# Probabilistic Specification Learning for Planning with Safety Constraints

Kandai Watanabe[1], Nicholas Renninger[2], Sriram Sankaranarayanan[1], and Morteza Lahijanian[1].

***Abstract*— This paper proposes a framework for learning task specifications from demonstrations, while ensuring that the learned specifications do not violate safety constraints. Furthermore, we show how these specifications can be used in a planning problem to control the robot under environments that can be different from those encountered during the learning phase. We formulate the specification learning problem as a grammatical inference problem, using probabilistic automata to represent specifications. The edge probabilities of the resulting automata represent the demonstrator's preferences. The main novelty in our approach is to incorporate the safety property during the learning process. We prove that the resulting automaton always respects a pre-specified safety property, and furthermore, the proposed method can easily be included in any Evidence-Driven State Merging (EDSM)-based automaton learning scheme. Finally, we introduce a planning algorithm that produces the most desirable plan by maximizing the probability of an accepting trace of the automaton. Case studies show that our algorithm learns the true probability distribution most accurately while maintaining safety. Since, specification is detached from the robot's environment model, a satisfying plan can be synthesized for a variety of different robots and environments including both mobile robots and manipulators.**

## I. Introduction

Autonomous robots make decisions against varying environments while maintaining key safety properties in domains such as deep-sea and space exploration; assistive and service domains (e.g., human-shared assembly lines); and surgical robotics. As robots are getting more autonomous, they are also expected to perform more complex tasks. Often times, however, the *explicit and precise specifications* of such tasks are unavailable, or too cumbersome to be provided by non-experts. However, these tasks can be easily demonstrated through various means – demonstrations through human operation or data collected from past manual operations. Given such demonstrations, the robot needs to be able to infer the task specification, so that it may autonomously satisfy the specification even against varying environments. In this regard, we identify four key challenges: (a) identifying a rich specification formalism that permits efficient and precise learning algorithms from the given demonstrations; (b) ensuring that key safety properties are satisfied by the resulting specification; (c) capturing "operator preferences" or "hidden costs" from the demonstrations; and (d) using the specifications to guide autonomous behavior under environments that may be different from those under which the demonstrations were provided. In this paper, we propose solutions to all four problems using the framework of probabilistic automata learned using grammatical inference. We focus on *safety-property constrained learning*: wherein the final learned specification must satisfy the safety properties. Our proposed learning approach ensures that the safety properties constrain the intermediate steps of the learning algorithm as opposed to an "after-the-fact" approach wherein the safety properties are intersected with the final specifications to rule out any unsafe behavior. We show that our safety-enabled learning approach clearly outperforms the after-the-fact approach.

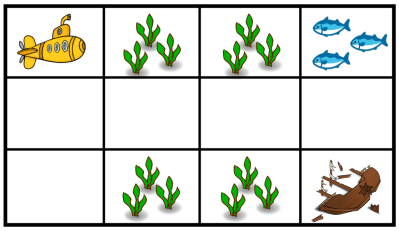


Fig. 1: Schematic of an autonomous deep-sea science mission.

For example, consider an underwater robot in a deep-sea science mission as schematically illustrated in Figure 1. The scientists want the robot to explore a shipwreck on the seafloor, study the behavior of a school of fish, while keeping away from the dangerous coral reefs. The goal of this work is to enable autonomous execution of this task by just exposing the robot to demonstrations from tele-operations or from related past missions. From these demonstrations, the robot should be able to infer the task and generate the necessary plan to accomplish it. We call this problem *Specification Learning from Demonstrations*, which can be regarded as a new form of *Learning from Demonstration* (LfD) [1].

Many LfD studies focus on either learning a policy or a reward structure [1], [2]. For policy learning, techniques such as *reinforcement learning* (RL) [3] and Dynamic Movement Primitives [4], [5] are typically used to learn a function that maps agent states to actions. In reward learning, a scalar reward function that maps agent states to rewards is learned via, e.g., *inverse reinforcement learning* (IRL) [6]–[9], to later train a policy on an agent. While very powerful, these methods learn a function that is specific to the environment (and robot) model used during training and hence are fragile to the changes to those models. More importantly, those methods are restricted to Markovian tasks. For example, the

This work was supported in part by the University of Colorado Boulder Autonomous Systems Interdisciplinary Research Theme; NSF award numbers 1932189 and 1836900; JASSO Overseas Graduate Fellowship and Keio University Global Fellowship.

[1]Authors are with the Departments of Computer Science and Aerospace Engineering Sciences at University of Colorado Boulder, Boulder, Colorado. {first.lastname}@colorado.edu

[2]Author is with MITRE Corporation. This work was done while he was a student in the Dept. of Aerospace Engineering Sciences at University of Colorado Boulder. nicholas.renninger@colorado.edu

robot task in Figure 1, which requires a visit to both the shipwreck and school of fish in any order, is non-Markovian. To achieve it, the robot has to maintain a memory of previous locations to decide the next location to visit. Hence, it is important to enable LfD for non-Markovian tasks, but such an extension is nontrivial for the existing LfD methods.

An alternative approach to expressing tasks is to use formal languages such as *linear temporal logic* (LTL) [10], which is widely used in formal verification and increasingly employed in robotics in recent years, e.g., [11]. Such languages enable formal expression of rich missions, including non-Markovian tasks [12] as well as *liveness* ("something good eventually happens") and *safety* ("something bad never happens") requirements. Other important benefits of formal languages is in their ease of interpretability and flexibility to compose multiple specifications. Such benefits have even led to their use in RL, e.g., [13]–[15]. Nevertheless, writing correct formal specifications requires domain knowledge.

