(\Gamma_{\langle z,r\rangle} \leftarrow$ find($\Gamma, \langle z, r\rangle$)) $=$ `null` **then**
7: $\Gamma_{\langle z,r\rangle} \leftarrow$ new(); nrSel $\leftarrow 0$; InitEquivalenceClass($\Gamma, \Gamma_{\langle z,r\rangle}$, nrSel)
8: insert($\Gamma_{\langle z,r\rangle}, v_{\text{new}}$)
9: **return** v_{new}

Algorithm 4: Pseudocode for adding a new vertex to the motion tree.

Pseudocode for adding a new vertex to $\mathcal{T}$ is given in Algorithm 4. As mentioned, the new vertex v_{new} has s_{new} as its state, v as its parent, and u as the input control that was applied to generate s_{new} (Algorithm 4:1). The new vertex keeps track of the corresponding region in the workspace decomposition and the proposition label associated with that region (Algorithm 4:2). If no region of interest $\mathcal{P}_1, \ldots, \mathcal{P}_n$ has been reached or v_{new} is still in the same region as its parent, then $v_{\text{new}}.z$ is set to $v.z$ (Algorithm 4:3). Otherwise, $v_{\text{new}}.z$ is obtained by the automaton transition from $v.z$ with prop(v_{new}) as the input (Algorithm 4:4). A search is performed to determine whether $\Gamma_{\langle v_{\text{new}}.z, v_{\text{new}}.r\rangle}$ already exists in Γ. If not, a new equivalence class is created and added to Γ (Algorithm 4:7). At this time, InitEquivalenceClass, as described in Section 4.4.1 and Algorithm 2, invokes the discrete search over $\mathcal{A}$ and $D = (R, E)$ to compute the heuristic cost and uses it to determine the weight associated with the newly created equivalence class. In this way, `Argos` can use the new equivalence classes to expand $\mathcal{T}$ in future iterations.

4.4.3. Refining the partition. After expanding $\mathcal{T}$ from $\Gamma_{\langle z,r\rangle}$, if r is not one of the regions of interest $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and its volume is larger then a minimum threshold, `Argos` refines the workspace

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

decomposition $D = (R, E)$ by partitioning r into several nonoverlapping regions $r_{\text{part}_1}, \ldots, r_{\text{part}_k}$. The reason for the partition is to better identify which parts of r lead to successful expansions. The decomposition refinement, as shown by the experimental results, also reduces the dependency of `Argos` on the initial decomposition. As a result of partitioning r, tree vertices in any equivalence class $\Gamma_{\langle z',r\rangle} \in \Gamma$ with r as its region are distributed among $\Gamma_{\langle z',r_{\text{part}_1}\rangle}, \ldots, \Gamma_{\langle z',r_{\text{part}_k}\rangle}$. After redistributing the vertices, those $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ that are nonempty, i.e., $|\Gamma_{\langle z',r_{\text{part}_i}\rangle}| > 0$ are added to Γ so that they can be used for future expansions. Pseudocode for REFINEPARTITION(Γ, r, $\Gamma_{\text{EmptySplits}}$) is given in Algorithm 5. More details follow.

REFINEPARTITION($\Gamma, r, \Gamma_{\text{EmptySplits}}$)

1: $\langle r_{\text{part}_1}, \ldots, r_{\text{part}_k}\rangle \leftarrow$ PARTITIONREGION(r); UPDATEDECOMPOSITION($D, r, \langle r_{\text{part}_1}, \ldots, r_{\text{part}_k}\rangle$)
2: UPDATESHORTESTPATHS($D, \langle r_{\text{part}_1}, \ldots, r_{\text{part}_k}\rangle, \mathcal{P}$)
3: keys $\leftarrow \{\langle z', r\rangle : \Gamma_{\langle z',r\rangle} \in \Gamma\}$; $S \leftarrow \emptyset$
4: **for** $\langle z', r\rangle \in$ keys **do**
5: **for** $i = 1 \ldots k$ **do**
6: $\Gamma_{\langle z',r_{\text{part}_i}\rangle} \leftarrow$ NEW(); NrSel($\Gamma_{\langle z',r_{\text{part}_i}\rangle}$) $\leftarrow$ NrSel($\Gamma_{\langle z',r\rangle}$); INSERT($S, \Gamma_{\langle z',r_{\text{part}_i}\rangle}$)
7: **for** $v \in \Gamma_{\langle z,r\rangle}$ **do**
8: $r_{\text{part}_i} \leftarrow$ LOCATEREGION($\langle r_{\text{part}_1}, \ldots, r_{\text{part}_k}\rangle$, sstate($v$)); INSERT($\Gamma_{\langle z',r_{\text{part}_i}\rangle}, v$)
9: REMOVE(Γ, keys)
10: **for** $\Gamma_{\langle z',r_{\text{part}_i}\rangle} \in S$ **do**
11: **if** $|\Gamma_{\langle z',r_{\text{part}_i}\rangle}| > 0$ **then** INITEQUIVALENCECLASS($\Gamma, \Gamma_{\langle z',r_{\text{part}_i}\rangle}$)
12: **else** INSERT($\Gamma_{\text{EmptySplits}}, \langle z', r_{\text{part}_i}\rangle$, NrSel($\Gamma_{\langle z',r_{\text{part}_i}\rangle}$))

Algorithm 5: Pseudocode for refining the partition.

In the case of a triangulation decomposition, PARTITIONREGION(r) (Algorithm 5:1) partitions r into three triangles using the centroid as the splitting point. In the case of a subdivision decomposition, r is partitioned into two halves along the largest dimension. The decomposition graph $D = (R, E)$ is also updated to reflect the removal of r and the addition of $r_{\text{part}_1}, \ldots, r_{\text{part}_k}$. The shortest paths from each r_{part_i} to every $\mathcal{P}_j$ need to also be updated since `Argos` uses these shortest paths to compute the heuristic costs associated with equivalence classes in Γ. As rerunning Dijkstra's shortest-path algorithm is computationally expensive (since region partitioning will be performed hundreds or thousands of times during a typical run of `Argos`), the shortest paths are locally adjusted from their neighbors (Algorithm 5:2). In particular, the shortest path from r_{part_i} to $\mathcal{P}_j$ is computed by looking up the shortest paths from the neighbors of r_{part_i} to $\mathcal{P}_j$. The cost of the edge from r_{part_i} to its neighbor r' is added to the path from r' to $\mathcal{P}_j$. The shortest among the resulting paths is then used as the path from r_{part_i} to $\mathcal{P}_j$, i.e., $\min_{r' \in \text{neigh}(r_{\text{part}_i})}(\text{cost}(r', r_{\text{part}_i}) + \text{MINCOSTPATH}_D(r', \mathcal{P}_j))$.

