Mission and Motion Planning for Autonomous Underwater Vehicles Operating in Spatially and Temporally Complex Environments

James McMahon and Erion Plaku

Abstract

This paper seeks to enhance the autonomy of underwater vehicles. The proposed approach takes as input a mission specified via a regular language and automatically plans a collision-free, dynamically-feasible, and low-cost trajectory which enables the vehicle to accomplish the mission. Regular languages provide a convenient mathematical model that frees users from the burden of unnatural low-level commands and instead allows them to describe missions at a high level in terms of desired objectives. To account for the constraints imposed by the mission, vehicle dynamics, collision avoidance, and the complex spatial and temporal variability of the ocean environment, the approach tightly couples mission planning with sampling-based motion planning. A key aspect is a discrete abstraction obtained by combining the finite automaton representing the regular language with a navigation roadmap constructed by probabilistic sampling. The approach searches the discrete abstraction in order to compute low-cost and collision-free navigation routes that are compatible with the mission. Sampling-based motion planning is then used to expand a tree of dynamically-feasible trajectories along the navigation routes. The approach is validated both in simulation and field experiments. Results demonstrate the efficiency and the scalability of the approach and show significant improvements over related work.

Index Terms

AUV, sampling-based motion planning, mission planning, regular languages, robotics

I. INTRODUCTION

Autonomous underwater vehicles (AUVs) are a critical component of oceanic research and development. With applications relevant to industry [1], defense [2], and environmental research [3], AUVs are required to autonomously perform increasingly sophisticated missions. In order to perform such missions, an AUV must contain a mission planner that can reason about both high-level tasks and low-level actions. While there exist planners that separately consider both the high-level and low-level aspects, due to the dynamics...
of the vehicle and the interactions between the environment and the AUV, there is a growing need to reason about both simultaneously. This need is evident when considering multiple tasks over large bodies of water across long time durations. For example, while an AUV executes a series of tasks that occur over a time span of several hours, the environment could change dramatically with respect to tide and currents. If the intertwined dependencies between the tasks and AUV dynamics are not considered in the initial planning phases then interactions with the environment could be overlooked. Not properly considering the dynamically changing environment would then result in wasting valuable time or missing data-collecting opportunities while the AUV attempts to execute tasks that are no longer feasible.

Furthermore, a mission planner should also reduce the cognitive load of the operator so that errors in planning are reduced. This requires a user interface that reduces the input into a series of high-level tasks as opposed to specifying numerous specific low-level motions or actions. This can be done by abstracting mission elements through an easy to use structured language. This paper makes it possible to consider missions specified as regular languages and automatically plan the vehicle motions to accomplish the mission. As an illustration, the mission of reaching regions $R_1, R_2, \ldots, R_n$ one after the other can be specified via the regular expression

$$\phi_{\text{seq}}(\pi_1, \pi_2, \ldots, \pi_n) = \pi_1 \pi_2 \ldots \pi_n,$$

where $\pi_i$ denotes the proposition “AUV reached region $R_i$.” As another example, in a data acquisition mission, the AUV has the flexibility to visit the regions in any order – such mission can be expressed as

$$\phi_{\text{cov}}(\pi_1, \pi_2, \ldots, \pi_n) = \bigvee_{(i_1, \ldots, i_n) \in \text{perm}(1, n)} \pi_{i_1} \pi_{i_2} \ldots \pi_{i_n},$$

where $\text{perm}(1, n)$ denotes all the permutations of $\{1, \ldots, n\}$. It is also possible to impose a partial order where a group of regions has to be visited before another group, e.g., the mission “reach $R_1, \ldots, R_{\lceil \frac{n}{2} \rceil}$ in any order and then reach $R_{\lceil \frac{n}{2} \rceil+1}, \ldots, R_n$ in any order” can be expressed as

$$\phi_{\text{po}}(\pi_1, \pi_2, \ldots, \pi_n) = \phi_{\text{cov}}(\pi_1, \ldots, \pi_{\lceil \frac{n}{2} \rceil}) \phi_{\text{cov}}(\pi_{\lceil \frac{n}{2} \rceil+1}, \ldots, \pi_n).$$

This can be generalized to several groups where the AUV is required to visit each region in a group before proceeding to the next group, i.e.,

$$\phi_{\text{groups}}(\pi_1, \pi_2, \ldots, \pi_n) = \phi_{\text{cov}}(G_1) \phi_{\text{cov}}(G_2) \ldots \phi_{\text{cov}}(G_k).$$
where $G_1, G_2, \ldots, G_k$ denotes a partition of $\{\pi_1, \ldots, \pi_n\}$ into $k$ groups. Regular languages could also be used to express missions where the AUV has to alternate between different groups. As an example, the mission “alternate reaching a region from $\{R_1, \ldots, R_{\lceil \frac{n}{2} \rceil}\}$ and $\{R_{\lceil \frac{n}{2} \rceil + 1}, \ldots, R_n\}$ until all regions have been reached” can be expressed as

$$\phi_{\text{alternate}}(\pi_1, \pi_2, \ldots, \pi_n) = \bigvee_{(i_1, i_3, \ldots, i_{2\lceil \frac{n}{2} \rceil - 1}) \in \text{perm}(1, \lceil \frac{n}{2} \rceil)} \pi_{i_1} \pi_{i_2} \ldots \pi_{i_n}.$$  

(5)

These examples illustrate the convenience and expressiveness of using regular languages to specify AUV missions. Regular languages have several other properties. First, any mission that requires the AUV to visit a finite number of regions can be expressed as a regular language. This follows from the fact that any finite language is a regular language. Second, regular languages are closed\(^1\) under union, intersection, concatenation, complementation, reversal, quotient, and numerous other operations. This makes it possible to construct hierarchical missions by combining regular languages via closure operations. Third, regular languages can be expressed via regular expressions, regular grammars, deterministic finite automata (DFAs), nondeterministic finite automata (NFAs), alternating finite automata, and several other models of computation. This allows the user to choose a convenient model to express a particular mission. Fourth, regular languages include a special class of Linear Temporal Logic (LTL) referred to as syntactically co-safe LTL [4]. LTL has been widely used in model checking and in planning to specify desired properties by combining propositions with logical (and, or, not) and temporal (always, eventually, next, until) operators. A syntactically co-safe LTL formula is an LTL formula that does not use the “always” operator when written in positive normal form. Syntactically co-safe LTL formulas are characterized by having finite good prefixes which form a regular language. Using tools from model checking, a syntactically co-safe LTL formula can be automatically converted into a DFA [5], [6]. This conversion gives the user the flexibility to also use syntactically co-safe LTL formulas, when convenient, to express the mission.

Planning collision-free and dynamically-feasible trajectories that satisfy missions specified by regular languages introduces unique challenges. The planner must be able to operate both in the discrete domain dominated by the structure of the regular language and in the continuous domain dominated by the nonlinear dynamics, collision avoidance, physical constraints, and drift caused by the time-varying ocean currents. The intertwined dependencies between the high-level mission requirements and the feasible

\(^1\)Regular languages are closed under an operation $f$ when, for any regular language $\mathcal{L}_1, \ldots, \mathcal{L}_n$, the result $f(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is also regular.
Motions require a tight coupling between mission and motion planning. In fact, not every word \( \sigma = \langle \pi_{i_1}, \ldots, \pi_{i_L} \rangle \) that satisfies the mission specification given by some regular language \( \mathcal{L} \) is necessarily feasible. Due to constraints imposed by the vehicle dynamics, collision avoidance, and ocean currents, it may be difficult or impossible to reach the regions of interest in the order specified by \( \sigma \). As a result, one of the main challenges is determining which \( \sigma \) is feasible among the words in \( \mathcal{L} \). Even when \( \mathcal{L} \) is finite, the number of words could increase exponentially fast with respect to the number of propositions, as it is the case for the coverage, partial order, grouping, and alternating missions shown in Eqn. 2–5.

To address these challenges, this paper proposes an approach, termed PUMMA (Planner for Underwater Motions and Missions given by an Automaton), which tightly couples mission planning with sampling-based motion planning. The approach leverages from sampling-based motion planning the idea of selectively sampling and exploring the space of feasible motions. In particular, starting from the initial state, a motion tree is incrementally expanded by adding collision-free and dynamically-feasible trajectories as tree branches. A navigation roadmap is introduced in order to effectively guide the motion-tree expansion. Drawing from the Probabilistic RoadMap (PRM) approach [7], the navigation roadmap is obtained by sampling waypoints over the operational area and connecting neighboring waypoints in order to construct a network of navigation routes that avoid known obstacles, areas that are deemed too dangerous for the AUV, or other forbidden regions. The navigation roadmap is combined with the DFA representing the regular language in order to compute sequences of waypoints that are compatible with the mission specification. In particular, the motion tree is partitioned into equivalence classes based on a mapping from motion-tree trajectories to DFA states and roadmap waypoints. Heuristic costs based on shortest-path distances over the roadmap and the DFA are used to estimate the feasibility of each equivalence class. Equivalence classes that are deemed feasible are expanded along navigation routes that seek to reduce the time it takes the AUV to accomplish the mission. The approach is validated both in simulation and field experiments with an OceanServer Iver2 [8] vehicle. Results demonstrate its efficiency and scalability and show significant improvements over related work.

A preliminary version of this work appeared as conference proceedings [9], [10]. The work in [9] provided a proof-of-concept on the benefits of using a roadmap to guide the motion-tree expansions, while the work in [10] showed the advantages of partitioning the motion-tree into equivalence classes. The work in [9] used a simplified environment for AUVs, while the work in [10] was only for ground vehicles. In both cases, syntactically co-safe LTL was used for specifying the tasks. The approach presented here offers
several algorithmic improvements over the preliminary versions and extended experimental evaluations. In particular, it introduces a partition of the motion tree into equivalence classes based on a combination of the roadmap with the automaton representing the regular language. The roadmap construction is improved by using the meshes for the forbidden regions and the bathymetry map to generate short and safe navigation routes. The discrete search is enhanced by incorporating heuristic costs based on shortest-path distances to accepting automaton states. By following the navigation routes, the motion-tree is expanded more effectively. The proposed approach also incorporates predictive models of ocean currents. The experiments in the preliminary versions were conducted only in simulation and over small areas. The experiments in this paper are conducted with realistic AUV models operating in large areas obtained from real environments. In addition, this paper includes field experiments with an OceanServer Iver2 vehicle.

The rest of the article is organized as follows. Section II discusses related work. Section III formally defines the problem. The proposed approach is described in Section IV followed by experimental results in Section V. The paper concludes in Section VI with a discussion and directions for future research.

II. RELATED WORK

The use of planning algorithms for AUVs has been investigated for both high-level mission planning and low-level motion planning. The discussion below focuses on these two categories and highlights the contributions offered by the proposed approach.

A. AUV Motion Planning

Related work on path planning for AUVs often imposes a discretization over the operational area and uses A* to find an optimal path over the discretization. The discretization is obtained by imposing a regular grid or building a quadtree to represent the collision-free areas [11]. Since the discretization does not take the AUV dynamics into account, the planned paths are not necessarily dynamically feasible. To account for the ocean currents, the work in [12] introduces a heuristic cost function which estimates the time the AUV would need to travel from one grid point to the next. The work in [13] accounts for the ocean currents and generates optimal paths over a 2D discretization by combining A* and Fast Marching. Risk-aware path planners have also been developed which minimize the risk of collisions by taking into account the uncertainty associated with the prediction of the ocean currents and the AUV navigation [14].