In recent years, a new line of research has emerged with a focus on learning formal specifications from data [12], [16]–[19]. Most work has been concerned with learning temporal logic formulas with the purpose of classification and prediction from user data (in the supervised learning sense) [17], [18] or interpretation and planning for tasks [19]. Those studies restrict the exploration problem to a set of formula templates provided *a priori*. Recent work [12] overcomes this restriction by iterating over all combinations of formulas. The method is based on maximum a posterior learning and can account for noisy samples. It however is slow due to the large space of exploration for formulas. Another important issue with formula learning methods for the purpose of planning is that they typically need to be translated to an automaton, which could lead to the *state-explosion* problem [10], [11]. Work [20] overcomes this issue by directly learning a *Deterministic Finite Automaton* (DFA). They however assume the structure of the DFA is known and only learn the transitions between the DFA states while an oracle labels each sample with DFA states.

In this work, we propose a new approach to task specification learning where we infer formal task specifications as probabilistic automata. Our approach used ideas from the field of *grammatical inference* (GI) [21], by modifying existing "evidence-driven state merging" algorithms in GI to learn the specification as a *Probabilistic Deterministic Finite Automaton* (PDFA). We further extend this method to incorporate safety properties during the learning process, so that all runs of the final PDFA satisfy the safety properties. Furthermore, we propose a planning algorithm with the inferred PDFA that generates the most preferred plan to achieve the task for any robot that can be abstracted to a deterministic transition system.

To the best of our knowledge, this is the first work that employs PDFA as a learning model for task specifications in robotics. The key technical contributions include: (a) the derivation of the safety guaranteed PDFA learning algorithm that is compatible with any EDSM techniques; (b) a planning algorithm with a learned PDFA that can handle varying environments; and (c) a set of experiments that show the efficacy of the proposed algorithms in both mobile and manipulator robots. We also provide a comparison case study against [12] to highlight the similarities and differences between the two approaches.

## II. Problem Formulation

In this section, we describe and formalize our problem of task specification learning with preferences and safety constraints. Our aim is to use an interpretable learning model that we can use for planning for a robot to perform the task according to the user's preferences.

### A. Task Specifications and demonstrations

We assume the demonstrator has a task in mind that needs to be achieved in finite time. Our goal is to learn this task in the form of a deterministic finite state automaton (DFA) that reflects the overall goals of the demonstrator. Such automata accommodate a large class of tasks, including the ones that can be specified using co-safe LTL [22] and *LTL over finite traces* [23] formulas. Nevertheless, to avoid a trivial solution to the problem (e.g., a trivial specification that accepts all possible behaviors), we consider two important caveats.

We specify, on the side, a safety property specification that the robot must not violate. The safety property characterizes a potentially infinite set of negative examples for our learner consisting of those specifications that violate the safety property.

We want to encode the demonstrator's *preferences* between various ways of satisfying an intended specification. For instance, a demonstrator may prefer to avoid a collision with an obstacle by *steering left* preferentially since it may position the vehicle to avoid a future collision with less effort.

Below, we define the mathematical machinery needed to formalize this problem.

For a given task $'_{\text{task}}$, let $= fp_1;:::;p_kg$ be a set of atomic propositions (boolean predicates) that represent important facts about the world that is relevant to the task. The set of atomic propositions that is true at a state of the world is called a *symbol*. The set of all symbols is denoted by . Note that $= 2$ . A demonstration $!$ (also known as a *trace* or *word*) is a finite sequence of symbols, i.e., $! = !_1!_2:::!_n$, where $!_i \; 2$ for all $1 \quad i \quad n$.

From demonstrations, we want to learn task $'_{\text{task}}$. However, instead of directly learning it as a temporal logic formula, our goal is to learn it in the form of a DFA.

**Definition 1** (DFA)**.** *A deterministic finite automaton (DFA) is a tuple* $A = (Q; \; ; q_0; \; ; F)$*, where*

$Q$ *is a finite set of states,*
*is a finite set of input symbols,*
$q_0 \; 2 \; Q$ *is the initial state,*
$: Q \quad ! \; Q$ *is the transition function, and*
$F \quad Q$ *is the set of final or accepting states.*

The transition function can be also viewed as a relation $Q \quad Q$, where every transition is a tuple $(q; \; ; q^0) \; 2$

iff $q^0 = (q; \ )$, where $\ 2 \ $. A *run* of a DFA $A$ on trace $! = !_1!_2:::!_n$ is a sequence of states $z = z_0z_1:::z_n$, where $z_0 = q_0$ and $z_i = (z_{i\ 1}; !_i)$ for $i = 1;:::;n$. A run $z$ is called *accepting* if $z_n \ 2 \ F$. A trace $!$ is accepted by $A$ if it induces an accepting run. The set of all traces that are accepted by DFA $A$ is called the language of $A$ and is denoted by $L(A)$.

We call a demonstration $!$ valid if it is accepted by the DFA that represents $'_{\text{task}}$, i.e., achieves task $'_{\text{task}}$.

**Definition 2** (Valid Demonstration)**.** *A valid demonstration for task $'_{task}$ is a finite trace $! \ 2 \ $ such that $! \ 2 \ L(A_{'_{task}})$, where $A_{'_{task}}$ is the DFA that represents $'_{task}$.*

### *B. Safety Specification*

To express the safety constraints, we use safe LTL [22].