In the next step, all the equivalence classes $\Gamma_{\langle z',r\rangle} \in \Gamma$ with r as their region are retrieved as they are affected by the partition of r (Algorithm 5:3). For each $\Gamma_{\langle z',r\rangle}$, new equivalence classes $\Gamma_{\langle z',r_{\text{part}_1}\rangle}, \ldots, \Gamma_{\langle z',r_{\text{part}_k}\rangle}$ are created and are temporarily inserted into a set S. Each $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ inherits the number of selections from $\Gamma_{\langle z',r\rangle}$ (Algorithm 5:5–6). The vertices in $\Gamma_{\langle z',r\rangle}$ are distributed among $\Gamma_{\langle z',r_{\text{part}_1}\rangle}, \ldots, \Gamma_{\langle z',r_{\text{part}_k}\rangle}$ depending on the new regions associated with them (Algorithm 5:7–8).

After processing all the $\Gamma_{\langle z',r\rangle} \in \Gamma$ with r as their region, they are all removed from Γ (Algorithm 5:9). Iterating over S, all the new nonempty equivalence classes are added to Γ (Algorithm 5:10–11). At this time, as described in Section 4.4.1 and Algorithm 2, INITEQUIVALENCECLASS(Γ, $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$) uses the discrete search over $\mathcal{A}$ and D to determine the heuristic cost of $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$.

All the empty $\Gamma_{\langle z',r_{\text{part}_i}\rangle} \in S$ are inserted into $\Gamma_{\text{EmptySplits}}$. This is done to keep track of the number of times $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ has been selected. Later tree expansions could cause the insertion of $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ into Γ. In such cases, it is not desirable to use zero as the number of selections since $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ came from a partition of $\Gamma_{\langle z',r\rangle}$ which had been selected in the past. Starting with a zero selection would reward $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$. Using the number of selections of $\Gamma_{\langle z',r\rangle}$ for $\Gamma_{\langle z',r_{\text{part}_i}\rangle}$ makes it possible to carry over the selection penalties applied to its parent.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

5. Experiments and Results

The performance of `Argos` is tested in simulation using ground and aerial vehicle models with nonlinear dynamics where the robot operates in complex environments and is required to perform various tasks specified as regular languages over regions of interest. `Argos` is compared to `LTLSyclop` and `RRT`-based planners. The impact of the initial decomposition and of the partition refinement is also evaluated.

5.1. Scenes and robot models

Figure 5 shows an illustration of the scenes and robot models with nonlinear dynamics used in the experiments. In the first scenario, a ground vehicle is required to move up and down an elevated racetrack while avoiding collisions with numerous obstacles scattered at random throughout the environment. The vehicle is controlled by setting the engine force and changing the steering angle. The vehicle state s includes position (x, y, z), orientation frame (rot), steering angle (ψ), linear velocities (v_x, v_y, v_z), and angular velocities $(\omega_x, \omega_y, \omega_z)$. The projection function is defined as $\text{PROJ}(s) = (x, y)$ (since it is a ground vehicle, z is not used). The physics-based engine Bullet[13] is used as the underlying simulator. The steering linkages in the vehicle model follow the principles of Ackermann steering geometry. The vehicle model in Bullet seeks to simulate its interactions with the environment, taking into account the wheel friction, suspension stiffness, suspension damping, friction slip, and compression. Due to the complexity of the interactions between the vehicle and the nonflat terrain, Bullet, as other physics-based engines, relies on numerical simulations since closed formulas are generally not available. The parameter settings used for the experiments are the same as the default values in the version 2.8.3 of the Bullet vehicle simulator. The environment dimensions are 120 m × 120 m. The maximum height of the racetrack is 4 m. The vehicle dimensions are 2.2 m × 1 m × 0.45 m. The maximum speed is 14 m/s, the maximum steering angle is $\pi/3.6$ rad, and the maximum rate of change for the steering angle is $\pi/3$ rad/s. The time step is $dt = 0.05$ s.

In the second scenario, a snake-like robot, modeled as a car pulling trailers, is required to move in a maze environment. The state $s = (x, y, \theta_0, v, \psi, \theta_1, \ldots, \theta_\ell)$ includes the position (x, y), orientation (θ_0), velocity $(|v|)$, and steering angle (ψ) of the head link as well as the orientation θ_i of the i-th trailer, where ℓ is in the number of trailers. The projection is defined as $\text{PROJ}(s) = (x, y)$. The robot is controlled by setting the acceleration (u_a) and the rotational velocity of the steering angle (u_ω). The differential equations of motion, adapted from[37] [pp. 731], are defined as

$$\dot{x} = v\cos(\theta_0)\cos(\psi) \quad \dot{y} = v\sin(\theta_0)\cos(\psi) \quad \dot{\theta_0} = v\sin(\psi)/L \quad \dot{v} = u_a \quad \dot{\psi} = u_\omega \tag{25}$$

$$\dot{\theta_i} = \tfrac{v}{d}(\sin(\theta_{i-1}) - \sin(\theta_0)) \textstyle\prod_{j=1}^{i-1} \cos(\theta_{j-1} - \theta_j), \tag{26}$$

where $L = 0.15$ m is the body length and d is the hitch length (set to a small value, $d = 0.01$ m, so that the robot resembles a snake). $\text{MOTION}(s, u, dt)$ is implemented using fourth-order Runge–Kutta numerical integration. In the experiments, the number of links is set to $\ell = 10$. The head and each link has length 0.15 m and width 0.08 m. The environment dimensions are 12 m × 12 m. The following bounds are also used: speed $|v| \leq 5$ m/s, acceleration $|u_a| \leq 2$ m/s^2, steering angle $|\psi| \leq \pi/3.6$ rad, rotational velocity of steering angle $|u_\omega| \leq \pi/3$ rad/ s. The time step is $dt = 0.05$ s.

In the third scenario, an aerial vehicle is required to fly through small openings in various rooms. The aerial vehicle model is based by augmenting a ground model to fly parallel to the xy-plane. The state $(x, y, z, \theta, v, \psi, v_z)$ includes the position (x, y, z), orientation (θ), velocity on the xy-plane (v), steering angle (ψ), and the velocity along the z-axis (v_z). The projection is defined as $\text{PROJ}(s) = (x, y, z)$. In addition to u_a and u_ω, the aerial vehicle is controlled by setting the acceleration u_{a_z} along the z-axis. The differential equations of motion are defined as

$$\dot{x} = v\cos(\theta)\cos(\psi) \quad \dot{y} = v\sin(\theta)\cos(\psi) \quad \dot{z} = v_z \tag{27}$$

$$\dot{\theta} = v\sin(\psi)/L \quad \dot{v} = u_a \quad \dot{\psi} = u_\omega \quad \dot{v}_z = u_{a_z} \tag{28}$$

$\text{MOTION}(s, u, dt)$ is again implemented using fourth-order Runge–Kutta integration. The following setup was used in the experiments: environment dimensions 120 m × 120 m × 14 m, aerial-vehicle dimensions 8.6 m × 4.3 × 4.3 m, body length $L = 8.6$ m, speed $|v|, |v_z| \leq 5$ m/s, acceleration

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

Table I. The number of states in the minimal DFA required for the various tasks considered in the experiments.

