Compositional Reinforcement Learning from Logical Specifications

Kishor Jothimurugan  
University of Pennsylvania

Suguman Bansal  
University of Pennsylvania

Osbert Bastani  
University of Pennsylvania

Rajeev Alur  
University of Pennsylvania

Abstract

We study the problem of learning control policies for complex tasks given by logical specifications. Recent approaches automatically generate a reward function from a given specification and use a suitable reinforcement learning algorithm to learn a policy that maximizes the expected reward. These approaches, however, scale poorly to complex tasks that require high-level planning. In this work, we develop a compositional learning approach, called DIIRL, that interleaves high-level planning and reinforcement learning. First, DIIRL encodes the specification as an abstract graph; intuitively, vertices and edges of the graph correspond to regions of the state space and simpler sub-tasks, respectively. Our approach then incorporates reinforcement learning to learn neural network policies for each edge (sub-task) within a Dijkstra-style planning algorithm to compute a high-level plan in the graph. An evaluation of the proposed approach on a set of challenging control benchmarks with continuous state and action spaces demonstrates that it outperforms state-of-the-art baselines.

1 Introduction

Reinforcement learning (RL) is a promising approach to automatically learning control policies for continuous control tasks—e.g., for challenging tasks such as walking [11] and grasping [6], control of multi-agent systems [31, 22], and control from visual inputs [28]. A key challenge facing RL is the difficulty in specifying the goal. Typically, RL algorithms require the user to provide a reward function that encodes the desired task. However, for complex, long-horizon tasks, providing a suitable reward function can be a daunting task, requiring the user to manually compose rewards for individual subtasks. Poor reward functions can make it hard for the RL algorithm to achieve the goal; e.g., it can result in reward hacking [3], where the agent learns to optimize rewards without achieving the goal.

Recent work has proposed a number of high-level languages for specifying RL tasks [5, 29, 24, 34, 19]. A key feature of these approaches is that they enable the user to specify tasks compositionally—i.e., the user can independently specify a set of short-term subgoals, and then ask the robot to perform a complex task that involves achieving some of these subgoals. Existing approaches for learning from high-level specifications typically generate a reward function, which is then used by an off-the-shelf RL algorithm to learn a policy. Recent works based on Reward Machines [19, 35] have proposed RL algorithms that exploit the structure of the specification to improve learning. However, these algorithms are based on model-free RL at both the high- and low-levels instead of model-based RL. Model-free RL has been shown to outperform model-based approaches on low-level control tasks [10]; however, at the high-level, it is unable to exploit the large amount of available structure. Thus, these approaches scale poorly to long horizon tasks involving complex decision making.
We empirically evaluate our approach on a “rooms environment” (with continuous state and action spaces), where a 2D agent must navigate a set of rooms to achieve the specification, as well as a challenging “fetch environment” where the goal is to use a robot arm to manipulate a block to achieve the specification. We demonstrate that DiRL significantly outperforms state-of-the-art deep RL algorithms for learning policies from specifications, such as SPECTRL, TLTL, QRM and HRM, as well as a state-of-the-art hierarchical RL algorithm, R-AVI, that uses state abstractions, as the complexity of the specification increases. In particular, by exploiting the structure of the specification to decouple high-level planning and low-level control, the sample complexity of DiRL scales roughly linearly in the size of the specification, whereas the baselines quickly degrade in performance. Our results demonstrate that DiRL is capable of learning to perform complex tasks in challenging continuous control environments. In summary, our contributions are as follows:

- A novel compositional algorithm to learn policies for continuous domains from complex high-level specifications that interleave high-level model-based planning with low-level RL.
- A theoretical analysis of our algorithm showing that it aims to maximize a lower bound on the satisfaction probability of the specification.
- An empirical evaluation demonstrating that our algorithm outperforms several state-of-the-art algorithms for learning from high-level specifications.