**Definition 3** (Safe Syntax)**.** *A syntactically safe LTL formula over $\ $ is recursively defined as*

$$' := p \mid :p \mid ' \_ ' \mid ' \wedge ' \mid X ' \mid G '$$

*where $p \ 2 \ $, $:$ (negation), $\_$ (disjunction), and $\wedge$ (conjunction) are boolean operators, and $X$ ("next") and $G$ ("globally") are temporal operators.*

Safe LTL formulas reason over infinite traces, but finite traces are sufficient to violate them [22]. We denote the set of finite traces that violate safety formula $'_{\text{safe}}$ by $L(:'_{\text{safe}})$. Hence, given valid demonstrations and safety formula $'_{\text{safe}}$, we want to learn a DFA $A_{'_{\text{task}}}$ such that it does not accept any trace that violates safety, i.e, $L(A_{'_{\text{task}}}) \setminus L(:'_{\text{safe}}) = ;$.

### *C. Preferences*

On the learned DFA, we also want to encode the preferences of the demonstrator. We assume that the preferences are correlated with the number of demonstrations of the same trace. Hence, preferences can be quantified as weights over traces and normalized over the entire language, which can be viewed as a probability distribution over $L(A_{'_{\text{task}}})$. The higher the probability of an accepting trace is, the more preferred it is to the demonstrator. Hence, our aim is to learn a probabilistic DFA (PDFA), which captures both the accepting traces and their corresponding probabilities.

**Definition 4** (PDFA)**.** *A probabilistic DFA (PDFA) is a tuple $A^{\mathrm{P}} = (A; \ _{\mathrm{P}}; F_{\mathrm{P}})$, where $A$ is a DFA, and $\ _{\mathrm{P}} : \ ! \ [0; 1]$ assigns a probability to every transition in $\ $ such that $\sum_{\ 2 \ } \ _{\mathrm{P}}(q; \ ; \ (q; \ )) = 1$ for every $q \ 2 \ Q$, and $F_{\mathrm{P}} : Q \ ! \ [0; 1]$ assigns a probability of terminating at each state, where $F_{\mathrm{P}}(q) = 0$ if $q \not\in F$.*

Consider trace $! = !_1!_2:::!_n$ and its induced run $z = z_0z_1:::z_n$ on PDFA $A^{\mathrm{P}}$. The probability of $!$ is given by

$$P(!) = \prod_{i=1}^{\ } \ _{\mathrm{P}}(z_{i\ 1}; !_i; z_i) \ \ F_{\mathrm{P}}(z_n):$$

We say $A^{\mathrm{P}}$ accepts $!$ iff $P(!) > 0$, and the demonstrator prefers $!$ over $!^0 \ 2 \ (2 \ \ )$ iff $P(!) > P(!^0)$. The language of $A^{\mathrm{P}}$ is the set of traces with non-zero probabilities, i.e.,

$$L(A^{\mathrm{P}}) = f! \ 2 \ (2 \ \ ) \ \ j \ P(!) > 0g:$$

Therefore, our goal becomes to learn a $A^{\mathrm{P}}_{'_{\text{task}}}$ that captures the task $'_{\text{task}}$, the demonstrator's preferences (probability distribution over traces), and never violates $'_{\text{safe}}$.

### *D. Robot Model and Plans*

Once a specification is learned, we want to synthesize a plan for a robot that can realize the specification. To do so, we assume that we are given an abstraction model of the robot as a deterministic transition system. Such an abstraction is commonly used and constructed in formal approaches to both mobile robotics [11], [24], [25] and robotic manipulators [26], [27].

**Definition 5** (DTS)**.** *A deterministic transition system (DTS) is a tuple $T = (X; A; x_0; \ _T; \ ; L)$, where*

*$X$ is a finite set of states,*
*$A$ is a finite set of controls or actions,*
*$x_0 \ 2 \ X$ is the initial state,*
*$\ _T : X \ \ A \ ! \ X$ is the (partial) transition function,*
*$\ $ is a finite set of atomic propositions (predicates), and*
*$L : X \ ! \ 2^{\ }$ is a labeling function that maps each state to the set of predicates that are true at that state.*

A *plan* $\ = \ _0 \ _1::: \ _{n\ 1}$ is a sequence of actions, where $\ _i \ 2 \ A$ for all $0 \ \ i \ \ n \ \ 1$. By executing $\ $, the robot generates a *trajectory* $s = s_0s_1:::s_n$, where $s_0 = x_0$ and $s_{i+1} = \ _T(s_i; \ _i)$. A *valid plan* is plan $\ $ that respects the transition function $\ _T$, i.e, $\ _T(s_i; \ _i)$ exists for all $0 \ \ i \ \ n \ \ 1$. We denote the set of all valid plans by $\ $.

The observation trace of the above trajectory is the sequence of observed labels, i.e., $\ = L(s_0)L(s_1):::L(s_n)$. We refer to the set of all observation traces that a robot can generate as the language of $T$, i.e., $L(T) = f \ \ j \ 2 \ g$.

In the planning problem, we are interested in a plan $\ $ that generates an observation trace $\ $ that achieves task $'_{\text{task}}$ and is the most preferred behavior, i.e.,

$$= \arg \max_{\ 2 L('_{\text{task}}) \setminus L(T)} P(!):$$

### *E. Problem*

We are now able to formally define our problem.