Task	Number of states in minimal DFA	at $n = 19$
ϕ^n_{seq}	n	19
ϕ^n_{cov}	2^n	524,288
ϕ^n_{po}	$2^{\lceil \frac{n}{2} \rceil} + 2^{\lfloor \frac{n}{2} \rfloor} - 1$	1,535
$\phi^n_{\text{zig-zag}}$	$\sum_{\ell=0}^{n} \binom{\lceil \frac{n}{2} \rceil}{\lceil \frac{\ell}{2} \rceil} \binom{\lfloor \frac{n}{2} \rfloor}{\lfloor \frac{\ell}{2} \rfloor}$	184,756

$|u_a|, |u_{a_z}| \leq 2$ m/s^2, steering angle $|\psi| \leq \pi/3.6$ rad, rotational velocity of steering angle: $|u_\omega| \leq \pi/3$ rad/s. The time step is $dt = 0.05$ s.

5.2. Tasks and problem instances

The experiments use sequencing ϕ^n_{seq}, coverage ϕ^n_{cov}, partial-order ϕ^n_{po}, and zig-zag $\phi^n_{\text{zig-zag}}$ tasks as defined in Eqs. (1)–(7) (Section 1 and 3.2). For each task, the number of propositions is varied as $n = 5, 7, 9, \ldots, 19$. Table I shows the number of states in the minimal DFA required to represent each task.

For each scene and each n, 60 instances are created by randomly placing the robot and the regions of interest $\mathcal{P}_1, \ldots, \mathcal{P}_n$. These instances are denoted by $\mathcal{I}_{\langle \text{scene}, n \rangle}$. In each instance, the robot placement is generated by repeatedly sampling a position and orientation at random until the robot is not in collision with any of the obstacles. The placement of $\mathcal{P}_k$ is generated by repeatedly sampling a position and orientation at random until $\mathcal{P}_k$ is at least a certain distance d away from the robot and the previously placed regions $\mathcal{P}_1, \ldots, \mathcal{P}_{k-1}$. The distance d is used to ensure that the regions of interest are not too close to one another or to the robot. In the case of the physics-based vehicle (scene 1), $\lceil 0.4n \rceil$ of the regions of interest are scattered on the elevated ractrack, while the remaining regions are placed outside the racetrack. In this way, the vehicle is forced to go up and down the racetrack in accordance with the task specification. Figure 5 shows some examples of the instances used in the experiments.

Let X denote a planner, S a scene, ϕ a task, and n the number of regions of interest. The planner X is run 60 times, one for each instance in $\mathcal{I}_{\langle S, n \rangle}$, with the objective of finding a solution for ϕ. A maximum time of 40 s is set for each run. Results report the average runtime and standard deviation for $\langle X, \phi, S, n \rangle$ calculated after dropping the five best and worst runtimes to reduce the influence of outliers. Runtime includes everything from reading the input file, which describes the workspace, obstacles, regions of interest, robot models, and the DFA representing the task to be solved, to reporting that a solution is found. Reading the input file is in the order of milliseconds. Experiments were run on an Intel Core i7 (CPU: 1.90 GHz, RAM: 4 GB) using Ubuntu 14.04 and GNU g++-4.8.2.

5.3. Results

`Argos` is compared to `LTLSyclop`.[4–6,43,44] The implementation of `LTLSyclop` has been optimized to use features recommended in the various publications such as switching between shortest and random paths, abandoning the current guide when little progress is made, and starting the discrete search from recently explored abstract states instead of the initial abstract state.

Experiments were also run with an `RRT`-based approach, which used the automaton as an external monitor to keep track of the automaton states associated with each trajectory. The `RRT`-based approach, however, failed to solve the problem instances. For this reason, it is not included in the graphs. Without guidance toward accepting automaton states, the `RRT` exploration became stuck.

5.3.1. Scalability with the number of propositions. Figure 6 shows the results for each scene and task when varying the number of propositions $n \in \{5, 7, \ldots, 19\}$. Results show that `Argos` is an order of magnitude faster than `LTLSyclop`. As n increases, `LTLSyclop` starts to time out (set to 40s) and so is unable to solve the larger problem instances. These results are in agreement with related work,[4–6,43,44] which has shown experimental results for `LTLSyclop` with tasks involving 1–10 propositions. `LTLSyclop` often wastes valuable computational time before realizing that the current abstract path, which it seeks to follow, should be abandoned and a new one should be computed instead. Also, `LTLSyclop` considers for expansions tree vertices that are along the abstract path that