**Motivating example.** Consider an RL-agent in the environment of interconnected rooms in Figure 1. The agent is initially in the blue box, and their goal is to navigate to either the top-left room $S_1$ or the bottom-right room $S_2$, followed by the top-right room $S_3$, all while avoiding the red block $O$. This goal is formally captured by the SPECTRL specification $\phi_{ex}$ (middle top). This specification is comprised of four simpler RL subtasks—namely, navigating between the corner rooms while avoiding the obstacle. Our approach, DiRL, leverages this structure to improve learning. First, based on the specification alone, it constructs the abstract graph $G_{ex}$ (see middle bottom) whose vertices represent the initial region and the three subgoal regions, and the edges correspond to subtasks (labeled with a safety constraint that must be satisfied).

However, $G_{ex}$ by itself is insufficient to determine the optimal path—e.g., it does not know that there is no path leading directly from $S_2$ to $S_3$, which is a property of the environment. These differences

---

Footnote: Our implementation is available at [https://github.com/keyshor/dirl](https://github.com/keyshor/dirl)
can be represented as \((a \text{ priori unknown})\) edge costs in \(G_{ex}\). At a high level, DtRL trains a policy \(\pi_e\) for each edge \(e\) in \(G_{ex}\), and sets the cost of \(e\) to be \(c(e; \pi_e) = -\log P(e; \pi_e)\), where \(P(e; \pi_e)\) is the probability that \(\pi_e\) succeeds in achieving \(e\). For instance, for the edge \(S_0 \rightarrow S_1\), \(\pi_e\) is trained to reach \(S_1\) from a random state in \(S_0\) while avoiding \(O\). Then, a naïve strategy for identifying the optimal path is to (i) train a policy \(\pi_e\) for each edge \(e\), (ii) use it to estimate the edge cost \(c(e; \pi_e)\), and (iii) run Dijkstra’s algorithm with these costs.

One challenge is that \(\pi_e\) depends on the initial states used in its training—e.g., training \(\pi_e\) for \(e = S_1 \rightarrow S_3\) requires a distribution over \(S_1\). Using the wrong distribution can lead to poor performance due to distribution shift; furthermore, training a policy for all edges may unnecessarily waste effort training policies for unimportant edges. To address these challenges, DtRL interweaves training policies with the execution of Dijkstra’s algorithm, only training \(\pi_e\) once Dijkstra’s algorithm requires the cost of edge \(e\). This strategy enables DtRL to scale to complex tasks; in our example, it quickly learns a policy that satisfies the specification with high probability. These design choices are validated empirically—as shown in Figure 1—DtRL quickly learns to achieve the specification, whereas it is beyond the reach of existing approaches.

**Related Work.** There has been recent work on using specifications based on temporal logic for specifying RL tasks [2, 7, 12, 18, 30, 17, 40, 15, 39, 23]. These approaches typically generate a (usually sparse) reward function from a given specification which is then used by an off-the-shelf RL algorithm to learn a policy. In particular, Li et al. [29] propose a variant of Linear Temporal Logic (LTL) called TLTL to specify tasks for robots, and then derive shaped (continuous) rewards from specifications in this language. Jothimurugaran et al. [24] propose a specification language called SPECTRL that allows users to encode complex tasks involving sequences, disjunctions, and conjunctions of subtasks, as well as specify safety properties; then, given a specification, they construct a finite state machine called a *task monitor* that is used to obtain shaped (continuous) rewards. Icarte et al. [19] propose an automaton based model called *reward machines* (RM) for high-level task specification and decomposition as well as an RL algorithm (QRM) that exploits this structure. In a later paper [35], they propose variants of QRM including an hierarchical RL algorithm (HRM) to learn policies for tasks specified using RMs. Camacho et al. [9] show that one can generate RMs from temporal specifications but RMs generated this way lead to sparse rewards. Kuo et al. [27] and Vaezipoor et al. [36] propose frameworks for multitask learning using LTL specifications but such approaches require a lot of samples even for relatively simpler environments and tasks. There has also been recent work on using temporal logic specifications for multi-agent RL [16, 33].