**Problem.** *Given demonstrations $\ = [!_i]_{i=1}^{n_\Omega}$ that are sampled from a hidden task specification DFA $A_{'_{task}}$ according to some preferences (probability distribution) of the demonstrator and a safety specification $'_{safe}$,:*

1) *learn $'_{task}$ and the demonstrator's preferences as a PDFA $A^{\mathrm{P}}_{'_{task}}$ such that safety is never violated, i.e, $L(A^{\mathrm{P}}_{'_{task}}) \setminus L(:'_{safe}) = ;$;*
2) *furthermore, given a DTS robot model $T$, compute a plan for the robot $T$ that generates the most preferred behavior that satisfies $'_{task}$, i.e, generates the trace with the highest probability in $L(A^{\mathrm{P}}_{'_{task}}) \setminus L(T)$.*

For Problem 1, we use grammatical inference [21] while incorporating the safety property during the learning process,

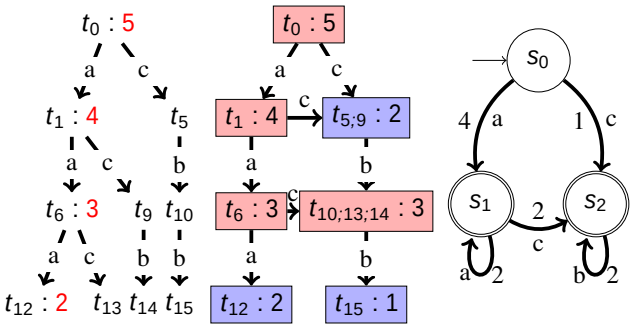



Fig. 2: Schematic illustration of evidence-driven state merging (EDSM) algorithm. **(Left)** A frequency prefix tree acceptor (FPTA) is constructed from the given demonstrations with frequencies greater than 1 shown in red; **(Middle)** intermediate automaton as states are merged according to criteria that differ across various algorithms with frequencies shown at each node; and **(Right)** the final frequency DFA (FDFA) that is learned is shown in red.

as described in Section III. We use this PDFA to solve Problem 2 as detailed in Section IV.

## III. Safety Guaranteed PDFA Learning

In this section, we explain how a PDFA can be learned from demonstrations and present our method that can embed safety specification to guarantee safety on the outcome. We first show a general PDFA learning algorithm, and then we describe our algorithms to incorporate safety.

### A. Grammatical Inference: PDFA Learning

PDFA learning has been extensively studied as part of *grammatical inference* (GI) with existing algorithms such as ALERGIA, DSAI, and MDI, that can learn PDFAs from unlabeled demonstrations [21]. These algorithms are all based on a principle called *evidence-driven state-merging* (EDSM). At a high level, EDSM approaches find an appropriate structure for an automaton $A^{\mathsf{P}}$ and simultaneously estimate the probability distribution parameters $F_{\mathsf{P}}$ and $_{\mathsf{P}}$ given a set of sample traces. This is achieved by first constructing a large (prefix) tree from the samples, and repeatedly merging the states of the tree in order to form an automaton that is as simple as possible while continuing to accept the sample traces from the demonstration. The various algorithms (e.g., ALERGIA and MDI) differ on what states are merged.

Figure 2 shows a general scheme for an EDSM-based algorithm for learning a PDFA. The initial step is to construct a *frequency prefix tree acceptor* (FPTA) from the traces in (Fig. 2-Left) and then incrementally merge states of the FPTA, two at a time, based on a *compatibility* criterion that varies depending on the actual algorithm. As two states are merged, so are their subtrees in the FPTA (Fig. 2-Middle). The nodes of the intermediate automata are variously colored red/blue using a coloring scheme to influence how states are selected for merging. Furthermore, algorithms also maintain frequencies alongside the nodes based on the number of demonstration traces that reach a particular node. These frequencies are also combined during the state merging process. The final result is a *frequency DFA* (FDFA) wherein frequencies along edges indicate how often they are taken by a demonstration (Fig. 2-Right). The frequencies of all outgoing edges are normalized to yield a distribution.

The various PDFA learning algorithms such as ALERGIA or MDI differ on how they implement the *compatibility* check for whether two given nodes can be merged. For instance, the ALERGIA algorithm implements a statistical test based on frequencies to compare if two states are compatible, whereas the MDI approach first temporarily merges two states and their subtrees, while accepting the merge if a metric computed on automaton after the merge is smaller than that before the merge. We assume that the basic PDFA learning algorithm as a given, and our goal is to learn while respecting a safety property.

### B. Learning with Safety Specification

We now consider two different approaches for learning with a safety specification. The first method is a *post-processing* technique that simply runs the PDFA learning algorithm on the given demonstration traces and then subsequently intersects the resulting PDFA with the automaton for the safety property. The second method incorporates the safety specification during the learning process by modifying the EDSM algorithm. In particular, the merges are defined so that the result continues to satisfy the safety specifications.