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

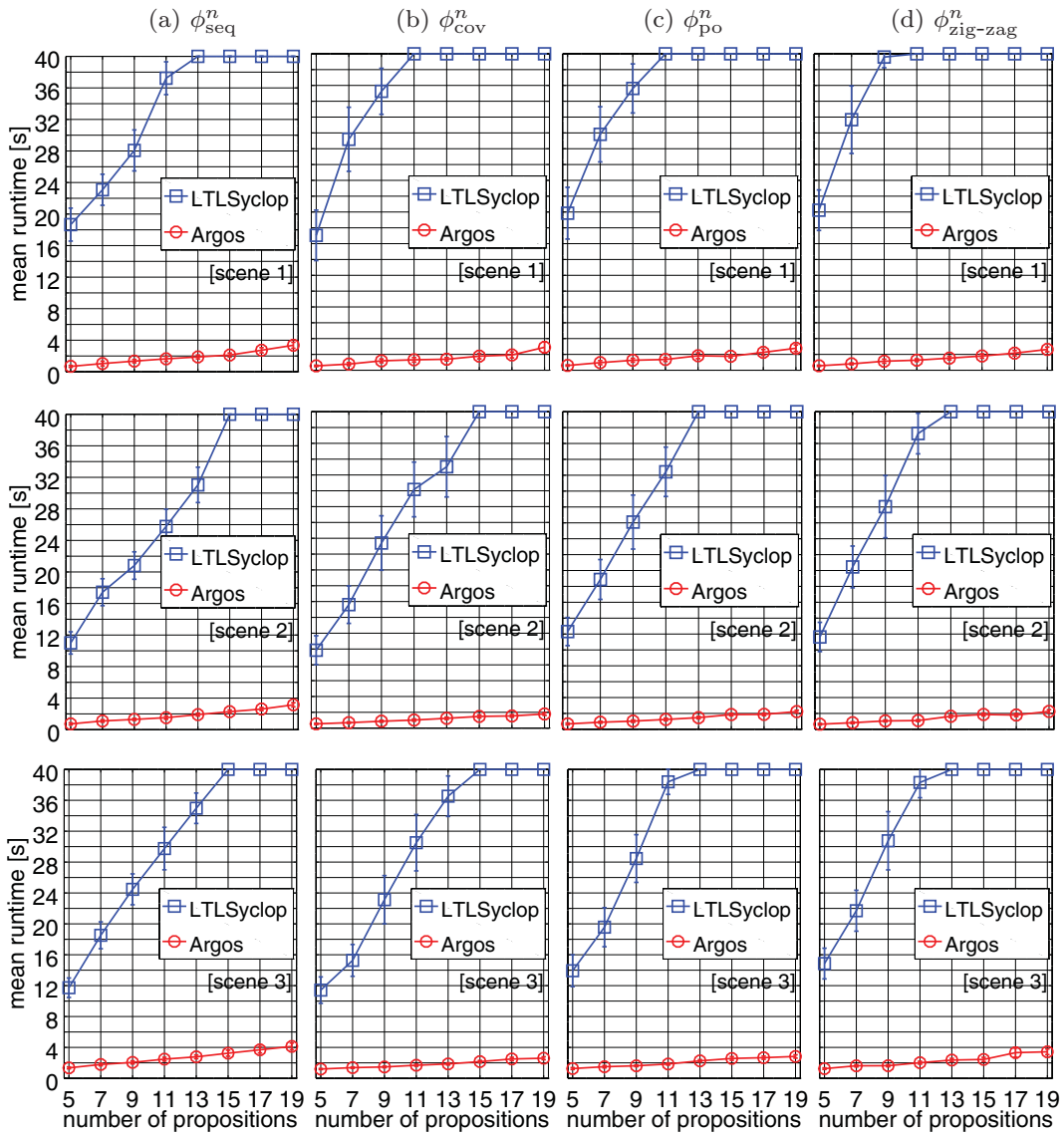


Fig. 6. Results on running time as a function of the number of propositions $n \in \{5, 7, \ldots, 19\}$ for various tasks and scenes. Bars indicate one standard deviation. The RRT-based approach, which used the automaton as an external monitor, is not included in the graphs as it was not able to solve the problem instances. A maximum time of 40 s is set for each run. Runtime includes everything from reading the input file to reporting that a solution is found.

it is following. This makes it difficult to explore along new directions which is necessary in order to find more feasible abstract paths toward accepting automaton states.

In contrast, Argos efficiently solves in 2–4 s even the largest problem instances with $n = 19$. The efficiency of Argos comes from the partition of the motion tree into equivalence classes, heuristic costs associated with each equivalence class, selection penalties, and the partition refinement. In fact, the partition into equivalence classes makes it possible to group together vertices that provide the same abstract information. Heuristic costs based on short abstract paths estimate the feasibility of reaching an accepting automaton state from each equivalence class. By applying a penalty after each selection of an equivalence class, Argos reduces the likelihood of overexploration or becoming stuck while attempting expansions from what could turn out to be an infeasible equivalence class. By refining the partition after each expansion, Argos promotes rapid expansions while discovering new ways to reach accepting automaton states. All these factors make it possible to efficiently plan

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

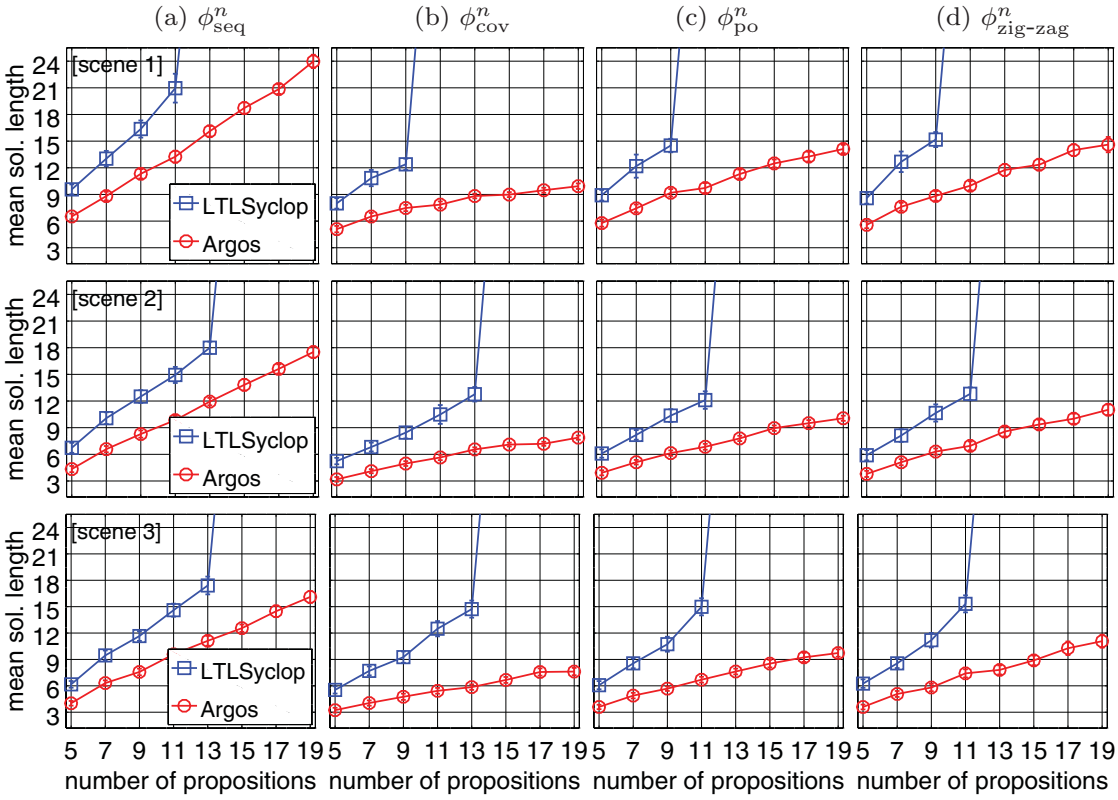


Fig. 7. Results on solution length as a function of the number of propositions $n \in \{5, 7, \ldots, 19\}$ for various tasks and scenes. Bars indicate one standard deviation. Solution length is measured as the distance traveled by the robot. The graphs show the solution length divided by the length of the diagonal of the workspace bounding box.

collision-free and dynamically feasible trajectories that satisfy task specifications given as regular languages.