Another class of approaches relies on sampling-based motion planning. A variant of the Rapidly-exploring Random Tree (RRT) algorithm is used in [15] to plan collision-free paths. During planning,
a linearized model of the vehicle dynamics is taken into account. RRT is also used in [16] to plan the motions of a glider. The work in [17] uses the notion of homotopy classes to guide the RRT expansion. Other work has used 3D plane mapping [18], predictive control [19], nonlinear trajectory generation [20], genetic algorithms [21], sliding wavefront expansions [22], cost function minimization [23], optimization over particle swarms [24], mixed integer linear programming [25], and recursive greedy algorithms [26]. The contribution of this paper over the related work is the ability of the proposed approach to take into account the vehicle dynamics, ocean currents, and missions specified by regular languages.

B. AUV Mission Planning

In more advanced systems, the tasks the AUV is required to complete are also considered during the planning phases. High-level AUV planning has focused on enhancing the autonomy of control architectures by introducing deliberative planners to reason about tasks. Traditional graph-search algorithms [27] are used to search the discretization and generate approximate plans quickly in order to react to variations observed in the environment. Multi-objective optimization [28] is used to generate low-cost paths that can cover specific regions while adapting to in-situ measurements [29]. Sampling-based motion planning is used for inspecting of a ship hull [30]. Plan repair using partial order planning is used to react to changes in the environment [31] and temporal constraint-based planning is used to robustly deliberate about future plans and actions [32]. Further work has been to apply context-sensitive reasoning [33], self organizing maps [34], genetic algorithms to solve a traveling-salesmen problem using cost defined as energy used [35], and goal driven autonomy [36]. The proposed approach, in distinction from these methods, takes into account during planning not only the high-level mission specification but also the vehicle dynamics, the effect of the ocean currents, and forbidden regions in order to plan collision-free and dynamically-feasible trajectories that enable the AUV to accomplish the mission.

The planning domain definition language (PDDL) is used in [37] to model the inspection task of an underwater structure. The PDDL planner relies on a probabilistic roadmap to provide a connected network of waypoints which is then searched in order to compute a sequence of waypoints that enables the AUV to observe every part of the underwater structure. The planned path is not necessarily dynamically feasible since neither the vehicle dynamics nor the ocean currents are taken into account during the roadmap construction. In fact, roadmaps are generally not appropriate for incorporating dynamics since each roadmap edge requires exact solutions to the two-point boundary value problem in order to steer the vehicle from one endpoint to the other. Due to the nonlinearity of the vehicle dynamics, exact solutions
to the two-point boundary value problem are generally not available while numerical solutions render the roadmap construction impractical due to the significant computational cost [38], [39]. In contrast, PUMMA uses the roadmap only as a guide to obtain navigation routes along which to expand a motion tree. Each branch in the motion tree is obtained by applying input controls, taking into account the vehicle dynamics and the effect of the ocean currents. Moreover, the proposed approach can be used for any mission given by some regular language over regions of interest as opposed to the work in [37] which is for a specific PDDL description corresponding to an inspection task.

Model checking in combination with sampling-based motion planning was used in the LTLSyclop framework in order to account for tasks given by syntactically co-safe LTL formulas [40]–[43]. LTLSyclop was developed for ground vehicles and was not applied to AUVs in these papers. LTLSyclop suffers from scalability issues since at each iteration it performs a computationally-expensive discrete search to determine a sequence of cells in a grid or a triangular decomposition of the environment that is compatible with the LTL specification. The discrete search uses edge costs that depend on the free volume and coverage of each cell by the motion tree. Afterwards, LTLSyclop tries to expand the motion tree along the cells in the sequence. This is an iterative process where at each iteration LTLSyclop selects a cell \( c_i \) from the sequence and tries to expand the motion tree to reach the next cell, \( c_{i+1} \), in the sequence. More specifically, a random point \( p \) is sampled uniformly at random inside \( c_{i+1} \) and the motion tree is expanded from the closest tree vertex by applying control inputs that seek to steer the vehicle toward \( p \). As a result, LTLSyclop often spends considerable computational time before realizing that the current sequence of cells is difficult or even impossible to follow due to constraints imposed by the vehicle dynamics, ocean currents, and forbidden regions. In contrast, the proposed approach introduces a partition of the motion tree into equivalence classes and promotes expansions from those equivalences classes that are deemed to be close to completing the overall mission. Moreover, the proposed approach does not rely on a grid or a triangular decomposition but instead constructs a roadmap to obtain safe navigation routes along which to expand the motion tree. The proposed approach allows for missions given by regular languages which includes the class of syntactically co-safe LTL formulas. Comparisons to LTLSyclop show that the proposed approach offers significant improvements both in terms of the runtime and solution costs.

Rapidly-exploring Random Graph (RRG) [44] was developed as a variant of RRT to accomodate specifications given by deterministic \( \mu \)-calculus, which is a superset of LTL. RRG created dense graphs which considerably increased its running time. By using the notion of near and far neighbors, Sparse RRG
(SRRG) created a sparse graph with cycles which is used to find paths that satisfy not just syntactically co-
safe but instead any LTL formula [45]. SRRG was used to find paths for a point robot with no dynamics.
As an RRT variant, however, SRRG lacks global guidance and as a result spends significant time exploring
the state space. Comparisons show that the proposed approach is significantly faster than SRRG.

III. Preliminaries

This section describes the AUV simulator, the syntax and semantics of the regular languages used for
specifying missions, and the problem statement.

A. AUV Simulator

This paper uses the MOOS-IvP framework [46] in order to obtain a 3D AUV simulator that models
the vehicle dynamics even when considering the external drift forces caused by the ocean currents. The
simulated vehicle corresponds to an OceanServer Iver2 [8], which is also the vehicle used in the field
experiments. The vehicle has a diameter of 14.7 cm, length of 127 cm, and weight of 19 kg.

The MOOS-IvP simulator maintains the vehicle state in terms of its position, depth, orientation, velocity,
acceleration, and buoyancy rate. The state space, denoted by \( S \subset \mathbb{R}^{\text{dim}(S)} \), defines the range of values for
the state variables, where \( \text{dim}(S) \in \mathbb{N} \) denotes the number of the state variables. For convenience, the
notations \( \text{POS}(s) \) and \( \text{DEPTH}(s) \) are used to denote the position and depth associated with a state \( s \in S \) with
the configuration defined as \( \text{CFG}(s) = (\text{POS}(s), \text{DEPTH}(s)) \). The control space, denoted by \( U \subset \mathbb{R}^{\text{dim}(U)} \),
defines the variables that are used to control the vehicle. For the OceanServer Iver2 model, the control
\( u \in U \) specifies the desired heading, depth, and speed. MOOS-IvP does not place any restrictions on
the model used for the time-varying ocean currents. It simply requires an external function of the form
\( \text{DRIFT}(x, y, d, t) \) which specifies the drift forces at position \((x, y)\), depth \(d\), and time \(t\). As discussed in
Section V, this paper uses data from the Chesapeake Bay Operational Forecast System for modeling the
time-varying ocean currents in the areas where the experiments are conducted.

The new state is obtained by propagating the vehicle dynamics based on the current state, control, and
ocean drift forces. More specifically, MOOS-IvP provides a function

\[
s_{\text{new}} \leftarrow \text{SIMULATOR}(s, u, f, dt),
\]

which computes the new state \( s_{\text{new}} \in S \) by simulating the vehicle dynamics for a small time step \( dt \)
when the current state is \( s \in S \), the input control is \( u \in U \), and the drift is \( f \in \mathbb{R}^3 \). As it is common
in sampling-based motion planning, the time step, $dt$, is set to a small value to increase the accuracy of the simulator and not to leave large gaps between consecutive states. Internally, MOOS-IvP uses PID controllers [47] to determine appropriate actuator values that would steer the vehicle along the heading, depth, and speed specified by $u$. The results are in the form of rudder, thrust, and elevator. The actuator values are then used to propagate the current state according to the vehicle dynamics. The position is further updated by applying the drift forces to the vehicle’s center of gravity. The MOOS-IvP simulator has been tested in many applications and shown to accurately model the vehicle dynamics [46].

This work allows for the specification of forbidden regions that the AUV should always avoid. The forbidden regions could correspond to known obstacles, areas that are too dangerous for the AUV, or the ocean floor. A function $\text{COLLISION} : S \rightarrow \{\text{true, false}\}$ is provided as input where $\text{COLLISION}(s)$ determines whether or not the state $s$ places the vehicle in a forbidden region. For example, when the forbidden regions are specified as polyhedra or triangular meshes, as it is the case for the experiments in this paper, collision-detection packages such as PQP [48] can be used for the implementation of $\text{COLLISION}$. In addition, $\text{COLLISION}(s)$ checks whether or not $\text{DEPTH}(s)$ is below the maximum depth at $\text{POS}(s)$ (if so, the vehicle would be colliding with the ocean floor). A bathymetry map is used to provide depth information over the area where the experiments are conducted. The bathymetry map is often specified as a grid over the $xy$-plane with the maximum depth information provided for each cell in the grid. A triangulated surface is automatically constructed from the bathymetry map using standard heightfield-to-mesh construction algorithms available in the Blender software [49]. The triangulated surface is used by $\text{COLLISION}(s)$ to determine whether or not the vehicle violates the maximum depth condition when placed according to the position, depth, and orientation specified by $s$.

In summary, let $\mathcal{O}_{\text{forbidden}} = \{O_1, \ldots, O_m\}$ denote the polyhedra or triangular meshes associated with the forbidden regions. Let $\mathcal{O}_{\text{DepthSurf}}$ denote the triangular mesh constructed from the bathymetry map. The function $\text{COLLISION}(s)$ returns true if and only if the vehicle collides with one of the meshes $O_1, \ldots, O_m, \mathcal{O}_{\text{DepthSurf}}$ when placed at the position, depth, and orientation specified by the state $s \in S$.

B. Mission Specifications

Missions are specified as regular languages over propositions corresponding to goal regions. Let $\mathcal{W}$ denote the 3D bounding box of the operational area of the AUV. Let $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, $\mathcal{R}_i \subseteq \mathcal{W}$, denote the goal regions, which are assumed to be disjoint, i.e., $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ for all $1 \leq i < j \leq n$. Each $\mathcal{R}_i$ is labeled with a proposition $\pi_i$, which is said to be true only when the AUV is inside $\mathcal{R}_i$. The set of
propositions, defined as \( \Pi = \{ \pi_1, \ldots, \pi_n \} \), corresponds to the alphabet of the regular language. Examples of sequencing, coverage, partial ordering, grouping, and alternating missions are shown in Eqn. 1–5.