*1) Post-process Algorithm:* From $'_{\text{safe}}$, we first construct a DFA $A_{:'_{\text{safe}}}$ that accepts precisely all those traces that violate the safety property [22]. Then, by complementing $A_{:'_{\text{safe}}}$, we obtain $A_{\text{safe}} = (Q^s; \ ; q_0^s; \ ^s; F^s)$ that accepts all the traces that do not violate $'_{\text{safe}}$. Let $A^{\mathsf{P}} = (Q; \ ; q_0; \ ; F; \ _{\mathsf{P}}; F_{\mathsf{P}})$ be the PDFA learned from the given demonstration traces without considering the safety property. We intersect the languages of $A^{\mathsf{P}}$ and $A_{\text{safe}}$ by constructing a product automaton $A^{\mathsf{P}}_{\text{safe}} = A^{\mathsf{P}} \ A_{\text{safe}} = (Q_{\text{safe}}; \ ; q_{0;\text{safe}}; \ _{\text{safe}}; F_{\text{safe}}; \ _{\mathsf{P};\text{safe}}; F_{\mathsf{P};\text{safe}})$, where $Q_{\text{safe}} = Q \quad Q^s$, $q_{0;\text{safe}} = (q_0; q_0^s)$; $F_{\text{safe}} = F \quad F^s$;

$$_{\text{safe}}((q;q^s); \ ) = (q^0;q^{s0}) \text{ if } q^0 = \ (q; \ ) \wedge q^{s0} = \ ^s(q^s; \ );$$

$F_{\mathsf{P};\text{safe}}((q;q^s)) = F_{\mathsf{P}}(q)$, and

$$_{\mathsf{P};\text{safe}}((q;q^s); \ ;(q^0;q^{s0})) = \begin{cases} \frac{_{\mathsf{P}}(q; \ ;q')}{N(q;q^s)} & \text{if } (q^0;q^{s0}) = \ _{\text{safe}}((q;q^s); \ ) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $N(q;q^s)$ is the normalizing function such that $\sum_{(\ ;q'_{\text{safe}}) \in (\ Q_{\text{safe}})} {}_{\mathsf{P};\text{safe}}((q;q^s); \ ;q^0_{\text{safe}}) = 1$. The resulting PDFA is guaranteed to be safe due to the intersection of languages. However, this method of pruning (imposing safety) as a post-process step alters the probability distributions over the next-state transitions, since we remove the transitions that violate safety and renormalize the probability distribution at each state, as shown in (1). This overrides the probability distributions constructed by the original PDFA learning algorithm in an unpredictable manner. Therefore, while this method of imposing safety always succeeds, its probability distributions may not reflect the preferences embedded in the demonstrations accurately.

*2) Safety-Incorporated Learning Algorithm using "Pre-Processing":* Whereas the *post-processing* approach enforces safety after the PDFA is learned, the pre-processing approach guarantees that the intermediate results also preserve safety, hence preventing alterations to the probability distributions due to unsafety. The main idea is to build the PDFA that generalizes the demonstrated traces but carries along with it information about how the generalization satisfies the safety property $'_{\text{safe}}$ at the same time in the form of a simulation relation with $A_{\text{safe}}$.

**Definition 6** (Simulation Relation)**.** *A simulation relation $R$ between two automata $A$ and $B$ is a relation between their states, $R \quad Q_A \quad Q_B$*

*(a) Initial states of $A$ relate to the initial states of $B$;*

*(b) If pair $(s;t) \ 2 \ R$, where $s \ 2 \ Q_A$ and $t \ 2 \ Q_B$, and automaton $A$ can transition from $s$ to $s^0 \ 2 \ Q_A$ on symbol , then there must exist a state $t^0 \ 2 \ Q_B$ such that automaton $B$ transitions from $t$ to $t^0$ on the same symbol and $(s^0; t^0) \ 2 \ R$;*

*(c) For each $(s;t) \ 2 \ R$, if $s$ is final in $A$ then $t$ must be final in $B$.*

**Theorem 1.** *Let $R$ be a simulation relation between automata $A$ and $B$. It follows that $L(A) \quad L(B)$.*

The proof simply shows by induction that for any accepting run corresponding to an input trace $!$ in automaton $A$ from the initial state to a final state, there exists an accepting run in $B$ for the same trace $!$ from its initial state to the final state. The relation $R$ allows us to construct such a run.

The key idea behind the *pre-process* approach is to maintain a simulation relation between the FDFA and safety automaton $A_{\text{safe}}$ at all intermediate states. The key is to restrict the merging of states so that we can guarantee that a simulation relation between the original automaton and $A_{\text{safe}}$ before merging can be modified to yield a simulation relation between the merged automaton and $A_{\text{safe}}$ afterwards.

Formally, we build the safety FPTA by augmenting the initial FPTA so that each state is now a tuple of the form $(t_j; s_k)$ wherein $t_j$ is a node in the original FPTA and $s_k$ is the state in $A_{\text{safe}}$ reached when the prefix that leads upto the state $t_j$ is run through $A_{\text{safe}}$. Thus, we ensure that every branch not only corresponds to a demonstration but also to a valid trace in $A_{\text{safe}}$.

Let $R$ be a relation between states of the FPTA and $A_{\text{safe}}$ that contains all nodes $(t_j; s_k)$ in the safety FPTA.

**Lemma 1.** *Assuming no demonstration trace violates the safety property $'_{safe}$, then $R$ is a simulation relation between the initial FPTA and the automaton $A_{safe}$.*

We can represent any intermediate FDFA state in the form $(T; s)$ wherein $T$ is a set of states from the initial FPTA, and $s$ is state in $A_{\text{safe}}$. Next, we modify the EDSM approach to allow a merge between two states $(T_i; s_k)$ and $(T_j; s_l)$ only if $s_k = s_l$. The result of the merge creates a state $(T_i \ [ \ T_j; s_k)$.