Figure 7 shows the results on solution length, measured as the distance traveled by the robot. As shown, `Argos` generates considerably shorter solutions. This is mainly due to the heuristic costs associated with each equivalence class which promote expansions from those equivalence classes that are deemed to be close to accepting automaton states. Also note that considerably shorter solutions are generated for the coverage task ϕ^n_{cov} than for the sequencing task ϕ^n_{seq}. This is expected since in the coverage task, `Argos` is given the flexibility to visit the regions in any order.

5.3.2. Impact of the workspace decomposition. Subdivision decompositions were used for all scenes. Triangulations were used only for scenes 1 and 2 since 3 involves an aerial vehicle. Experiments were conducted with Tri_{Del}, Tri_{35}, Tri_{50}, Tri_{75}, where Tri_{Del} denotes a confirming Delaunay triangulation and Tri_ℓ corresponds to a triangulation where the average area of a triangle is approximately $\text{area}(\mathcal{W})/\ell^2$. Figure 5 shows some examples.

Results in Figs. 6 and 7 were obtained by using the triangulation Tri_{50} for scenes 1 and 2, and the subdivision decomposition for scene 3. Figure 8 shows results when varying the workspace decomposition. The results indicate that the performance of `LTLSyclop` becomes worse when considering fine-grained decompositions, such as Tri_{75}, as it becomes more difficult to search the product space of the automaton and the workspace decomposition and find feasible abstract paths. Similar observations were made in related work regarding the performance of `LTLSyclop`.[4–6,43]

The results show that `Argos` works well with different workspace decompositions. The partition refinement that takes place during the motion-tree expansion reduces its dependence on the initial workspace decomposition. Nevertheless, the initial decomposition should not be too fine grained as it makes each region behave as a vertex (since a small number of vertices will be associated with each region). When the initial decomposition is too coarse, `Argos` might not be as efficient in the

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

Table II. Impact of the partition refinement. Entries marked with X indicate failure to solve the problem instances within the 40 s bound per run.

			Number of propositions							
			5	7	9	11	13	15	17	19
Task	Method	Refinement	mean runtime [s]							
ϕ_{seq}	`Argos`	yes	0.7	1.0	1.4	1.6	1.9	2.1	2.7	3.4
(scene 1)	`Argos`	no	0.7	1.2	1.2	1.5	1.8	2.8	3.4	6.5
	`LTLSyclop`	yes	25.3	39.6	X	X	X	X	X	X
	`LTLSyclop`	no	18.7	23.1	28.1	37.2	X	X	X	X
ϕ_{cov}	`Argos`	yes	0.5	0.7	0.9	1.0	1.2	1.5	1.5	1.8
(scene 2)	`Argos`	no	1.2	1.9	2.2	3.5	4.2	3.9	4.5	6.4
	`LTLSyclop`	yes	21.2	36.6	X	X	X	X	X	X
	`LTLSyclop`	no	9.8	15.6	23.4	30.2	33.0	X	X	X
ϕ_{po}	`Argos`	yes	0.7	0.9	1.0	1.2	1.5	1.9	1.9	2.2
(scene 2)	`Argos`	no	2.9	3.7	3.7	4.4	4.9	6.3	5.7	6.8
	`LTLSyclop`	yes	26.4	38.5	X	X	X	X	X	X
	`LTLSyclop`	no	12.3	18.9	26.1	32.4	X	X	X	X
$\phi_{\text{zig-zag}}$	`Argos`	yes	1.2	1.6	1.6	2.0	2.4	2.4	3.3	3.4
(scene 3)	`Argos`	no	2.2	2.8	2.9	3.9	3.9	4.3	5.3	5.9
	`LTLSyclop`	yes	29.0	38.6	X	X	X	X	X	X
	`LTLSyclop`	no	14.9	21.7	30.8	38.3	X	X	X	X

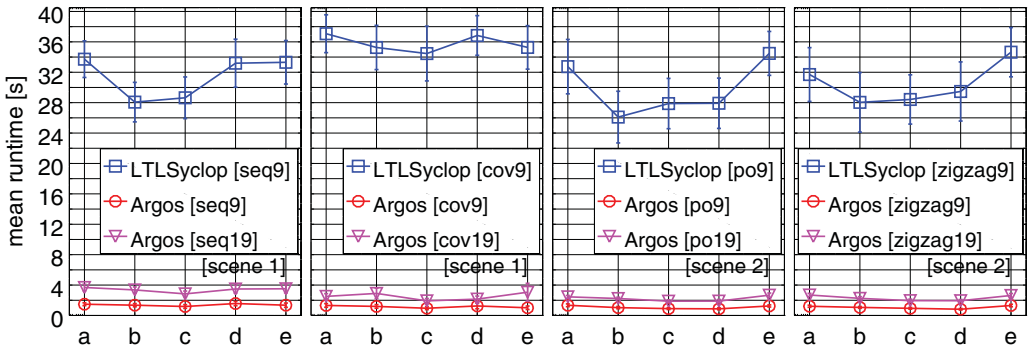


Fig. 8. Results when using various workspace decompositions: (a) $\text{Tri}_{75.}$ (b) $\text{Tri}_{50.}$ (c) $\text{Tri}_{35.}$ (d) $\text{Tri}_{\text{Del.}}$ (e) Subdivision.

beginning but will quickly improve as the initial decomposition is refined during the motion-tree expansion. When considering a new problem instance, we recommend using a subdivision as the initial workspace decomposition since it is simpler to use than a triangulation and applicable to both 2D and 3D environments.

5.3.3. Impact of the partition refinement. Experiments were also conducted to evaluate the impact of the partition refinement on the overall performance of `Argos`. We also modified the implementation of `LTLSyclop` so that it could also refine the workspace decomposition as it expanded the motion tree. Table II provides a summary of the results.

The results indicate the importance of the partition refinement on the performance of `Argos`, especially as the problems become more challenging. The partition refinement makes it possible for `Argos` to effectively discover new ways to expand the search toward accepting automaton states.

The refinement process has the opposite effect on the performance of `LTLSyclop`. As the refinement considerably increases the number of regions, it becomes increasingly difficult for `LTLSyclop` to find feasible abstract paths that it can follow.