Internally, the planner uses a DFA to represent the mission. When the user specifies the mission via a regular expression, standard algorithms from automata theory are used to automatically convert the regular expression into a minimal DFA [50]. Fig. 1 shows DFAs for specific instances of the missions described in Eqn. 1–5. The formal definition of a DFA 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, \( \Pi \) is the input alphabet, \( \delta : Z \times \Pi \rightarrow Z \) is the transition function, \( z_{\text{init}} \in Z \) is the initial state, and \( \text{ACCEPT} \subseteq Z \) is the set of accepting states. The extended transition function \( \hat{\delta} : Z \times \Pi^* \rightarrow Z \) is defined as

\[
\forall z \in Z, \sigma \in \Pi^*, \pi \in \Pi:\quad \hat{\delta}(z, \sigma) = \delta(z, \sigma) = \delta(z, \sigma \pi),
\]

where \( \sigma \) denotes the empty string and \( \Pi^* \) denotes all the strings that can be formed by concatenating zero or more symbols from \( \Pi \). The DFA \( \mathcal{A} \) accepts the input string \( \sigma \in \Pi^* \) if and only if \( \hat{\delta}(z_{\text{init}}, \sigma) \in \text{ACCEPT} \).

![Fig. 1. Examples of DFAs for instances of the missions described in Eqn. 1–5. The superscript in \( \phi \) denotes the number of propositions. In the case of \( \phi_{\text{groups}} \), the groups correspond to \( G_1 = \{ \pi_1, \pi_2, \pi_3 \} \), \( G_2 = \{ \pi_4, \pi_5 \} \), and \( G_3 = \{ \pi_6, \pi_7 \} \). To simplify the figures, if \( \hat{\delta}(z, \pi) \) is not shown, it means that \( \delta_A(z, \pi) = z_{\text{rej}} \), where \( z_{\text{rej}} \) is a reject state, i.e., \( \delta_A(z_{\text{rej}}, \pi) = z_{\text{rej}} \) for all \( \pi \in \Pi \).](image)

C. Problem Statement

A motion trajectory is obtained by applying a sequence of controls in succession. Let \( t \) denote the start time, and suppose that, starting from the state \( s \in S \), the control inputs \( \langle u_0, u_1, \ldots, u_{\ell-1} \rangle \), \( u_i \in U \), are applied in succession for one time step \( dt \). This gives rise to a trajectory \( \zeta : \{0, 1, \ldots, \ell\} \rightarrow S \) where

\[
\zeta(0) = s \text{ and } \forall i \in \{0, \ldots, \ell - 1\} : \zeta(i + 1) = \text{SIMULATE}(\zeta(i), u_i, f_i, dt)
\]

with \( f_i = \text{DRIFT(POS(\zeta(i))), DEPTH(\zeta(i)), t + i \ast dt} \) corresponding to the drift forces.
Consider the trajectory $\zeta : \{0, 1, \ldots, \ell\} \to S$. Recall that $\zeta(i)$ is said to reach the goal region $R_j$ if \( \text{CFG}(\zeta(i)) \in R_j \). The trajectory $\zeta$ could reach zero or more regions from $R$. Let $R_{j_1}, R_{j_2}, \ldots, R_{j_k}$ denote the sequence of regions in the order reached by $\zeta$, where $R_{j_i} \neq R_{j_{i+1}}$ for every $1 \leq i < k$. Using the propositional labels associated with the regions in $R$, $\text{WORD}(\zeta)$ is then defined as $\langle \pi_{j_1}, \pi_{j_2}, \ldots, \pi_{j_k} \rangle \in \Pi^*$. Fig. 2 shows an example. As a result, $\zeta$ satisfies the mission specification if and only if an accepting automaton state is reached when running $A$ with $\text{WORD}(\zeta)$ as the input, i.e.,

$$
\hat{\delta}_A(z_{\text{init}}, \text{WORD}(\zeta)) \in \text{ACCEPT}_A.
$$

Fig. 2. Illustration of a trajectory $\zeta$ satisfying $\phi_{\text{alternate}}(\pi_1, \pi_2, \pi_3, \pi_4)$. Note that $\text{WORD}(\zeta) = \langle \pi_1, \pi_3, \pi_2, \pi_4 \rangle$ since the sequence of the goal regions reached by $\zeta$ is $R_1, R_3, R_2, R_4$. Forbidden regions are shown in gray.

The problem studied in this paper is stated below.

**Problem 1:** Given

- an operational area $\mathcal{W}$,
- goal regions $R = \{R_1, \ldots, R_n\}$, where $R_i \subseteq \mathcal{W}$ is labeled with proposition $\pi_i$,
- a mission specification as a DFA $A = \langle Z, \Pi, z_{\text{init}}, \delta, \text{ACCEPT} \rangle$ over propositions $\Pi = \{\pi_1, \ldots, \pi_n\}$,
- polyhedra or triangular meshes for the forbidden regions $O_{\text{forbidden}} = \{O_1, \ldots, O_m\}$, a triangular mesh $O_{\text{DepthSurf}}$ constructed from the bathymetry map, and a function $\text{COLLISION} : S \to \{\text{true, false}\}$
  to check for collisions with the forbidden regions and the ocean floor,
- a model of the time-varying ocean currents as a function of the form $\text{DRIFT}(x, y, d, t)$,
- a simulator of the form $s_{\text{new}} \leftarrow \text{SIMULATOR}(s, u, f, dt)$, and an initial state $s_{\text{init}} \in S$

compute a sequence of controls $\langle u_0, \ldots, u_{\ell-1} \rangle$, $u_i \in \mathcal{U}$, such that the trajectory $\zeta : \{0, 1, \ldots, \ell\} \to S$ obtained by starting at $s_{\text{init}}$ and applying the controls in succession for one time step $dt$, avoids the forbidden regions and satisfies the mission specification, i.e.,

$$
\forall i \in \{0, 1, \ldots, \ell\} : \text{COLLISION}(\zeta(i)) = \text{false} \quad \text{and} \quad \hat{\delta}_A(z_{\text{init}}, \text{WORD}(\zeta)) \in \text{ACCEPT}_A.
$$
**Fig. 3.** Illustration of a motion tree. Each edge $(v_i, v_j)$ represents a collision-free and dynamically-feasible trajectory from $v_i$ to $v_j$.

**Fig. 4.** Schematic illustration of the proposed approach, PUMMA.

### IV. METHOD

The search for a trajectory that satisfies the mission specification is conducted by expanding a tree of collision-free and dynamically-feasible motions. Starting from the initial state as the root, the motion tree is incrementally expanded by adding new branches. Each branch corresponds to a collision-free and dynamically-feasible trajectory obtained by applying input controls and simulating the vehicle dynamics for several time steps. The motion tree is maintained as a directed acyclic graph $T = (V_T, E_T)$. Fig. 3 provides an illustration. Consider a vertex $v \in V_T$. Let $\text{TRAJ}_T(v)$ denote the trajectory from the root of $T$ to $v$. A solution is found when the sequence of regions reached by $\text{TRAJ}_T(v)$ is accepted by $A$, i.e.,

$$
\hat{\delta}_A(z_{\text{init}}, \text{WORD}(\text{TRAJ}_T(v))) \in \text{ACCEPT}_A.
$$

(11)

A discrete abstraction, obtained by combining the DFA $A$ with a navigation roadmap, serves to guide the motion-tree expansion. The DFA $A$ is used to track the progress made by the motion tree toward reaching an accepting automaton state. In particular, each vertex $v \in V_T$ is associated with the automaton state obtained by running $A$ with $\text{WORD}(\text{TRAJ}_T(v))$ as the input. The roadmap is obtained by generating
collision-free configurations and connecting neighboring configurations with collision-free paths in order to construct a network of navigation routes that avoid the forbidden regions. The discrete abstraction is used to induce a partition of the motion tree into equivalence classes by grouping together vertices based on their corresponding automaton states and closest roadmap configurations.

A schematic illustration of the approach is provided in Fig. 4. Proceeding iteratively, each iteration consists of (i) selecting an equivalence class, and (ii) expanding the motion tree from a vertex associated with the selected equivalence class. During the selection process, preference is given to those equivalence classes that are deemed to be close to an accepting automaton state. Such estimates are obtained by searching the automaton and the roadmap for a sequence of waypoints that is compatible with the mission specification. After selecting an equivalence class attempts are made to expand the motion tree along its sequence of waypoints. The new vertices generated during the expansion are added to the motion tree. New equivalence classes could also be created when these vertices reach new goal regions. The process of selecting and expanding an equivalence class is repeated until a solution is found.

The rest of the section is organized as follows. Components of the approach, such as the navigation roadmap, abstract search, and motion-tree partition, are described in more detail in Sections IV-A, IV-B, and IV-C, respectively. The overall approach is described in Section IV-D.

A. Navigation Roadmap

A navigation roadmap is used to guide the overall expansion of the motion tree. In particular, the roadmap solves a relaxed version of the problem that does not take into account the vehicle dynamics or the drift caused by the ocean currents. In the relaxed setting, the objective is to quickly compute sequences of waypoints that are compatible with the mission specification. These waypoints provide navigation routes which guide the expansion of the motion tree. When expanding the motion tree, the vehicle dynamics and the drift caused by the ocean currents are taken into account in order to generate dynamically-feasible motion trajectories. As discussed in Section IV-D, the approach will seek alternative navigation routes when the motion-tree expansion along the current navigation route becomes difficult.

Drawing from PRM [7], the navigation roadmap is constructed by sampling collision-free configurations and connecting neighboring configurations with collision-free paths. Pseudocode for the roadmap construction is provided in Alg. 1. An illustration is shown in Fig. 5. Examples of the roadmaps for the scenes used in the experiments are shown in Fig. 7. The roadmap is maintained as a weighted undirected graph

\[ RM = (V_{RM}, E_{RM}, \text{COST}_{RM}) \]

where \( V_{RM} \) denotes the vertices, \( E_{RM} \) denotes the edges, and \( \text{COST}_{RM} \)}
NAVIGATION ROADMAP($\mathcal{W}, \mathcal{R}, \mathcal{O}_{\text{forbidden}}, \mathcal{O}_{\text{DepthSurf}}, n_{\text{add}}, n_{\text{neighbors}}$)

1. $V_{RM} \leftarrow \emptyset$; $E_{RM} \leftarrow \emptyset$; attempts $\leftarrow \emptyset$
2. for each $\mathcal{R}_i \in \mathcal{R}$ do
   \{ $q_i \leftarrow$ some point in $\mathcal{R}_i$; $V_{RM} \leftarrow V_{RM} \cup \{q_i\}$ \}
3. repeat
4.   $V \leftarrow \emptyset$; for $i = 1 \ldots n_{\text{add}}$ do $\{ q \leftarrow$ generate valid configuration; $V \leftarrow V \cup \{q\}; V_{RM} \leftarrow V_{RM} \cup \{q\} \}$
5.   for each $q \in V$ do
6.     for each $q' \in \text{NEARESTNEIGHBORS}(V_{RM}, q, n_{\text{neighbors}})$ do
7.       if $(q, q') \notin \text{attempts}$ then
8.         if $\text{ISPATHINCOLLISION}(q, q') = \text{false}$ then
9.           $E_{RM} \leftarrow E_{RM} \cup \{(q, q'), (q', q)\}$; $\text{COST}_{RM}(q, q') \leftarrow \text{COST}_{RM}(q', q) \leftarrow \frac{||q-q'||}{\text{CLEARANCE}(q) \cdot \text{CLEARANCE}(q')}$
10.         attempts $\leftarrow$ attempts $\cup \{(q, q'), (q', q)\}$
11. until $\text{SAMECOMPONENT}(RM, q_1, \ldots, q_n) = \text{true}$
12. return $RM = (V_{RM}, E_{RM}, \text{COST}_{RM})$

Algorithm 1: Pseudocode for constructing the navigation roadmap.