**Lemma 2.** *Let $A_1$ and $A_2$ be the automata before and after an EDSM merge that is compatible with respect to the $A_{safe}$ states. Let $R_1$ be the relation between the states of $A_1$ and those of $A_{safe}$ that is a simulation relation. We can construct a simulation relation $R_2$ between the states of $A_2$ and $A_{safe}$.*

Combining Lemmas 1 and 2, we conclude by induction on the number of merging steps that the final resulting PDFA must have a simulation relation to the safety automaton $A_{\text{safe}}$. Since we have a simulation relation, we conclude that the language of the final resulting FDFA and PDFA are contained in the that of $A_{\text{safe}}$, i.e., the resulting PDFA does not accept a trace that violates $'_{\text{safe}}$.

## IV. Planning with PDFA

Once a task is learned as a PDFA, we are interested in synthesizing a plan that not only achieves the task but also respects the demonstrator's preferences (Problem 2). Here, we introduce a method to do this for any robot with a DTS (abstract) model. We first construct a (product) graph that captures all the ways the robot can achieve the task by composing the DTS with the learned PDFA. Then, we reduce the optimal planning problem to a search on this graph.

The plans are computed over the product graph $P = A^{\mathrm{P}} \ T$, where $T$ is a DTS obtained by augmenting $T$ with a new initial state $x_0$ with a transition to $x_0$. Formally, $T = (X; A; x_0; \ _T; \ ; L)$, where $X = X \ [ \ fx_0g$, and $_T(x; a) = x_0$ if $x = x_0$, otherwise $_T(x; a) = \ _T(x; a) \ 8a \ 2 \ A$. This augmentation allows to correctly observe the label of $x_0$ and assign the edge weights in $P$ according to the probabilities in $A^{\mathrm{P}}$ through the product rule below. Given learned $A^{\mathrm{P}} = (Q; \ ; q_0; \ ; F; \ _{\mathrm{P}}; F_{\mathrm{P}})$ and $T$, we construct *weighted product graph* $P = (Q^P; q_0^P; E; W)$, where

$Q^P = (Q \quad X) \ [ \ fq_t^P g$ is a set of states, where $q_t^P$ is the terminal state,
$q_0^P = (q_0; x_0)$ is the initial state,
$E \quad Q^P \quad Q^P$ is a set of edges, and
$W : E \ ! \ \mathbb{R}^{<0}$ is a weight function that assigns to each edge $e \ 2 \ E$ a weight according to its probability in $A^{\mathrm{P}}$.

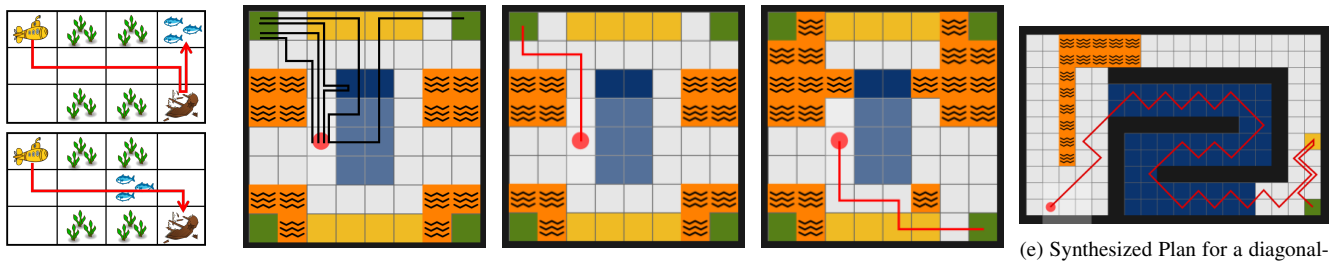
The constructions of $E$ and $W$ are as follows.

Edge $e = ((q; x); (q^0; x^0)) \ 2 \ E$ if $q^0 = (q; L(x^0))$ and $x^0 = \ _T(x; a)$ for some $a \ 2 \ A$. Then, $W((q; x); (q^0; x^0)) = \log( \ _{\mathrm{P}}(q; L(x^0); q^0))$.
Edge $e = ((q; x); q_t^P)) \ 2 \ E$ if $F_{\mathrm{P}}(q) > 0$. Then, its weight $W((q; x); q_t^P) = \log(F_{\mathrm{P}}(q))$.

Product graph $P$ captures the constraints of both the robot and task along with the demonstrator's preferences. Let $= (q_0; x_0)(q_1; x_0) : : : (q_n; x_{n \ 1})q_t^P$ be a path over $P$. The projection of this path (with the deletion of $q_t^P$) onto $A^{\mathrm{P}}$ is an accepting run with the trace $! = !_1 : : : !_n$. The projection of on $T$ is the robot trajectory that generates the accepting trace $!$. The probability of this trace is in fact the inverse logarithm of the total weight of , i.e.,

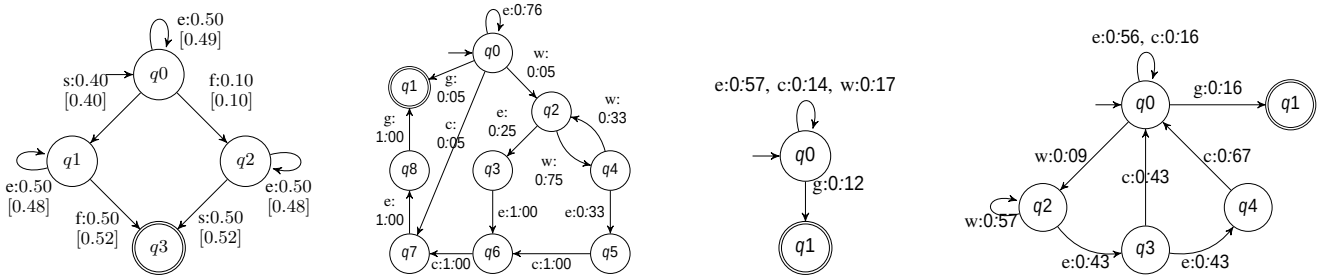
$$\sum_{i=0} W( \ _i; \ _{i+1}) = \sum_{i=0}^{n-1} \log( \ _{\mathrm{P}}(q_i; q_{i+1})) + \log(F_{\mathrm{P}}(q_n))$$

$$= \log \prod_{i=0}^{n-1} \ _{\mathrm{P}}(q_i; q_{i+1}) \quad F_{\mathrm{P}}(q_n) \quad = \log(P(!)):$$