5.3.4. Runtime distribution. Figure 9 shows the percentage of runtime taken by the various components of `Argos`. As the number of propositions increases, the percentage of the runtime taken by the workspace decomposition becomes smaller. In the case of scene 3, which corresponds

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

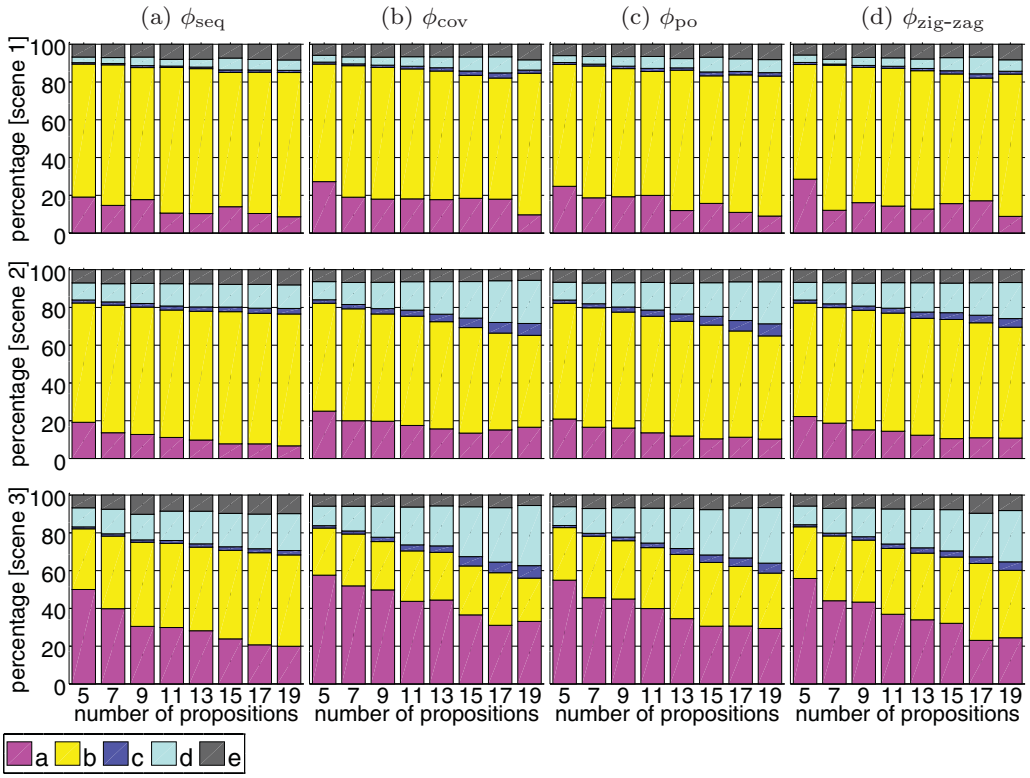


Fig. 9. Runtime percentage for various components of `Argos`: (a) Workspace decomposition (Algorithm 1:1–2) (b) MOTION + VALID (Algorithm 3:4–5) (c) Heuristic costs for new equivalence classes (Algorithm 4:7) (d) Partition refinement (Algorithm 5) (e) Other. Results correspond to `Argos` being run with subdivision workspace decompositions.

to the aerial vehicle model, the subdivision decomposition requires more time since the workspace is three dimensional. The simulation of motion dynamics and collision checking take most of the time as the motion tree requires tens of thousands of vertices to solve the problem instances. The computation of the heuristic costs for equivalence classes created as a result of motion-tree expansions increases quite slowly with the number of propositions, demonstrating the efficiency of the discrete search. The percentage of time taken by the partition refinement also increases slowly with the number of propositions. Most of the time in the partition refinement, around 93%, is taken by the computation of the heuristic costs for the new equivalence classes resulting from the partition while the rest is taken by updating the decomposition graph and redistributing the tree vertices among the partitioned equivalence classes. Also note the small percentage taken by other components (see category (e) in Fig. 9) including SELECTABSTRACTCLASS, LOCATEREGION, and miscellaneous bookkeeping updates.

5.3.5. Impact of the heuristic strength and selection penalties. All the results presented so far on the performance of `Argos` have used $\alpha = 8$ as the heuristic strength and $\beta = 0.95$ as the selection penalty when computing the weights associated with the equivalence classes (Section 4.4.1). Table III provides a summary of the results when varying the heuristic strength and selection penalty. The results show that the selection penalty is necessary to ensure that `Argos` does not become stuck while attempting to expand the motion tree from what could be an infeasible equivalence class. In fact, the runtime of `Argos` increases significantly when $\beta = 0.999$ which applies almost no selection penalty. When β is too small, `Argos` benefits less from the heuristic and as a result the running time increases. Similar observations hold for the heuristic strength α: a large value makes `Argos` too greedy and a small value diminishes the benefits of the heuristic.

The results show, however, that `Argos` works well for a wide range of parameter values. When considering a new problem instance, our recommendation is to start with $\alpha = 8$ and $\beta = 0.95$ as it worked well for a variety of scenes, tasks, and robots. The recommended range is $\alpha \in [4, 10]$ and $\beta \in [0.85, 0.97]$.

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

Table III. Results on the mean runtime [s] when varying the heuristic strength (α) and the selection penalty (β) used in the computation of the weights associated with the equivalence classes (Section 4.4.1). Results are shown for the case of scene 2 with $\phi^{19}_{\text{zig-zag}}$ as the task and Tri_{50} as the workspace decomposition used by `Argos`.

$\beta \downarrow$	$\alpha \rightarrow$	1	2	4	8	10	12
0.999		18.3 s	22.7 s	26.7 s	30.4 s	37.9 s	35.8 s
0.97		4.5 s	3.8 s	3.2 s	4.0 s	3.7 s	7.1 s
0.95		4.4 s	3.4 s	3.4 s	2.6 s	3.2 s	6.6 s
0.90		5.6 s	4.2 s	3.1 s	2.8 s	2.6 s	6.0 s
0.85		5.7 s	4.8 s	4.1 s	3.1 s	2.9 s	5.0 s
0.70		6.1 s	6.4 s	5.7 s	4.2 s	3.4 s	7.8 s
0.50		8.4 s	8.3 s	8.4 s	5.7 s	4.6 s	14.3 s

Table IV. Mean runtime when using (a) a differential-drive and (b) an Ackerman steered vehicle model. Results were obtained using scene 2. Entries marked with X indicate failure to solve the problem instances within the 40 s bound per run.

		ϕ^{19}_{seq}	ϕ^{19}_{cov}	ϕ^{19}_{po}	$\phi^{19}_{\text{zig-zag}}$
(a)	`Argos`	1.21 s	1.24 s	1.32 s	1.28 s
(a)	`LTLSyclop`	24.61 s	31.49 s	X	X
(b)	`Argos`	1.86 s	1.89 s	1.95 s	1.83 s
(b)	`LTLSyclop`	33.82 s	X	X	X

5.3.6. Using other vehicle models. Experiments so far have used a physics-based ground vehicle model, a snake-like robot model, and an aerial vehicle model. The results have shown that `Argos` can efficiently plan for these different vehicle models. To further make this point, additional experiments are conducted using a differential-drive model and an Ackerman steered vehicle operating in scene 2 (Fig. 5).