Fig. 5. Illustration of a navigation roadmap. Configurations $q_1, \ldots, q_4$ are associated with the goal regions $\mathcal{R}_1, \ldots, \mathcal{R}_4$, respectively.

provides the cost for each edge. Each roadmap vertex corresponds to a configuration $q = (q_x, q_y, q_d)$ which defines the position and depth. Initially, a configuration $q_i$ is added to the roadmap for each goal region $\mathcal{R}_i \in \mathcal{R}$ (Alg. 1:2). The configuration $q_i$ could correspond to the centroid of $\mathcal{R}_i$ or some other point inside $\mathcal{R}_i$. Afterwards, the roadmap is populated with a number of valid configurations generated at random (Alg. 1:4). A configuration is considered valid if it is not too close to a forbidden region and it is within the depth limits provided by the bathymetry map. To speed up the generation of valid configurations, the triangular mesh $\mathcal{O}_{\text{DepthSurf}}$ constructed from the bathymetry map (Section III-A) is sampled to generate candidate positions. More specifically, a configuration $q$ is generated by first sampling a point $(q_x, q_y, z)$ at random on $\mathcal{O}_{\text{DepthSurf}}$. This is achieved by selecting a triangle at random from $\mathcal{O}_{\text{DepthSurf}}$ and then generating a point at random inside the selected triangle. Note that $z$ provides the maximum depth at $(q_x, q_y)$. The configuration $q$ is then set to $(q_x, q_y, q_d)$, where $q_d$ is obtained by generating a random number between 0 and $z$. If $q$ is closer than a clearance threshold (set to the length of the vehicle) to the forbidden regions, then $q$ is discarded. The above procedure is repeated until a valid configuration $q$ is generated.

The procedure for generating a valid configuration is invoked numerous times in order to populate the roadmap (Alg. 1:4). Attempts are then made to connect each newly-added configuration to several of its
nearest neighbors in the roadmap (Alg. 1:5–10). The path from $q$ to $q'$ is considered valid if it avoids the forbidden regions and remains within the depth limits provided by the bathymetry map (Alg. 1:8). To speed up collision checking, a box is constructed from $q$ to $q'$ whose width and height are set to the maximum dimension of the vehicle. The PQP [48] collision-checking package is then used to determine whether or not the box collides with any of the triangular meshes $O_1, \ldots, O_m$ corresponding to the forbidden regions or the triangular mesh $O_{\text{DepthSurf}}$ corresponding to the bathymetry map. If there is no collision, the edges $(q, q')$ and $(q', q)$ are added to the roadmap (Alg. 1:9).

The cost of a roadmap edge $(q, q') \in E_{RM}$ takes into account not only the distance between $q$ and $q'$ but also their distance from the forbidden regions and depth limits, i.e.,

$$
\text{COST}_{RM}(q, q') = \frac{||q - q'||}{\text{CLEARANCE}(q) \text{CLEARANCE}(q')},
$$

(12)

where $\text{CLEARANCE}(q)$ denotes the minimum distance from $q$ to $O_1, \ldots, O_m, O_{\text{DepthSurf}}$, or the boundaries of $\mathcal{W}$ (Alg. 1:9). In this way, $(q, q')$ is deemed to have low cost if $q$ and $q'$ are near each other and away from forbidden regions, depth limits, and the boundaries of the operational area. As a result, low-cost paths in the roadmap provide short and safe navigation routes.

The process of generating roadmap configurations and edges is repeated until the configurations corresponding to the goal regions belong to the same roadmap component (Alg. 1:10). This ensures that the roadmap contains paths that reach all the goal regions.

**B. Abstract Search**

In order to compute sequences of waypoints that satisfy the mission specification, the navigation roadmap $RM = (V_{RM}, E_{RM}, \text{COST}_{RM})$ is combined with the DFA $A$. Consider an abstract state $\langle z, q \rangle$, where $z \in Z_A$ is an automaton state and $q \in V_{RM}$ is a roadmap configuration. Suppose $\sigma = \langle \pi_{j_1}, \pi_{j_2}, \ldots, \pi_{j_k} \rangle \in \Pi^*$ is a word that leads to an accepting automaton state when starting from $z$, i.e., $\hat{\delta}_A(z, \sigma) \in \text{ACCEPT}_A$. Since $\pi_{j_i}$ is associated with $\mathcal{R}_{j_i}$, a sequence of waypoints that starts at $q$ and reaches $\mathcal{R}_{j_1}, \mathcal{R}_{j_2}, \ldots, \mathcal{R}_{j_k}$ in order satisfies the mission. As an example, referring to the DFA for the alternating mission in Fig. 1(e) and the roadmap in Fig. 5, if $\langle z, q \rangle$ refers to $\langle z_0, q_{16} \rangle$, then $\sigma$ could be $\langle \pi_1, \pi_3, \pi_2, \pi_4 \rangle$ and the sequence of waypoints could be $\Lambda = \langle q_{16}, q_1, q_{15}, q_3, q_2, q_8, q_4 \rangle$. In order to compute such sequences, the search is conducted over both $A$ and $RM$ using a function of the form

$$
\langle \sigma, \Lambda \rangle \leftarrow \text{ABSTRACTSEARCH}(A, RM, \langle z, q \rangle),
$$

(13)
which computes a word \( \sigma \) that reaches an accepting automaton state and a minimum-cost roadmap path \( \Lambda \) that satisfies the mission specification when starting at \( z \in Z_A \) and \( q \in V_{RM} \).

The abstract search can be conducted using any graph-search algorithm for finding minimum-cost paths. Graph-search algorithms require as input a function \( \text{EDGES}((z', q')) \) to determine the abstract edges coming out of \((z', q')\) for any \( z' \in Z_A \) and \( q' \in V_{RM} \). In order to combine the DFA \( \mathcal{A} \) with the roadmap \( RM \), the abstract edges are defined as

\[
\text{EDGES}((z', q')) = \{ (z'', q_i) : \pi_i \in \Pi \land z'' = \delta_A(z', \pi_i) \land \text{HAS}((RM, q', q_i, \pi_i) = \text{true}) \}. \quad (14)
\]

In this way, there is an abstract edge from \((z', q')\) to \((z'', q_i)\) if there is an automaton transition from \( z' \) to \( z'' \) with input \( \pi_i \), where \( \pi_i \) is the label and \( q_i \) is the roadmap configuration associated with \( R_i \). Moreover, there should be a roadmap path from \( q' \) to \( q_i \). The function \( \text{HAS}((RM, q', q_i, \pi_i) \) returns true if and only if there is a roadmap path from \( q' \) to \( q_i \) that does not go through any of the roadmap configurations labeled with \( \pi_j \in \Pi \setminus \{ \pi_i \} \). This ensures that only \( \pi_i \) is satisfied along this path. In other words, reaching \( R_i \) from \( q' \) leads to an automaton transition from \( z' \) to \( z'' \).

Continuing with our example, referring to the DFA for the alternating mission in Fig. 1(e) and the roadmap in Fig. 5, \( \text{EDGES}((z_0, q_{16})) = \{ (z_1, q_1), (z_2, q_2) \}, \text{EDGES}((z_1, q_1)) = \{ (z_3, q_3), (z_4, q_4) \}, \text{EDGES}((z_2, q_2)) = \{ (z_5, q_5), (z_6, q_4) \}, \) and so on.

The cost of an abstract edge \((z', q'), (z'', q_i)\) is defined as the cost of the shortest path in the roadmap \( RM \) from \( q' \) to \( q_i \), i.e.,

\[
\text{COST}((z', q'), (z'', q_i)) = \text{COST}_{\text{SHORTEST}}(RM, q', q_i). \quad (15)
\]

As an example, \( \text{COST}((z_0, q_{16}), (z_2, q_2)) = \text{COST}_{RM}(q_{16}, q_4) + \text{COST}_{RM}(q_4, q_2) \) since \((q_{16}, q_{14}, q_2)\) is the shortest roadmap path from \( q_{16} \) to \( q_2 \). In order to compute the costs of the abstract edges efficiently, Dijkstra’s algorithm is used to precompute the shortest paths in the roadmap \( RM \) to each \( q_1, \ldots, q_n \), where \( q_i \) is the roadmap configuration associated with the goal region \( R_i \). A single call to Dijkstra’s algorithm with \( q_i \) as the source computes the shortest paths from any roadmap configuration \( q' \in V_{RM} \) to \( q_i \). Therefore, Dijkstra’s algorithm is invoked only \( n \) times, using each \( q_1, \ldots, q_n \) as the source. The shortest paths and their costs are stored and retrieved whenever needed. In this way, \( \text{COST}((z', q'), (z'', q_i)) \) is implemented efficiently by retrieving the cost of the shortest roadmap path from \( q' \) to \( q_i \).

Using the above definitions for the abstract edges and their costs, \( \text{ABSTRACT-search}(\mathcal{A}, RM, (z, q)) \)
is implemented as A* search in order to find the minimum-cost path from $\langle z, q \rangle$ to an accepting automaton state. As described in Section IV-D, the abstract search is used to guide the motion-tree expansion.

C. Motion Tree and Equivalence Classes

As mentioned, a motion tree, represented as a directed acyclic graph $T = (V_T, E_T)$, is used to conduct the search for a trajectory that satisfies the mission specification. The motion tree takes into account the vehicle dynamics and the drift caused by the ocean currents. Each vertex $v \in V_T$ is associated with a valid state in $S$, denoted by $v.s$. Each edge $(v, v') \in E_T$ corresponds to a dynamically-feasible motion from $v.s$ to $v'.s$. Fig. 3 provides an illustration.

To track the progress made with respect to the mission, each vertex $v \in V_T$ is associated with the automaton state obtained by running $A$ with the sequence of regions reached by $\text{TRAJ}_T(v)$ as input, i.e.,

$$v.z = \hat{\delta}_A(z_{\text{init}}, \text{WORD}(\text{TRAJ}_T(v))).$$

As an example, referring to the motion tree in Fig. 3 and the DFA for the alternating mission in Fig. 1(e), vertices $v_1, \ldots, v_7$ are associated with $z_0$, vertices $v_8, \ldots, v_{17}$ with $z_1$, vertices $v_{18}, \ldots, v_{24}$ with $z_3$, vertices $v_{25}, \ldots, v_{34}$ with $z_7$, and vertex $v_{35}$ with $z_9$. If $v.z \in \text{ACCEPT}_A$, then $\text{TRAJ}_T(v)$ provides a trajectory that satisfies the mission specification.