(a) Non-Markovian Task (b) Five demos from [12] (c) Synthesized Plan (d) Synthesized Plan (e) Synthesized Plan for a diagonal-moving robot

Fig. 3: Various environments and robots considered for the case studies. (a) Learning and planning for the non-Markovian task. (b) Environment and demonstrations from [12]. (c)-(e) Synthesized plans (shown in red) based on the learned task from (b).




(a) PDFA for Fig. 3a (b) Vanilla, $\alpha = 0.4$, Fig. 3b. (c) Vanilla, $\alpha = 4$, Fig. 3b. (d) Pre-process, $\alpha = 4$, Fig. 3b

Fig. 4: The task specification and the learned PDFAs for the scenarios in Fig. 3a and 3b. Each letter represents a symbol with a single atomic proposition s={$ship$}, f={$fish$}, b={$blue$}, c={$carpet$}, g={$green$}, p={$purple$}, and $e = \emptyset$. The termination probability $F_P$ of double-edged states is 1 and 0 at all other states.

Therefore, to compute a robot plan that produces an accepting trace with the highest probability in $L(A^P) \setminus L(T)$, it is enough to find a path on $P$ that reaches the terminal state $q_t^P$ with the maximum total weight, i.e.,

$$\arg \max_{\rho \in L(A^P) \setminus L(T)} P(\rho) = \text{PROJ}\left(\arg \max_{\rho} \sum_{i=0}^{j} W(s_i, s_{i+1})\right),$$

where is the set of paths of $P$ that terminate in $q_t^P$, and PROJ is the projection operator that maps to its corresponding trace $\rho$. Note that the all the edge weights of $P$ are negative, hence, it is a simple graph search problem that can be performed using algorithms such as Dijkstra's.

## V. Case Studies and Evaluations

We illustrate the performance of the proposed algorithms in five case studies. Our implementation of the EDSM algorithm is based on the MDI method that is used in the *flexfringe* library [28]. We call the basic algorithm the *Vanilla* algorithm. All the case studies were run on a MacBook Pro with 2.3 GHz Dual-Core Intel Core i5 and 16 GB RAM. Videos of all case studies are available to view[1].

### A. Learning and Planning for Non-Markovian Tasks

In this case study, we consider the robotic scenario in Figure 1. The task is to visit both the school of fish and the shipwreck in any order and always avoid coral reefs. The preference is to visit the shipwreck first. A PDFA representation of this specification is shown in Figure 4a.

To learn this task, we sampled 1000 traces from this PDFA on the gridworld environment in Figure 1. From these demonstrations, the Vanilla algorithm learned a PDFA with the same exact structure as the true PDFA and probabilities within 0.02 of the true values (in square brackets in Fig. 4a).

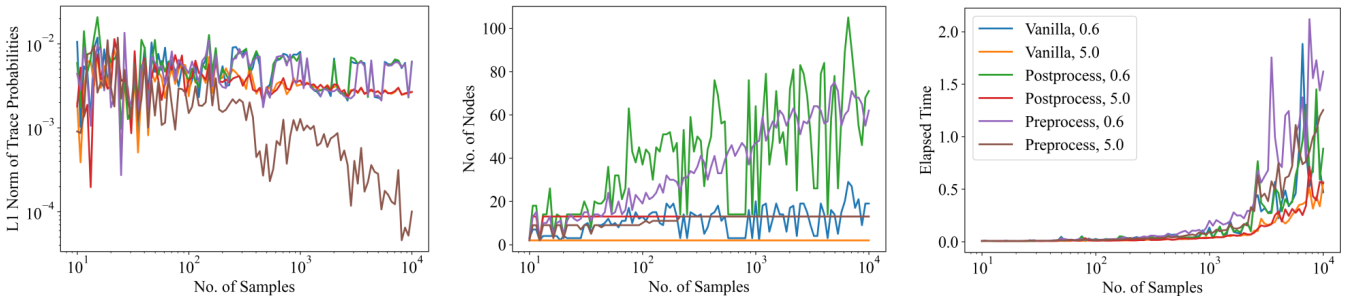
As the PDFA shows, our method correctly learned the non-Markovian task of visiting both the shipwreck and the school of fish in both orders and favors going to the shipwreck first. Using this PDFA, our planner generated the robot trajectory shown in Figure 3a (top), which correctly visits the shipwreck first and then the school of fish. Next, we changed the environment by moving the location of the fish to be on the robot's way to the shipwreck as shown in Figure 3a (bottom). This figure also shows the synthesized plan in this environment using the same learned PDFA. Notice that the robot is not visiting the shipwreck first due to environmental constraint. Instead, it visits the fish and then the shipwreck, which is also a correct behavior. This generality is the strength of learning the specification rather than learning a policy that is strongly dependent on the environment.