The equations of motion for the differential drive are defined as

$$\dot{x} = (r/2)(\omega_\ell + \omega_r)\cos(\theta) \quad \dot{y} = (r/2)(\omega_\ell + \omega_r)\sin(\theta) \quad \dot{\theta} = (r/L)(\omega_r - \omega_\ell) \\ \dot{\omega}_\ell = u_\ell \quad \dot{\omega}_r = u_r, \tag{29}$$

where $r = 0.1$ m is the wheel radius, $L = 0.24$ is the body length (body width is 0.22 m), (x, y) is the position, θ is the orientation, ω_ℓ, $\omega_r(|\omega_\ell|, |\omega_r| \leq \pi/2\,\text{rad/s})$ are the rotational velocities of the left and right wheels; and u_ℓ, $u_r(|u_\ell|, |u_r| \leq 2\,\text{rad/s}^2)$ are the inputs for controlling the rate of change of the rotational velocities for the left and right wheels, respectively.

The equations of motions for the Ackerman steered vehicle are defined as

$$\dot{x} = v\cos(\theta) \quad \dot{y} = v\sin(\theta) \quad \dot{\theta} = v\tan(\psi)/L \quad \dot{v} = u_a \quad \dot{\psi} = u_\omega, \tag{30}$$

where $L = 0.24$ m is the body length (body width is 0.22 m), (x, y) is the position, θ is the orientation, $v(|v| \leq 5\ \text{m/s}^2)$ is the velocity, $\psi(|\psi| \leq \pi/3.6\,\text{rad})$ is the steering angle, $u_a(|u_a| \leq 2\ \text{m/s}^2)$ is the input control for the acceleration, and $u_\omega(|u_\omega| \leq \pi/3\,\text{rad/s})$ is the input control for the rotational velocity of the steering angle.

Table IV provides a summary of the results. As with the other experiments, `Argos` is shown to be significantly faster than `LTLSyclop`. The `RRT`-based approach failed to solve these problem instances. These results indicate that `Argos` works well with a wide variety of vehicle models.

6. Discussion

This paper presented an effective approach, `Argos`, for planning collision-free and dynamically feasible motion trajectories that enable a mobile robot to satisfy task specifications given as regular

https://doi.org/10.1017/S0263574715000417 Published online by Cambridge University Press

languages over regions of interest. The efficiency of `Argos` was derived from (i) partition of the sampling-based motion tree into equivalence classes; (ii) heuristic costs based on short abstract paths to estimate the feasibility of reaching an accepting automaton state from each equivalence class; and (iii) partition refinement in order to promote rapid expansion while discovering new ways to reach accepting automaton states. Comparisons to related work showed significant computational speedups of one order of magnitude.

A direction for future work is to use more advanced AI techniques to improve the discrete search, even exploiting previous invocations to more efficiently compute abstract paths. Another direction is to incorporate machine learning in order to automatically adjust the parameters based on the progress made during the tree expansion. We will also seek to apply `Argos` to inspection tasks involving ground, aerial, and underwater vehicles as well as extend `Argos` to multiple robots working together in order to more efficiently complete the assigned task.

Acknowledgments

This work is supported by NSF IIS-1449505 and NSF ACI-1440587. The work of McMahon is also supported by the Office of Naval Research, code 32.

References

1. A. P. Aguiar and J. P. Hespanha, "Trajectory-tracking and path-following of underactuated autonomous vehicles with parametric modeling uncertainty," *IEEE Trans. Autom. Control* **52**(8), 1362–1379 (2007).
2. C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins and G. J. Pappas, "Symbolic planning and control of robot motion," *IEEE Robot. Autom. Mag.* **14**(1), 61–71 (2007).
3. M. de Berg, O. Cheong, M. van Kreveld and M. H. Overmars, *Computational Geometry: Algorithms and Applications*, 3 ed. (Springer-Verlag, Berlin Heidelberg, 2008).
4. A. Bhatia, L. E. Kavraki and M. Y. Vardi, "Motion Planning with Hybrid Dynamics and Temporal Goals," *IEEE Conference on Decision and Control*, Atlanta, Georgia, USA (2010) pp. 1108–1115.
5. A. Bhatia, L. E. Kavraki and M. Y. Vardi, "Sampling-Based Motion Planning with Temporal Goals," *IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, USA (2010) pp. 2689–2696.
6. A. Bhatia, M. Maly, L. Kavraki and M. Vardi, "Motion planning with complex goals," *IEEE Robot. Autom. Mag.* **18**, 55–64 (2011).
7. E. F. Camacho and C. B. Alba, *Model Predictive Control* (Springer, London, 2013).
8. Y. Chen, D. Ding, A. Stefanescu and C. Belta, "A formal approach to the deployment of distributed robotic teams," *IEEE Trans. Robot.* **28**(1), 158–171 (2012).
9. P. Cheng, E. Frazzoli and S. LaValle, "Improving the performance of sampling-based motion planning with symmetry-based gap reduction," *IEEE Trans. Robot.* **24**(2), 488–494 (2008).
10. P. Cheng, G. Pappas and V. Kumar, "Decidability of Motion Planning with Differential Constraints," *IEEE International Conference on Robotics and Automation*, Rome, Italy (2007) pp. 1826–1831.
11. H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations* (MIT Press, 2005).
12. O. Chuy, E. Collins, D. Dunlap and A. Sharma, "Sampling-based direct trajectory generation using the minimum time cost function," *Experimental Robotics, Springer Tracts in Advanced Robotics* **88**, 651–666 (2013).
13. E. Coumans, "Bullet physics engine," (2012). Http://bulletphysics.org/; [accessed on 06/04/2015].
14. I. A. Şucan and L. E. Kavraki, "A sampling-based tree planner for systems with complex dynamics," *IEEE Trans. Robot.* **28**(1), 116–131 (2012).
15. S. Dalibard and J. P. Laumond, "Control of probabilistic diffusion in motion planning," *International Workshop on Algorithmic Foundations of Robotics*, *Springer Tracts in Advanced Robotics*, vol. 57 (2009) pp. 467–481.
16. N. T. Dantam and M. Stilman, "The motion grammar: Analysis of a linguistic method for robot control," *IEEE Trans. Robot.* **29**(3), 704–718 (2013).
17. J. A. DeCastro and H. Kress-Gazit, "Guaranteeing Reactive High-Level Behaviors for Robots with Complex Dynamics," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan (2013) pp. 749–756.
18. D. Devaurs, T. Simeon and J. Cortés, "Enhancing the Transition-Based RRT to Deal with Complex Cost Spaces," *IEEE International Conference on Robotics and Automation*, Tokyo, Japan (2013) pp. 4120–4125.
19. X. C. Ding, M. Lazar and C. Belta, "LTL receding horizon control for finite deterministic systems," *Automatica* **50**(2), 399–408 (2014).
20. D. D. Dunlap, C. V. Caldwell and E. G. Collins Jr, "Nonlinear Model Predictive Control using Sampling and Goal-Directed Optimization," *IEEE International Conference on Control Applications*, Yokohama, Japan (2010) pp. 1349–1356.
21. G. E. Fainekos, A. Girard, H. Kress-Gazit and G. J. Pappas, "Temporal logic motion planning for dynamic mobile robots," *Automatica* **45**(2), 343–352 (2009).