More broadly, there has been work on using *policy sketches* [5], which are sequences of subtasks designed to achieve the goal. They show that such approaches can speed up learning for long-horizon tasks. Sun et al. [34] show that providing semantics to the subtasks (e.g., encode rewards that describe when the subtask has been achieved) can further speed up learning. There has also been recent interest in combining high-level planning with reinforcement learning [11, 25, 13]. These approaches all target MDPs with reward functions, whereas we target MDPs with logical task specifications. Furthermore, in our setting, the high-level structure is derived from the given specification, whereas in existing approaches it is manually provided. Illanes et al. [20] propose an RL algorithm for reachability tasks that uses high-level planning to guide low-level RL; however, unlike our approach, they assume that a high-level model is given and high-level planning is not guided by the learned low-level policies. Finally, there has been recent work on applying formal reasoning for extracting interpretable policies [37, 38, 21] as well as for safe reinforcement learning [4, 26].

**2 Problem Formulation**

**MDP.** We consider a *Markov decision process* (MDP) \(M = (S, A, P, \eta)\) with continuous states \(S \subseteq \mathbb{R}^n\), continuous actions \(A \subseteq \mathbb{R}^m\), transitions \(P(s, a, s') = p(s' | s, a) \in \mathbb{R}_{>0}\) (i.e., the probability density of transitioning from state \(s\) to state \(s'\) upon taking action \(a\)), and initial states \(\eta : S \rightarrow \mathbb{R}_{>0}\) (i.e., \(\eta(s)\) is the probability density of the initial state being \(s\)). A *trajectory* \(\zeta \in \mathcal{Z}\) is either an infinite sequence \(\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots\) or a finite sequence \(\zeta = s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{k-1}} s_k\) where \(s_i \in S\) and \(a_i \in A\). A subtrajectory of \(\zeta\) is a subsequence \(\zeta_{t:k} = s_t \xrightarrow{a_t} \cdots \xrightarrow{a_{k-1}} s_k\). We let \(\mathcal{Z}_f\) denote the set of finite trajectories. A (deterministic) *policy* \(\pi : \mathcal{Z}_f \rightarrow A\) maps a finite trajectory to a fixed action. Given \(\pi\), we can sample a trajectory by sampling an initial state \(s_0 \sim \eta(\cdot)\), and then iteratively taking the action \(a_t = \pi(s_0, \cdot)\) and sampling a next state \(s_{t+1} \sim p(\cdot | s_t, a_t)\).
Abstract Reachability. We begin by defining the abstract reachability problem. A specification \( \phi \) in this language is a logical formula over trajectories that indicates whether a given trajectory \( \zeta \) successfully accomplishes the desired task. As described below, it can be interpreted as a function \( \phi : Z \rightarrow B \), where \( B = \{true, false\} \).

Formally, a specification is defined over a set of atomic predicates \( P_0 \), where every \( p \in P_0 \) is associated with a function \( [p] : S \rightarrow B \); we say a state \( s \) satisfies \( p \) (denoted \( s \models p \)) if and only if \( [p](s) = true \). For example, given a state \( s \in S \), the atomic predicate \( \text{reach } s \) (\( [\text{reach } s](s') = \|s' - s\| < 1 \)) indicates whether the system is in a state close to \( s \) with respect to the norm \( \| \cdot \| \). The set of predicates \( P \) consists of conjunctions and disjunctions of atomic predicates. The syntax of a predicate \( b \in P \) is given by the grammar \( b ::= p \mid (b_1 \land b_2) \mid (b_1 \lor b_2) \), where \( p \in P_0 \). Similar to atomic predicates, each predicate \( b \in P \) corresponds to a function \( [b] : S \rightarrow B \) defined naturally over Boolean logic. Finally, the syntax of \( \text{PECTRL} \) specifications is given by

\[
\phi ::= \text{achieve } b \mid \phi_1 \text{ ensuring } b \mid \phi_1; \phi_2 \mid \phi_1 \text{ or } \phi_2,
\]

where \( b \in P \). In this case, each specification \( \phi \) corresponds to a function \( [\phi] : Z \rightarrow B \), and we say \( \zeta \in Z \) satisfies \( \phi \) (denoted \( \zeta \models \phi \)) if and only if \( [\phi](\zeta) = true \). Letting \( \zeta \) be a finite trajectory of length \( t \), this function is defined by