In order to take advantage of the navigation roadmap $RM$, each vertex $v \in V_T$ is also associated with the nearest configuration in $RM$, i.e.,

$$v.q = \arg\min_{q' \in V_{RM}} ||q' - \text{CFG}(v.s)||.$$  \hspace{1cm} (17)

As an illustration, referring to the roadmap in Fig. 5 and the motion tree in Fig. 3, vertices $v_1, v_2, v_4, v_5$ are associated with $q_{16}$, vertex $v_3$ with $q_{10}$, vertices $v_6, \ldots, v_{10}$ with $q_1$, and so on.

Using the DFA $A$ and the roadmap $RM$, the motion tree is partitioned into equivalence classes. More specifically, an automaton state $z \in Z_A$ and a roadmap configuration $q \in V_{RM}$ define an equivalence class $\Gamma_{\langle z, q \rangle}$ which contains all the vertices in $V_T$ that are associated with both $z$ and $q$, i.e.,

$$\Gamma_{\langle z, q \rangle} = \{ v : v \in V_T \land v.z = z \land v.q = q \}.$$  \hspace{1cm} (18)

As an example, referring to the DFA for the alternating mission in Fig. 1(e), the roadmap in Fig. 5, and the motion tree in Fig. 3, $\Gamma_{\langle z_0, q_{16} \rangle} = \{ v_1, v_2, v_4, v_5 \}$, $\Gamma_{\langle z_0, q_{10} \rangle} = \{ v_3 \}$, $\Gamma_{\langle z_1, q_7 \rangle} = \{ v_{11}, v_{12} \}$.
This induces a partition of the motion tree into nonempty equivalence classes, i.e.,

$$\Gamma = \{ \Gamma_{(z,q)} : z \in Z_A \land q \in V_{RM} \land |\Gamma_{(z,q)}| > 0 \},$$  \hspace{1cm} (19)

where $|\Gamma_{(z,q)}|$ denotes the set cardinality. As described next, the equivalence classes are used to effectively guide the motion-tree expansion toward an accepting automaton state.

**D. Overall Approach**

Pseudocode for the overall approach, PUMMA, is given in Alg. 2 and a schematic illustration is shown in Fig. 4. PUMMA starts by computing the navigation roadmap $RM = (V_{RM}, E_{RM}, \text{COST}_{RM})$ (Alg. 2:1), which will be used to guide the expansion of the motion tree $T = (V_T, E_T)$. The motion tree $T$ is rooted at the vertex $v_{\text{init}}$ which is associated with the initial state $s_{\text{init}} \in S$ (Alg. 2:2). The root vertex is also associated with the initial automaton state, i.e., $v_{\text{init}}.z = z_{\text{init}}$, and the nearest roadmap configuration, i.e., $v_{\text{init}}.q = \arg\min_{q' \in V_{RM}} ||\text{CFG}(s_{\text{init}}) - q'||$. The first equivalence class $\Gamma_{(v_{\text{init}}.z,v_{\text{init}}.q)}$ is also created and added to $\Gamma$ (Alg. 2:3). Afterwards, a minimum-cost path $\Lambda_{(v_{\text{init}}.z,v_{\text{init}}.q)}$ that starts at the automaton state $v_{\text{init}}.z$ and roadmap configuration $v_{\text{init}}.q$ and reaches an accepting automaton state is computed by conducting an abstract search over both the DFA $A$ and the roadmap $RM$ (Alg. 2:4). As described later in the section, the path $\Lambda_{(v_{\text{init}}.z,v_{\text{init}}.q)}$ will be used to guide the expansion of the equivalence class $\Gamma_{(v_{\text{init}}.z,v_{\text{init}}.q)}$.

After the initialization, $T$ is iteratively expanded by adding new vertices and new edges. Each iteration consists of selecting an equivalence class and then expanding the motion tree from one of the vertices associated with the selected equivalence class. This process is repeated until a solution is found or an upper bound on the running time is reached (Alg. 2:5–20).

1) **Selecting an Equivalence Class:** PUMMA seeks to select an equivalence class $\Gamma_{(z,q)}$ from $\Gamma$ that is going to promote expansions of the motion tree toward an accepting automaton state. For this reason, the weight of an equivalence class $\Gamma_{(z,q)}$ is defined as

$$w(\Gamma_{(z,q)}) = \frac{\beta_{\text{NRSEL}(\Gamma_{(z,q)})}}{2^{d_A(z)} (\text{COST}(\Lambda_{(z,q)}))^\alpha}. \hspace{1cm} (20)$$

Then, the equivalence class in $\Gamma$ with the maximum weight is selected for expansion (Alg. 2:6), i.e.,

$$\arg\max_{\Gamma_{(z,q)} \in \Gamma} w(\Gamma_{(z,q)}). \hspace{1cm} (21)$$
MISSI ON AND M OTION PLANNING FOR AUVS 18

PUMMA(A, SIMULATOR, DRIFT, s_{init}, W, R, \mathcal{O}, \mathcal{O}_{DepthSurf}, n_{add}, n_{neighbors}, t_{max})

1. \(RM = (V_{RM}, E_{RM}, COST_{RM}) \leftarrow NAVIGATION\textsc{ROADMAP}(W, R, \mathcal{O}, \mathcal{O}_{DepthSurf}, n_{add}, n_{neighbors})\)

\(\triangleright\) initialize motion tree

2. \(v_{init}.s \leftarrow s_{init}; v_{init}.z \leftarrow z_{init}; v_{init}.q \leftarrow \arg\min_{q' \in V_{RM}} \|q' - CFG(s_{init})\|; v_{init}.t \leftarrow 0\)

3. \(V_T \leftarrow \{v_{init}\}; E_T \leftarrow \emptyset; \Gamma_{(v_{init}, z, v_{init}.q)} \leftarrow \{v_{init}\}; \Gamma \leftarrow \{\Gamma_{(v_{init}, z, v_{init}.q)}\}\)

4. \(\Lambda(v_{init}.z, v_{init}.q) \leftarrow \text{ABSTRACTSEARCH}(A, RM, (v_{init}.z, v_{init}.q))\)

5. \(\textbf{while} \ T I M E < t_{\text{max}}\)

\(\triangleright\) select equivalence class

6. \(\Gamma(\langle z, q \rangle) \leftarrow \arg\max_{\langle z', q' \rangle \in \Gamma} w(\Gamma(\langle z', q' \rangle)) \) where \(w(\Gamma(\langle z', q' \rangle)) = \left(\frac{\text{NRSEL}(\Gamma(\langle z', q' \rangle))}{\text{LENGTH}(\Lambda(\langle z', q' \rangle))}\right)^{\beta}\)

\(\triangleright\) select the target waypoint and the vertex from which to expand the motion tree

7. \(p \leftarrow \text{SELECTTARGETWAYPOINT}(\Lambda(v_{\text{target}})); v \leftarrow \arg\min_{v' \in \Gamma(\langle z, q \rangle)} \|p - CFG(v')\|\)

\(\triangleright\) expand the motion tree toward the target waypoint from the selected vertex

8. \(\textbf{for} \text{ several steps do}\)

9. \(u \leftarrow \text{CONTROLLER}(v, s, p); f \leftarrow \text{DRIFT}(\text{POS}(v, s), \text{DEPTH}(v, s), v, t); s_{\text{new}} \leftarrow \text{SIMULATE}(v, s, u, f, dt)\)

10. \(\textbf{if} \ \text{COLLISION}(s_{\text{new}}) = \text{true} \ \textbf{then} \ \text{break} \ \textbf{for} \text{ loop}\)

11. \(v_{\text{new}.s} \leftarrow s_{\text{new}}; v_{\text{new}.q} \leftarrow \arg\min_{q' \in V_{RM}} \|q' - CFG(s_{\text{new}})\|; v_{\text{new}.t} \leftarrow v.t + dt; v_{\text{new}.\pi} \leftarrow \text{PROP}(R, s_{\text{new}})\)

12. \(\textbf{if} \ v_{\text{new}.\pi} \in \Pi \ \text{and} \ v_{\text{new}.\pi} \neq v.\pi \ \text{then} \ v_{\text{new}.z} \leftarrow \delta_{A}(v.z, v_{\text{new}.\pi}) \ \text{else} \ v_{\text{new}.z} \leftarrow v.z\)

13. \(V_T \leftarrow V_T \cup \{v_{\text{new}}\}; E_T \leftarrow E_T \cup \{(v, v_{\text{new}})\}\)

14. \(\langle z', q' \rangle \leftarrow \langle v_{\text{new}.z}, v_{\text{new}.q} \rangle; \Gamma(\langle z', q' \rangle) \leftarrow \text{FIND}(\Gamma, \langle z', q' \rangle)\)

15. \(\textbf{if} \ \Gamma(\langle z', q' \rangle) = \text{null} \ \text{then} \ \text{return} \ \text{TRAJ}(v)\)

16. \(\Gamma(\langle z', q' \rangle) \leftarrow \text{NEW\textsc{EQUIVALENCECLASS}}(); \Lambda(\langle z', q' \rangle) \leftarrow \text{ABSTRACTSEARCH}(A, RM, \langle z', q' \rangle); \Gamma \leftarrow \Gamma \cup \{\Gamma(\langle z', q' \rangle)\}\)

17. \(\Gamma(\langle z', q' \rangle) \leftarrow \Gamma(\langle z', q' \rangle) \cup \{v_{\text{new}}\}\)

18. \(\textbf{if} \ v_{\text{new}.z} \in \text{ACCEPT}\_A \ \text{then} \ \text{return} \ \text{TRAJ}(v)\)

19. \(\textbf{if} \ \text{NEARTARGET}(\text{POS}(v_{\text{new}.s}, p)) = \text{true} \ \text{then} \ \text{break} \ \textbf{for} \text{ loop} \ \text{else} \ v \leftarrow v_{\text{new}}\)

20. \textbf{return} \text{null}

Algorithm 2: Pseudocode for the overall approach, PUMMA.

In the weight formulation, \(d_A(z)\) denotes the cost of the shortest path in \(A\) from \(z\) to an accepting automaton state. By having \(w(\Gamma(\langle z, q \rangle))\) increase exponentially as \(d_A(z)\) decreases, the selection procedure seeks to promote expansions of the motion tree from equivalence classes that are deemed to be close to accepting automaton states. This is strengthened by also taking into account the cost of the shortest path \(\Lambda(v_{\text{target}})\) from the abstract state \(\langle z, q \rangle\) to an accepting automaton state. The navigation route \(\Lambda(v_{\text{target}})\) is computed by conducting an abstract search over both the DFA \(A\) and the navigation roadmap \(RM\).

The term \(1/(\text{COST}(\Lambda(\langle z, q \rangle)))^{\alpha}\) serves as a heuristic to determine the feasibility of reaching an accepting automaton state by expanding the motion tree from \(\Gamma(\langle z, q \rangle)\). The exponent \(\alpha \geq 1\) provides a parameter to tune the impact of the heuristic on the selection procedure.