### B. Learning from Small Number of Samples with Safety

In this case study, we consider the environment and five demonstrations depicted in Figure 3b taken from [12] to learn the specification in a form of a PDFA as a comparison to the approach in [12], which is based on learning specification formulas. In this gridworld, each color represents an object, where orange is *lava*, blue is *water*, yellow is a *drying carpet*, white is an *empty* space, and green is a *charging station*. The task is to reach a charging station. However, the robot should not charge while it is wet. That is, once it gets wet (goes to water), the robot has to dry at the *drying carpet*.

*1) Small number of samples:* We first used the Vanilla algorithm with the five demonstrations, which learned the

[1] https://youtu.be/TU8MhPBDBBs




(a) L1 norm error (log scale) (b) Number of nodes (c) Computation time [s]

Fig. 5: Performance analysis for the proposed algorithm. Plots in (a)-(b) use the same legend as (c).

PDFA in Figure 4b. Note, in the learned PDFA, region green (*charging station*) must always be observed to reach the final state. This shows that the task of reaching the *charging station* is learned correctly. Next, on the right most branch of the PDFA, *carpet* is always observed when the robot gets wet. Again, the algorithm succeeded in learning the task of visiting *carpet* once the robots gets wet before reaching the *charging station*. One interesting observation is that the PDFA also learned that the robot has to go to the *charging station* in one step after leaving the *carpet*. This is in fact a bias in the samples since every shown demonstration that includes *carpet* has this property. If that is the intention of the demonstrator, then it is a correct behavior. If it is not, then it can be resolved by providing more samples.

Such one-step bias is not apparent in [12] because the "next" operator is not allowed in the syntax of the *language* they consider. In contrary, our method infers over regular language, which includes the next operator. Furthermore, in [12], it took 95 seconds to learn the specification from just 5 demonstrations whereas ours took less than 0.01 seconds.

*2) Hyperparamter choice and safety:* The PDFA in Figure 4b is the result of the Vanilla algorithm when the hyperparameter of is set to 0.4. It is a knob of how aggressive we allow the merges. Higher the value of is, the smaller the PDFA becomes. If we can tune the hyperparameter correctly, we can get a desirable result as described above. But, if we increase too much, some merges could induce unsafe behavior. Unwanted merges occur because the algorithm is simply trying to *minimize* the size of the structure. In fact, the question of how to choose a correct value for is an open problem. For = 4, the learned PDFA from the same demonstrations is shown in Figure 4c. This PDFA has no regards for safety and only requires to reach the *charging station*. We can mitigate this problem by embedding safety specification. We define the following safety formula:

$$'_{\text{safe}} = G : \text{lava} \wedge G(\text{water} \;!\; X('(:\text{charge};\text{carpet};\text{k}))$$

where ' is a formula recursively defined as: $'(a;b;k) = a \wedge (b\_X('(a;b;k \quad 1)))$ and $'(a;b;0) = a$, and is read, "visit $a$ for $k$ steps unless $b$ is visited". This formulas requires never going to *lava* and, if the robot enters *water*, it cannot *charge* unless it visits *carpet* or stays in *empty* for $k$ consecutive steps to get dry. We set $k = 10$ in all experiments.

From the same five demonstrations, we now learn PDFAs using the Post-process and Pre-process algorithms with = 4 subject to $'_{\text{safe}}$. The Post-process algorithm generates a large PDFA with 13 nodes and 36 edges since the safety DFA itself is large (12 nodes and 34 edges). Despite the size, it always guarantees no violation to $'_{\text{safe}}$. The PDFA generated by the Pre-process algorithm is shown in Figure 4d. It is small and correctly embeds both safety and liveness. Further, all the demonstrations are accepted by both learned PDFA. As for probabilities, the average L1 norm error was 1:65 $10^{-3}$ for the Post-process PDFA and 7:42 $10^{-5}$ for the Pre-process PDFA, indicating better performance by the Pre-process algorithm. The larger error in the probabilities of the Post-process PDFA is due to the composition with the safety DFA, which prunes away the unsafe traces in the learned PDFA, corrupting the learned probability distributions.

Next, we perform a thorough comparison of the learning methods by increasing the number of samples.

### C. Post-process versus Pre-process Algorithm

Here, the task is similar to the one above, but the goal is to quantitatively analyze and compare the performances of the proposed algorithms as the number of samples increases. We sampled demonstrations randomly from the true PDFA and used Post-process and Pre-process algorithms to learn PDFAs with hyperparameter values of = 0:6 (less aggressive merge) and = 5:0 (aggressive merge) to show the extreme results. We evaluated the resulting PDFAs with respect to the following metrics: L1 norm of the trace probability errors, number of states, and computation times. The results are shown in Figure 5 (all the plots share the same legend).

With respect to the error metric, the Pre-process algorithm (purple line for = 0:6 and brown line for = 5) consistently performs the best, followed by the Post-process and then the Vanilla algorithms. It indicates that the Pre-process algorithm is learning the correct distribution over the language of the target PDFA. As for the number of states and computation time metrics, the results indicate that Pre-process algorithm again performs better and faster than the others. From these results, we can say that the Pre-process algorithm is the best performing algorithm. Moreover, its output PDFA does not violate the safety across all the trials (checked but not shown in the figures).

### D. Planning for Various Robots in different Environments

From the learned PDFAs above, we picked one with a small L1 error norm. Then, using this PDFA, we planned