22. I. Filippidis, D. V. Dimarogonas and K. J. Kyriakopoulos, "Decentralized Multi-Agent Control from Local LTL Specifications," *IEEE Conference on Decision and Control*, Maui, Hawaii, USA (2012) pp. 6235–6240.
23. E. Frazzoli, M. A. Dahleh and E. Feron, "Trajectory Tracking Control Design for Autonomous Helicopters using a Backstepping Algorithm," *American Control Conference*, vol. 6, Chicago, Illinois, USA (2000) pp. 4102–4107.
24. E. Frazzoli, M. A. Dahleh and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Trans. Robot.* **21**(6), 1077–1091 (2005).
25. B. Gipson, M. Moll and L.E. Kavraki, "Resolution Independent Density Estimation for Motion Planning in High-Dimensional Spaces," *IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany (2013) pp. 2437–2443.
26. D. Hsu, R. Kindel, J. C. Latombe and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Int. J. Robot. Res.* **21**(3), 233–255 (2002).
27. K. Kanjanawanishkul and A. Zell, "Path Following for an Omnidirectional Mobile Robot Based on Model Predictive Control," *IEEE International Conference on Robotics and Automation*, Kobe, Japan (2009) pp. 3341–3346.
28. S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.* **30**(7), 846–894 (2011)
29. S. Karaman and E. Frazzoli, "Sampling-Based Algorithms for Optimal Motion Planning with Deterministic μ-Calculus Specifications," *American Control Conference*, Montreal, Canada (2012) pp. 735–742.
30. Keller, H., *Numerical Methods for Two-Point Boundary-Value Problems* (Dover, New York, NY, 1992).
31. M. Kloetzer and C. Belta, "Temporal logic planning and control of robotic swarms by hierarchical abstractions," *IEEE Trans. Robot.* **23**(2), 320–331 (2007).
32. H. Kress-Gazit, D. C. Conner, H. Choset, A. Rizzi and G. J. Pappas, "Courteous cars: Decentralized multi-agent traffic coordination," *IEEE Robot. Autom. Mag.* **15**(1), 30–38 (2008).
33. H. Kress-Gazit, G. E. Fainekos and G. J. Pappas, "Temporal logic-based reactive mission and motion planning," *IEEE Trans. Robot.* **25**(6), 1370–1381 (2009).
34. H. Kress-Gazit, T. Wongpiromsarn and U. Topcu, "Correct, reactive robot control from abstraction and temporal logic specifications," *IEEE Robot. Autom. Mag.* **18**(3), 65–74 (2011).
35. O. Kupferman and M. Vardi, "Model checking of safety properties," *Formal Methods Syst. Des.* **19**(3), 291–314 (2001).
36. Latvala, T., "Efficient Model Checking of Safety Properties," *Model Checking Software*, Lecture Notes in Computer Science, vol. 2648 (2003) pp. 74–88.
37. S. M. LaValle, *Planning Algorithms* (Cambridge University Press, Cambridge, MA, 2006).
38. S. M. LaValle, "Motion planning: The essentials," *IEEE Robot. Autom. Mag.* **18**(1), 79–89 (2011).
39. S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robot. Res.* **20**(5), 378–400 (2001).
40. J. McMahon and E. Plaku, "Sampling-Based Tree Search with Discrete Abstractions for Motion Planning with Dynamics and Temporal Logic," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, Illinois, USA (2014) pp. 3726–3733.
41. C. Ordonez, N. Gupta, W. Yu, O. Chuy and E. G. Collins, "Modeling of Skid-Steered Wheeled Robotic Vehicles on Sloped Terrains," *ASME Dynamic Systems and Control Conference*, Fort Lauderdale, Florida, USA (2012) pp. 91–99.
42. E. Plaku, "Robot Motion Planning with Dynamics as Hybrid Search," *AAAI Conference on Artificial Intelligence*, Bellevue, Washington, USA (2013) pp. 1415–1421.
43. E. Plaku, L. Kavraki and M. Vardi, "Falsification of LTL safety properties in hybrid systems," *Int. J. Softw. Tools Technol. Transfer* **15**(4), 305–320 (2013).
44. E. Plaku, L. E. Kavraki and M. Y. Vardi, "Falsification of LTL Safety Properties in Hybrid Systems," *Lecture Notes in Computer Science*, vol. 5505 (2009) pp. 368–382.
45. E. Plaku, L. E. Kavraki and M. Y. Vardi, "Motion planning with dynamics by a synergistic combination of layers of planning," *IEEE Trans. Robot.* **26**(3), 469–482 (2010).
46. R. Rajamani, *Vehicle Dynamics and Control* (Springer, New York, 2011).
47. V. Raman and H. Kress-Gazit, "Explaining impossible high-level robot behaviors," *IEEE Trans. Robot.* **29**(1), 94–104 (2013).
48. S. Rodriguez, X. Tang, J. M. Lien and N. M. Amato, "An Obstacle-Based Rapidly-Exploring Random Tree," *IEEE International Conference on Robotics and Automation*, Orlando, Florida, USA (2006) pp. 895–900.
49. G. Sánchez and J. C. Latombe, "On delaying collision checking in PRM planning: Application to multi-robot coordination," *Int. J. Robot. Res.* **21**(1), 5–26 (2002).
50. J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Comput. Geom.: Theory Appl.* **22**(1-3), 21–74 (2002).
51. A. Sistla, "Safety, liveness and fairness in temporal logic," *Form. Asp. Comput.* **6**, 495–511 (1994).
52. M. Smith, "Open dynamics engine," (2006). www.ode.org; [accessed on 06/04/2015].
53. A. Ulusoy, T. Wongpiromsarn and C. Belta, "Incremental controller synthesis in probabilistic environments with temporal logic constraints," *Int. J. Robot. Res.* **33**(8), 1130–1144 (2014).
54. C. I. Vasile and C. Belta, "Sampling-Based Temporal Logic Path Planning," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan (2013) pp. 4817–4822.
55. C. I. Vasile and C. Belta, "Reactive Sampling-Based Temporal Logic Path Planning," *IEEE International Conference on Robotics and Automation*, Hong Kong, China (2014) pp. 4310–4315.