The term \(\beta^{\text{NRSEL}(\Gamma(\langle z, q \rangle))}\), \(0 < \beta < 1\), provides a measure to counter the bias of the heuristic cost. In this expression, \(\text{NRSEL}(\Gamma(\langle z, q \rangle))\) denotes the number of times \(\Gamma(\langle z, q \rangle)\) has been previously selected for expansion. Therefore, \(w(\Gamma(\langle z, q \rangle))\) decreases exponentially as \(\text{NRSEL}(\Gamma(\langle z, q \rangle))\) increases. By applying a penalty after each selection, PUMMA seeks to avoid overexploration or becoming stuck when expansions from \(\Gamma(\langle z, q \rangle)\) are not feasible due to constraints imposed by the vehicle dynamics, ocean currents, depth limits, or the forbidden regions. In fact, suppose that there is no selection penalty. Let \(\Gamma(\langle z, q \rangle)\) be the equivalence class
selected for expansion. Suppose also that attempts to expand from $\Gamma_{z,q}$ fail. Since $w(\Gamma_{z,q})$ would not change, then $\Gamma_{z,q}$ would be selected again and again, and so PUMMA would become stuck. By having a selection penalty, $w(\Gamma_{z,q})$ is reduced after each selection. Eventually the weight of some other $\Gamma_{z',q'}$ would become greater than $w(\Gamma_{z,q})$. In this way, the selection penalty provides the flexibility to expand the motion tree from other equivalence classes.

2) Expanding an Equivalence Class: Once an equivalence class $\Gamma_{z,q}$ is selected, the objective is to expand the motion tree $T$ from one of the vertices of $\Gamma_{z,q}$ along the navigation route $\Lambda_{z,q}$. Let $R_{i_1} \in \mathcal{R}$ denote the first goal region reached by $\Lambda_{z,q}$. A target point $p = (p_x, p_y, p_d)$, where $(p_x, p_y)$ is the position and $p_d$ is the depth, is then generated by sampling a random point along the edge of $\Lambda_{z,q}$ from $q$ to $R_{i_1}$ (Alg. 2:7). More specifically, the shortest path in the roadmap $RM$ from $q$ to $q_{i_1}$ is first retrieved. One of the edges in this path is selected at random and the target point $p$ is sampled at random on this edge.

The motion tree $T$ is then expanded from the vertex $v \in \Gamma_{z,q}$ that is closest to $p$, i.e.,

$$v = \arg\min_{v' \in \Gamma_{z,q}} ||p - \text{CFG}(v'.s)||. \tag{22}$$

The expansion from $v$ toward $p$ seeks to make progress along $\Lambda_{z,q}$. If the tree expansion reaches $R_{i_1}$, it will enable an automaton transition to a state $z' = \delta_A(z, \pi_{i_1})$ that is closer to an accepting state.

In order to expand $T$ from $v$ toward $p = (p_x, p_y, p_d)$, control inputs $u$ are applied to $v.s$ for several time steps (Alg. 2:8–19). The control function sets the desired depth to $p_d$ (Alg. 2:9). The heading is set in such a way so that the vehicle starts turning toward $(p_x, p_y)$. As it moves toward $p$, the controller seeks to maintain a constant velocity (set to 1.5 m/s in the experiments). Internally, as discussed in Section III-A, the MOOS-IvP simulator uses PID controllers to determine appropriate actuator values that would steer the vehicle along the desired heading, depth, and speed. After determining the appropriate control input $u$, the MOOS-IvP simulator is used to determine the new state $s_{\text{new}}$ obtained by applying $u$ to the state $v.s$ for one time step $dt$. The simulator takes into account the vehicle dynamics and the drift $f$ caused by the ocean currents (Alg. 2:9). The drift $f$ is computed by the function $\text{DRIFT}$, which models the time-varying ocean currents, based on the position, depth, and time associated with the vertex $v$. If the new state $s_{\text{new}}$ results in collision with the forbidden regions or violates the depth limits imposed by the bathymetry map, the tree expansion toward $p$ stops (Alg. 2:10). The approach would then go back to line 6 and continue the iterations by selecting an equivalence class. If there are no collisions, a new

\[\text{The time } v.t \text{ corresponds to the time associated with the trajectory } \text{TRAJ}_T(v) \text{ from the root of } T \text{ to } v. \text{ The time } v.t \text{ is computed during the creation of the vertex } v \text{ by adding } dt \text{ to the time associated with the parent of } v.\]
vertex $v_{\text{new}}$ and a new edge $(v, v_{\text{new}})$ is added to the motion tree $\mathcal{T}$ (Alg. 2:11). The new vertex $v_{\text{new}}$ is associated with the nearest roadmap configuration to $\text{CFG}(s_{\text{new}})$. The function $\text{PROP} (\mathcal{R}, s_{\text{new}})$ determines the goal region, if any, reached by $s_{\text{new}}$. If some region $\mathcal{R}_i$ contains $s_{\text{new}}$ and $\mathcal{R}_i$ is different than the region containing its parent $v$, then the automaton state associated with $v_{\text{new}}$ is obtained by following the automaton transition from $v.z$ with input $\pi_i$, i.e., $v_{\text{new}}.z = \delta_A(v.z, \pi_i)$. Otherwise, $v_{\text{new}}$ is associated with the same automaton state as its parent $v$ (Alg. 2:13).

Afterwards, $\Gamma$ is searched to determine whether or not $\Gamma\langle z', q' \rangle$ with $\langle z', q' \rangle = \langle v_{\text{new}}.z, v_{\text{new}}.q \rangle$ already exists in $\Gamma$. If not, $\Gamma\langle z', q' \rangle$ is created as a new equivalence class (Alg. 2:14–16). At this time, a minimum-cost navigation route $\Lambda\langle z', q' \rangle$ to an accepting automaton state is computed by conducting an abstract search over both the automaton $A$ and the navigation roadmap $\mathcal{R}M$. The cost of $\Lambda\langle z', q' \rangle$ is used to determine the weight associated with $\Gamma\langle z', q' \rangle$, as described earlier in this section. The new equivalence class $\Gamma\langle z', q' \rangle$ is added to $\Gamma$ so that it can be used by the approach to expand the motion tree $\mathcal{T}$ in future iterations. The vertex $v_{\text{new}}$ is added to $\Gamma\langle z', q' \rangle$ (Alg. 2:17).

If $v_{\text{new}}.z \in \text{ACCEPT}_A$, then the search terminates successfully. In this case, $\text{TRAJ}_T(v_{\text{new}})$ provides a collision-free and dynamically-feasible motion trajectory that satisfies the mission specification (Alg. 2:18). If $v_{\text{new}}$ reaches or gets near (within some $d_{\text{target}}$ distance, set to $5m$ in the experiments) the target point $p$, then the expansion from $\Gamma\langle z, q \rangle$ terminates successfully. In such cases, the approach goes back to line 6 and continues with the selection of an equivalence class. Otherwise, the expansion toward $p$ continues from $v_{\text{new}}$ (Alg. 2:19). The process of selecting and expanding an equivalence class continues until a solution is found or an upper bound on the running time is reached.

In the worst-case scenario, PUMMA will generate $|\mathcal{V}_{\mathcal{R}M}| \cdot |Z_A|$ equivalence classes, one for each combination of roadmap configuration and automaton state. In the best-case scenario, PUMMA will generate as many equivalence classes as the number of configurations along the shortest abstract path from the initial roadmap configuration to an accepting automaton state. This corresponds to the case where PUMMA is able to follow the initial abstract path. Another property of PUMMA is that the number of times each equivalence class is selected for expansion cannot be bounded. In fact, assume to the contrary that $\Gamma\langle z, q \rangle$ is never selected for expansion after the $i$-th iteration. Hence, its weight will not change. Since some other equivalence class $\Gamma\langle z', q' \rangle$ must be selected, $w(\Gamma\langle z', q' \rangle)$ is reduced by the selection penalty factor $\beta$. As a result, after a finite number of iterations, the weight of every other equivalence class in $\Gamma$ will become less than $w(\Gamma\langle z, q \rangle)$. This leads to a contradiction since $\Gamma\langle z, q \rangle$ would now have to
Fig. 6. Scenes and the AUV model used in the simulation experiments. Figures may be best viewed in color and on screen. Due to differences in the scene dimensions, figures are shown using a different scale for each scene. Each figure also shows an example of a collision-free and dynamically-feasible trajectory computed by the approach for solving a given mission specification. Goal regions are labeled with a corresponding number; the start position of the AUV is indicated by S. (a) Solution is for a grouping mission $\phi_{\text{groups}}(G_1, G_2, G_3)$ (Eqn. 4) where the AUV has to reach all the regions in one group before moving on to the next. In the example, $G_1 = \{\pi_3, \pi_7, \pi_9, \pi_6\}$, $G_2 = \{\pi_1, \pi_4\}$, and $G_3 = \{\pi_2, \pi_5, \pi_8\}$. (b) Solution is for an alternating mission $\phi_{\text{alternate}}(\pi_1, \ldots, \pi_6)$ (Eqn. 5) where the AUV has to alternate between reaching a region from $\{R_1, R_2, R_3\}$ and $\{R_4, R_5, R_6\}$ until all regions have been reached. (c) Solution is for a partial-order mission $\phi_{\text{po}}(\pi_1, \ldots, \pi_{10})$ (Eqn. 3) where the AUV has to reach each of the regions $R_1, \ldots, R_5$ before reaching each of the regions $R_6, \ldots, R_{10}$. The figure also shows the drift caused by the ocean currents (as white arrows). The drift has been magnified (by a factor of 1000) to be visible in the figure. (d) Solution is for a coverage mission $\phi_{\text{cov}}(\pi_1, \ldots, \pi_{15})$ (Eqn. 2) where the AUV has to reach each region $R_1, \ldots, R_{15}$ (order does not matter). The figure also shows the ocean drift (magnified by a factor of 2000).

be selected since it has the maximum weight. This ensures that PUMMA will methodically explore all the equivalence classes.

V. EXPERIMENTS AND RESULTS

The approach, PUMMA, is evaluated both in simulation and field experiments. In simulation, the AUV is required to operate in environments consisting of complex bathymetry, time-varying ocean currents, and forbidden regions. The AUV is required to perform sequencing, coverage, partial order, grouping, alternating, and hierarchical missions as specified via regular languages. Field experiments are conducted with an OceanServer Iver2 AUV over the NRL Chesapeake Bay Detachment. Comparisons to related work show significant improvement in both runtime and cost of solution trajectories.
A. Models of Scenes, Bathymetry, and Time-Varying Ocean Currents

Fig. 6 shows the scenes and the OceanServer Iver2 AUV model (Section III-A) used in the simulation experiments. The figure also shows solution trajectories obtained by the approach. Examples of the navigation roadmaps for each scene are shown in Fig. 7.

The first scene, shown in Fig. 6(a), corresponds to the NRL Chesapeake Bay Detachment. The AUV is required to operate over a $1.2km \times 0.7km$ area with depth of $10m$. Forbidden regions are placed in such a way as to make it difficult for the AUV to navigate freely in order to demonstrate the ability of the approach to plan collision-free and dynamically-feasible trajectories in complex environments.