\[
\begin{align*}
\zeta &\models \text{achieve } b \quad \text{ if } \exists i \leq t, s_i \models b \\
\zeta &\models \phi \text{ ensuring } b \quad \text{ if } \zeta \models \phi \text{ and } \forall i \leq t, s_i \models b \\
\zeta &\models \phi_1; \phi_2 \quad \text{ if } \exists i < t, \zeta_{0:i} \models \phi_1 \text{ and } \zeta_{i+1:t} \models \phi_2 \\
\zeta &\models \phi_1 \text{ or } \phi_2 \quad \text{ if } \zeta \models \phi_1 \text{ or } \zeta \models \phi_2.
\end{align*}
\]

Intuitively, the first clause means that the trajectory should eventually reach a state that satisfies the predicate \( b \). The second clause says that the trajectory should satisfy specification \( \phi \) while always staying in states that satisfy \( b \). The third clause says that the trajectory should sequentially satisfy \( \phi_1 \) followed by \( \phi_2 \). The fourth clause means that the trajectory should satisfy either \( \phi_1 \) or \( \phi_2 \). An infinite trajectory \( \zeta \) satisfies \( \phi \) if there is a \( t \) such that the prefix \( \zeta_{0:t} \) satisfies \( \phi \).

We assume that we are able to evaluate \( [p](s) \) for any atomic predicate \( p \) and any state \( s \). This is a common assumption in the literature on learning from specifications, and is necessary to interpret a given specification \( \phi \).

Learning from Specifications. Given an MDP \( M \) with unknown transitions and a specification \( \phi \), our goal is to compute a policy \( \pi^* : Z_f \rightarrow A \) such that \( \pi^* \in \arg \max_\pi \Pr_{\zeta \sim D_\pi}[\zeta \models \phi] \), where \( D_\pi \) is the distribution over infinite trajectories generated by \( \pi \). In other words, we want to learn a policy \( \pi^* \) that maximizes the probability that a generated trajectory \( \zeta \) satisfies the specification \( \phi \).

We consider the reinforcement learning setting in which we do not know the probabilities \( P \) but instead only have access to a simulator of \( M \). Typically, we can only sample trajectories of \( M \) starting at an initial state \( s_0 \sim \eta \). Some parts of our algorithm are based on an assumption that we can sample trajectories starting at any state that has been observed before. For example, if taking action \( a_0 \) in \( s_0 \) leads to a state \( s_1 \), we can store \( s_1 \) and obtain future samples starting at \( s_1 \).

Assumption 2.1. We can sample \( p(\cdot \mid s, a) \) for any previously observed state \( s \) and any action \( a \).

3 Abstract Reachability

In this section, we describe how to reduce the RL problem for a given MDP \( M \) and specification \( \phi \) to a reachability problem on a directed acyclic graph (DAG) \( \mathcal{G}_\phi \), augmented with information connecting its edges to subtrajectories in \( M \). In Section 4 we describe how to exploit the compositional structure of \( \mathcal{G}_\phi \) to learn efficiently.

3.1 Abstract Reachability

We begin by defining the abstract reachability problem, and describe how to reduce the problem of learning from a \( \text{PECTRL} \) specification to abstract reachability. At a high level, abstract reachability is defined as a graph reachability problem over a directed acyclic graph (DAG) whose vertices

\[\text{Here, achieve and ensuring correspond to the “eventually” and “always” operators in temporal logic.}\]
correspond to subgoal regions—a subgoal region $X \subseteq S$ is a subset of the state space $S$. As discussed below, in our reduction, these subgoal regions are derived from the given specification $\phi$. The constructed graph structure also encodes the relationships between subgoal regions.

**Definition 3.1.** An abstract graph $G = (U, E, u_0, F, \beta, Z_{\text{safe}})$ is a directed acyclic graph (DAG) with vertices $U$, (directed) edges $E \subseteq U \times U$, initial vertex $u_0 \in U$, final vertices $F \subseteq U$, subgoal region map $\beta : U \to 2^S$ such that for each $u \in U$, $\beta(u)$ is a subgoal region, and safe trajectories $Z_{\text{safe}} = \bigcup_{e \in E} Z_e^{\text{safe}}$, where $Z_e^{\text{safe}}$ denotes the safe trajectories for edge $e \in E$.