The second scene, shown in Fig. 6(b), has an operational area of $1.2km \times 0.7km$ and depth of $60m$. Obstacles are placed vertically one after the other every $120m$ in such a way as to force the AUV to move up and down. Each obstacle has a height of $40m$, leaving a gap of $20m$ for passing. In fact, obstacles placed in odd positions go down to the maximum depth, so the only way to pass these obstacles is by going over them. Obstacles placed in even positions start at the surface, so the only way to pass these obstacles is by going under them. This scene demonstrates the ability of the approach to plan in 3D environments where the AUV has to change its depth as it seeks to accomplish the assigned mission.

The third scene, shown in Fig. 6(c), corresponds to a riverine environment (Patuxent river emptying into the Chesapeake Bay) with an operational area of $21.3km \times 9km$ and maximum depth of $45m$. The bathymetry map is obtained from the National Geophysical Data Center (NGDC) [51]. The riverine environment is characterized by complex bathymetry which inhibits the movement of the AUV throughout the river while becoming more expansive once it empties into the bay. Time-varying ocean currents are...
provided by the Chesapeake Bay Operational Forecast System (CBOFS) [52]. In simulation the data used is historical but for field operation over large bodies of water CBOFS provides forecast data including ocean currents, temperature, salinity, and water level. This scene demonstrates the ability of PUMMA to effectively plan feasible trajectories while taking into account the ocean drift.

The fourth scene, shown in Fig. 6(d), corresponds to an estuary (Chesapeake Bay exiting into the Atlantic Ocean) with a large operational area of $41.4\text{km} \times 62.9\text{km}$ and maximum depth of $50\text{m}$. The estuary is characterized by high traffic from surface vessels along shipping channels. When the AUV passes through these high-traffic regions (shown in brown), it must maintain a minimum distance from the surface in order to avoid collisions with surface vessels. The bathymetry map, obtained from NGDC [51], also poses unique challenges since the high-traffic regions are co-located with deep shipping channels. As in the third scene, data for the time-varying ocean currents is obtained from CBOFS [52]. This scene demonstrates the ability of the approach to plan over large operational areas characterized by high levels of traffic at the surface level.

As mentioned, $\text{DRIFT}(x, y, d, t)$ uses data obtained from CBOFS to determine the drift caused by the time-varying currents at position $(x, y)$, depth $d$, and time $t$. Each data file provides the drift forces over each cell in a regular 3D grid over a particular time interval. For the experiments, the time duration of each data file was set to 30 minutes. $\text{DRIFT}(x, y, d, t)$ is then computed by first locating the data file corresponding to time $t$ and then locating the grid cell corresponding to $(x, y, d)$. The data files were loaded as tables in memory (occupying less than 128MB) in order to perform lookups in constant time.

B. Mission Specifications and Problem Instances

Experiments are conducting with sequencing $\phi^{n}_{\text{seq}}$, coverage $\phi^{n}_{\text{cov}}$, partial-order $\phi^{n}_{\text{po}}$, grouping $\phi^{n}_{\text{groups}}$, alternating $\phi^{n}_{\text{alternate}}$ missions (Eqn. 1–5) as well as $\phi^{n}_{\text{mix}}$ which combines different tasks, i.e.,

$$
\phi^{n}_{\text{mix}}(\pi_1, \pi_2, \ldots, \pi_n) = \phi^{n}_{\text{seq}}(G_1)\phi^{n}_{\text{cov}}(G_2)\phi^{n}_{\text{alternate}}(G_3)\phi^{n}_{\text{po}}(G_4),
$$

(23)

where $G_1, G_2, G_3, G_4$ denotes a partition of $\{\pi_1, \ldots, \pi_n\}$ into 4 groups. In the formulas above, the superscript $n$ denotes the number of goal regions. For each mission, the number of goal regions is varied as $n = 1, 3, 5, \ldots, 15$. Random placements of goal regions and starting positions are used in order to evaluate the performance of the approach under different conditions. In fact, 60 random instances are generated for each combination of scene and value of $n$, denoted by $I_{(\text{scene}, n)}$. In each instance, the AUV
is placed at a random starting position. More precisely, a placement is repeatedly generated at random until the AUV does not collide with any of the forbidden regions. Each goal region corresponds to a 3D box whose dimensions are $20m \times 20m \times 4m$, $20m \times 30m \times 6m$, $100m \times 100m \times 10m$, and $200m \times 250m \times 4m$, respectively for each of the four scenes shown in Fig. 6. The goal regions $R_1, \ldots, R_n$ are placed one after the other. To place $R_i$, a random position and depth is generated until $R_i$ is not in collision and is at least a certain distance away from the AUV and the previously-placed regions $R_1, \ldots, R_{i-1}$. The separation distance, which is used to ensure that the goal regions are not too close to each other and the AUV, is set to $20m$, $100m$, $1km$, and $2km$, respectively for each of the four scenes in Fig. 6.

Note that $\phi^n_{\text{groups}}$ requires as input a partition of $\{\pi_1, \ldots, \pi_n\}$ into $k$ groups $G_1, \ldots, G_k$. For the experiments, $k$ is set to $\lceil n/3 \rceil$ and, for each problem instance, two different partitions were used. The first partition, denoted by $\phi^n_{\text{groupsG}}$, uses Gonzalez’s $k$-centers clustering which seeks to reduce the intra-group distances. The second approach, denoted by $\phi^n_{\text{groupsE}}$, partitions the goals into almost equally-sized groups by assigning $\pi_i$ to $G_{1 + ((i-1) \mod k)}$. Since goal regions are placed at random positions, this partition tends to increase the intra-group distances. The equally-sized partition is also used to obtain the four groups $G_1, G_2, G_3, G_4$ required for $\phi^n_{\text{max}}$, where $\pi_i$ is assigned to $G_{1 + ((i-1) \mod 4)}$.

C. Planner used in the Comparisons

The approach, PUMMA, is compared to LTLSyclop [40]–[43] and SRRG [45]. The other planning approaches discussed in related work could not be used since they do not incorporate one or more aspects of the problems considered in this paper, such as vehicle dynamics, drift, forbidden regions, 3D environments, or missions given by regular languages.

LTLSyclop was modified to accept a DFA as input instead of a syntactically co-safe LTL formula (this was straightforward since LTLSyclop converted the LTL formula into a DFA). LTLSyclop computes at each iteration a discrete path $\sigma$ over the abstraction $A \times RM$. Afterwards, LTLSyclop sets $\sigma_{\text{avail}}$ to contain all the abstract states $\langle z, q \rangle$ in $\sigma$ that have been reached by the motion tree. To explore the discrete path, LTLSyclop selects an abstract state $\langle z, q \rangle$ from $\sigma_{\text{avail}}$ and attempts to expand the motion tree toward the next abstract state in $\sigma$. This process is repeated several times. The implementation of LTLSyclop used in the experiments is optimized and includes the recommended features such as switching between shortest and random guides and abandoning the guide when no progress is made. Moreover, LTLSyclop is used with the same roadmap abstraction as PUMMA.
SRRG was modified to incorporate the vehicle geometry and dynamics since the original version conducted experiments only with a point robot with no dynamics [45]. This was made possible by using the same collision checker and controller used by PUMMA. SRRG uses far connections to maintain the motion tree as a sparse graph and near connections to create cycles. To reduce the runtime, the near connections were not used in the experiments in this paper since cycles are not needed in the motion tree when considering regular languages (since finite paths as opposed to infinite paths satisfy the specification).

D. Results

To evaluate the performance, PUMMA, LTLSyclop, and SRRG are run on every problem instance. Results for a planner, a scene $S$, and a mission specification $\phi$ with $n$ goal regions are based on 60 runs, one for each problem instance in $I_{(S,n)}$. A timeout of 100s is set for each run. Results show the mean runtime, solution length, and standard deviations after removing the five best and worst runs to avoid the influence of outliers. Runtime includes everything from reading the input file to reporting that a solution is found. In particular, the runtime for PUMMA and LTLSyclop includes the time it takes to construct the roadmap. Solution length is measured as the distance traveled by the AUV. Experiments were run on an Intel Core i7 (CPU: 1.90GHz, RAM: 4GB) using Ubuntu 14.04 and GNU g++-4.8.2.

1) Runtime: Fig. 8 shows the runtime results when running the planners on each of the scenes in Fig. 6 using sequencing, coverage, partial order, grouping, alternating, and hierarchical missions (Eqn. 1–5, 23) and varying the number of goal regions from 1 to 15. The results show that the approach, PUMMA, is significantly faster, by an order of magnitude or more, than LTLSyclop and SRRG.

These experiments confirm the scalability issues observed in LTLSyclop. LTLSyclop performs an expensive search at each iteration and often spends considerable computational time attempting to expand the motion tree along infeasible directions. In fact, LTLSyclop computes the guide as a path from the initial abstract state, i.e., $\langle v_{init.z}, v_{init.q} \rangle$, to an accepting automaton state and then seeks to expand the motion tree along the configurations associated with the guide, trying to reach them one after the other. As the guide is often long, LTLSyclop ends up spending considerable computational time before switching to a new guide. In contrast, the computational efficiency of PUMMA derives from the partition of the motion tree into equivalence classes, the navigation roadmap, the abstract search, and the use of selection penalties. In fact, the motion-tree partition makes it possible to group tree vertices that provide the same abstract information. By searching both the navigation roadmap and the DFA, the abstract search provides short and safe navigation routes along which to expand each equivalence class. Moreover, the length of
the abstract path serves as a heuristic cost to bias the expansion from equivalence classes that are deemed to be close to accepting automaton state. The greedy aspect of the heuristic costs is complemented by selection penalties which enable the discovery of new directions toward accepting automaton states. As shown in Fig. 8, PUMMA efficiently plans collision-free and dynamically-feasible trajectories that satisfy the various missions given by regular languages. PUMMA performs well across different scenes even in the case of the estuary scene which has a large operational area (41.4km × 62.9km).
SRRG, as an RRT-based planner, expands the motion tree toward randomly-sampled configurations. As such, SRRG lacks global guidance toward accepting automaton states. However, global guidance is necessary given the vast space that has to be searched, the complexity of the motion dynamics and of the mission. As a result, SRRG often times out. In contrast, the proposed approach uses the navigation roadmap and the automaton to effectively guide the motion-tree expansion.
2) Solution Length: Fig. 9 shows the results on solution length, which is measured as the distance traveled by the AUV. The results indicate that PUMMA generates significantly shorter solutions than LTLSyclop and SRRG. In fact, PUMMA promotes the generation of short solutions since it biases the motion-tree expansion from equivalence classes that have short abstract paths to accepting automaton states. Fig. 9 also shows that PUMMA generates significantly shorter solutions when considering coverage as opposed to sequencing missions. A coverage mission gives the approach the flexibility to determine the order in which to visit the regions so that it can reduce the distance traveled by the AUV.

3) Runtime Distribution: Fig. 10 shows the runtime distribution for various components of PUMMA, such as the roadmap construction, dynamics simulation and collision checking, and abstract search. Note that the roadmap construction depends on the environment, so the roadmap construction time remains constant as the number of goal regions is increased. As expected, the time spent on expanding the motion tree (simulation and collision checking for the new vertices) increases with the number of goal regions since the problem becomes more complex. Since the number of equivalence classes increases as the tree is expanded (as shown in Fig. 11), there is also an increase in the running time of the abstract search. Overall, the results show that the roadmap construction and abstract search do not constitute a bottleneck but instead effectively guide the expansion of the motion tree toward an accepting automaton state.

Fig. 11 shows the number of equivalence classes created by PUMMA over different missions and scenes. The results show a sublinear increase with respect to the number of goals. While the maximum number of equivalence classes is $|V_{RM}| \times |Z_A|$, the results also show that PUMMA explores only a fraction of the possible equivalence classes. By promoting expansions from equivalence classes associated with
4) Impact of the Navigation Roadmap: Since the navigation roadmap plays an important role in guiding the expansion of the motion tree, additional experiments were conducted to evaluate its impact. Fig. 12 shows the impact of the number of neighbors ($n_{\text{neighbors}}$) and the roadmap batch size ($n_{\text{add}}$) on the overall performance of PUMMA (all the experiments presented so far have used $n_{\text{neighbors}} = 15$ and $n_{\text{add}} = 5000$). When the number of neighbors is small, the roadmap has more difficulty capturing the connectivity of the environment and providing short and safe navigation routes. This results in an increase in the runtime.
and solution length since it becomes more difficult to effectively expand the motion tree. Using a large number of neighbors increases the runtime to construct the roadmap. Fig. 12 shows that PUMMA works well for a wide range of values, e.g., $n_{\text{neighbors}} \in [5, 20]$. When considering the impact of the batch size, recall that $n_{\text{add}}$ configurations will be added to the roadmap in batches until the roadmap connects all the goal regions. Due to the random sampling, sometimes a small number of configurations captures the connectivity. This results in a large standard deviation when using a small batch size. Using a large batch size increases the runtime of the roadmap construction, and, hence, of the overall approach. Fig. 12 shows that PUMMA works well for a wide range of batch sizes, e.g., $n_{\text{add}} \in [2500, 7500]$.

Fig. 13 shows the results when generating the roadmap vertices over a regular grid as opposed to using random sampling as described in Section IV-A. More specifically, each vertex in the grid is added to the roadmap only when it is not in collision and it is within the depth limits provided by the bathymetry map. In order to improve the connectivity of the roadmap, in addition to connecting each vertex to its $k$-nearest neighbors, attempts are made to also connect each vertex to its 8-neighbors in the grid. As expected, results show that the runtime of PUMMA depends on the grid resolution. When the grid is too sparse, the runtime increases since the roadmap has difficulty capturing the connectivity of the environment. At the other end, a dense grid requires more time to be computed. Results show that random sampling improves the runtime of PUMMA since it effectively captures the connectivity of the environment.

Fig. 14 shows the results when varying $\text{COST}(q, q')$ (Eqn. 12) which defines the cost of each roadmap edge $(q, q')$. The first alternative definition uses the minimum clearance of $q, q'$ as opposed to the product of the clearances in the denominator. The second alternative definition uses a non-symmetric cost function with respect to the depth, assuming that it is more difficult for the AUV to go down. In such cases, when
\(q'_d > q_d\), the original cost is multiplied by \(1 + (q'_d - q_d)/d_{\text{max}}\), where \(d_{\text{max}}\) is the maximum depth. The third alternative definition assumes it is more difficult to go up and so multiplies the original cost by \(1 + (q_d - q'_d)/d_{\text{max}}\) when \(q_d > q'_d\). Results show that PUMMA works well for all these cases.

5) Impact of the Parameters: Fig. 15 shows the results when varying \(\alpha\) (heuristic strength) and \(\beta\) (selection penalty) parameters which are used to determine the weights associated with the equivalence classes (Eqn. 20). A small value of \(\alpha\) reduces the impact of the heuristic which makes it more difficult to guide the motion-tree expansion toward an accepting automaton state. A large value increases the greediness of the approach, making it less likely to explore alternative navigation routes. Similar observations can be made for the impact of the selection penalty \(\beta\). Overall, the results show that PUMMA works well for a wide range of parameter values.
6) Impact of the Ocean Currents: Fig. 16 shows the results when scaling the drift vectors representing the ocean currents. Without scaling, the average magnitude of the drift caused by the ocean currents is 0.11 for the river scene. As expected, the runtime increases when the ocean currents become stronger since it is more difficult to follow the navigation routes. Overall, the results show that PUMMA can take into account the drift caused by the ocean currents and effectively plan collision-free and dynamically-feasible motion trajectories that satisfy complex mission specifications.

![Fig. 16. Results on runtime and solution length obtained by the approach when scaling the drift vectors representing the ocean currents. The runtime plot uses logscale for the y-axis with the label showing the actual value rather than its logarithm. Bars indicate the standard deviation. Results are shown for the river scene with $\phi_{\text{GroupC}}^{15}$ as the mission.]

E. Field Experiment

A field experiment with an OceanServer Iver2 AUV [8] was conducted at the NRL Chesapeake Bay Detachment. The AUV was given the mission of reaching several regions in succession while avoiding collisions with artificially imposed obstacles resembling a maze. The goal regions were on the surface while the obstacles extended from the surface down to the maximum depth. In this way, the AUV would be forced to navigate inside the maze in order to avoid the obstacles and reach the goal regions. The approach, in a matter of less than five seconds, planned a collision-free and dynamically-feasible trajectory that reached the goal regions in succession, as required by the mission specification. To further facilitate the execution of the planned trajectory, the framework broke it down into a series of waypoints spaced at 30m. Every fourth waypoint was designated as a surface waypoint resulting in 120m underwater tracks. The AUV used its on-board controllers to follow the planned trajectory. The length of the planned trajectory was 6.2km and its execution took 69 minutes. The AUV maintained an average speed of 1.5m/s.

Fig. 17 shows the planned trajectory and the actual trajectory of the AUV. The plot for the actual trajectory is obtained from on-board data which the vehicle stores as it moves along. Table I provides a
Fig. 17. Field experiment with an OceanServer Iver2 AUV conducted at the NRL Chesapeake Bay Detachment. The figure shows the planned trajectory (blue), the actual trajectory executed by the AUV (magenta), the goal regions (yellow squares), and the forbidden regions (red). The AUV started at the bay near region $R_5$ and executed a sequencing mission, i.e., reach $R_1, \ldots, R_5$ in order. Planning time was less than five seconds, the execution time was 69 minutes, and the length of the trajectory was 6.2 km.

<table>
<thead>
<tr>
<th>Median</th>
<th>Mean</th>
<th>Std</th>
<th>Quartile 1</th>
<th>Quartile 3</th>
<th>Percentage of Distance Traveled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.52m</td>
<td>1.64m</td>
<td>2.22m</td>
<td>0.28m</td>
<td>2.91m</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

TABLE I  
Error between the planned and actual trajectory calculated based on GPS data obtained during surfacing events.

quantitative comparison between the planned and the actual trajectory. Since highly accurate navigational values are available while the AUV has GPS connection, only these values are considered in the error calculation. In addition to the median, mean, first and third quartiles, a common measure in underwater navigation is the error as the percentage of the distance traveled. This quantity is computed as $\frac{100}{d} \sum_{i=1}^{k} ||s_i - a_i||$, where $d$ is the total distance traveled, $k$ is the number of surface waypoints, $s_i$ is the $i$-th surface waypoint, and $a_i$ is the actual point where the AUV surfaced while navigating to the $i$-th surface waypoint. The results show strong agreement between the planned and the actual trajectory. These results provide promising validation that the approach is capable of quickly generating feasible motions that can be followed by the vehicle in order to successfully accomplish complex missions.

VI. DISCUSSION

This paper presented an effective approach for planning collision-free, dynamically-feasible, and low-cost trajectories that enable a AUV to accomplish missions given as regular languages. By using regular
languages the approach seeks to free users from the burden of unnatural low-level commands and instead allow them to describe missions at a high level in terms of desired objectives. Closure properties of regular languages facilitate the construction of hierarchical missions by composing simpler ones via union, intersection, concatenation, complementation, reversal, quotient, and numerous other operations.

The approach presented a successful and tight coupling between high-level mission planning and low-level sampling-based motion planning. A key aspect of the approach was the use of a navigation roadmap and its coupling with the automaton for the regular language in order to effectively guide the expansion of a motion tree along short and safe navigation routes. The partition of the motion tree into equivalence classes enabled the approach to reason effectively about where it needs to spend its computational resources in order to quickly complete the mission. The use of penalties enabled the approach not to become stuck by discovering alternative routes to expand the motion tree.

Experiments with a variety of scenes and missions of increasing complexity, taking into account the vehicle dynamics and the drift caused by time-varying ocean currents, demonstrated the effectiveness and scalability of the approach. Comparisons to related work showed significant improvements both in terms of the runtime and solution length. Field experiments provided promising validation that the collision-free and dynamically-feasible trajectories planned by the approach can be followed by the actual vehicle.

A direction for future work includes integration with high-level reasoners, such as goal driven autonomy [36], in order to enable the approach to quickly respond to unexpected events. Another direction is integration with the sensor data gathered during execution in order to autonomously adapt to changing environmental and contextual conditions. Risk-aware planning [14] can potentially be integrated with the approach in order to reduce the risk of collisions by accounting for the navigational and current-prediction uncertainty. This integration could become particularly useful when dealing with strong currents. Applications in mine countermeasures scenarios for reacquisition and identification can also be considered. As the approach can accommodate general vehicle dynamics, it could potentially be applied in other settings such as planning motions for aerial vehicles while taking into account the wind.

Theoretical analysis can focus on showing probabilistic completeness, which guarantees that a solution, when it exists, will be found with probability approaching one as time goes to infinity. Probabilistic completeness can be derived by extending the arguments used to show the probabilistic completeness of sampling-based motion planning. In fact, given a feasible word \( \sigma = (\pi_{i_1}, \ldots, \pi_{i_k}) \), by the probabilistic completeness of sampling-based motion planning it can be argued that the motion tree will eventually reach
π₁, and from there π₂, and so on. Since the approach will eventually explore every word σ that satisfies the mission specification, it will eventually find a solution when it exists. We also plan to investigate the asymptotic optimality of the approach in terms of the solution cost.

ACKNOWLEDGMENTS

The authors thank Dr. Jeffrey Schindall for assistance with the field experiments.

REFERENCES


*James McMahon* is a Ph.D. candidate in computer science at Catholic University of America, working under the direction of Dr. Erion Plaku. His research focuses on motion and mission planning for underwater vehicles. He received the B.S. and M.S. degrees in mechanical engineering from Catholic University of America in 2009 and 2012, respectively. James is also a Mechanical Engineer at the Naval Research Laboratory in the Acoustics Division where he works toward enhancing the autonomy of underwater vehicles.
Erion Plaku is an assistant professor in the Department of Electrical Engineering and Computer Science at The Catholic University of America in Washington, DC. He earned his Ph.D. from Rice University, Houston, TX. He was a postdoctoral fellow in the Laboratory for Computational Sensing and Robotics at The Johns Hopkins University, Baltimore, MD. His research focuses on motion planning and discrete planning for automatic or human-machine cooperative task performance in complex domains, such as mobile robotics, autonomous underwater vehicles, manipulation, hybrid systems, and robotic-assisted surgery.