Intuitively, $(U, E)$ is a standard DAG, and $u_0$ and $F$ define a graph reachability problem for $(U, E)$. Furthermore, $\beta$ and $Z_{\text{safe}}$ connect $(U, E)$ back to the original MDP $M$; in particular, for an edge $e = u \to u'$, $Z_e^{\text{safe}}$ is the set of trajectories in $M$ that can be used to transition from $\beta(u)$ to $\beta(u')$.

**Definition 3.2.** An infinite trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ in $M$ satisfies abstract reachability for $G$ (denoted $\zeta \models G$) if there is a sequence of indices $0 = i_0 \leq i_1 < \cdots < i_k$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_k$ in $G$ such that

- $u_k \in F$,
- for all $j \in \{0, \ldots, k\}$, we have $s_{i_j} \in \beta(u_j)$, and
- for all $j < k$, letting $e_j = u_j \to u_{j+1}$, we have $\zeta_{i_j:i_{j+1}} \in Z_{e_j}^{\text{safe}}$.

The first two conditions state that the trajectory should visit a sequence of subgoal regions corresponding to a path from the initial vertex to some final vertex, and the last condition states that the trajectory should be composed of subtrajectories that are safe according to $Z_{\text{safe}}$.

**Definition 3.3.** Given MDP $M$ with unknown transitions and abstract graph $G$, the abstract reachability problem is to compute a policy $\hat{\pi} : Z_f \to A$ such that $\hat{\pi} \in \arg\max\pi \text{Pr}_{\zeta \sim D_\pi}[\zeta \models G]$.

In other words, the goal is to find a policy for which the probability that a generated trajectory satisfies abstract reachability is maximized.

### 3.2 Reduction to Abstract Reachability

Next, we describe how to reduce the RL problem for a given MDP $M$ and a specification $\phi$ to an abstract reachability problem for $M$ by constructing an abstract graph $G_\phi$ inductively from $\phi$. We give a high-level description here, and provide details in Appendix A in the supplement.

First, for each predicate $b$, we define the corresponding subgoal region $S_b = \{s \in S \mid s \models b\}$ denoting the set of states at which $b$ holds. Next, the abstract graph $G_\phi$ for $\phi = \text{achieve } b$ is shown in Figure 2. All trajectories in $Z_f$ are considered safe for the edge $e = u_0 \to u_1$ and the only final vertex is $u_1$ with $\beta(u_1) = S_b$. The abstract graph for a specification of the form $\phi = \phi_1 \land b$ ensuring $b$ is obtained by taking the graph $G_{\phi_1}$ and replacing the set of safe trajectories $Z_{\text{safe}}$, for each $e \in E$, with the set $Z_{\text{safe}} \cap Z_b$, where $Z_b = \{\zeta \in Z_f \mid \forall i . s_i \models b\}$ is the set of trajectories in which all states satisfy $b$. For the sequential specification $\phi = \phi_1 ; \phi_2$, we construct $G_\phi$ by adding edges from every final vertex of $G_{\phi_1}$ to every vertex of $G_{\phi_2}$ that is a neighbor of its initial vertex. Finally, choice $\phi = \phi_1$ or $\phi_2$ is handled by merging the initial vertices of the graphs corresponding to the two sub-specifications. Figure 1 shows an example abstract graph. The labels on the vertices are regions in the environment. All trajectories that avoid hitting the obstacle $O$ are safe for all edges. We have the following key guarantee:

**Theorem 3.4.** Given a SPECTRL specification $\phi$, we can construct an abstract graph $G_\phi$ such that, for every infinite trajectory $\zeta \in Z$, we have $\zeta \models \phi$ if and only if $\zeta \models G_\phi$. Furthermore, the number of vertices in $G_\phi$ is $O(|\phi|)$ where $|\phi|$ is the size of the specification $\phi$.

---

3We do not require that the subgoal regions partition the state space or that they be non-overlapping.
We give a proof in Appendix A. As a consequence, we can solve the reinforcement learning problem when the edge cost which it discovers using $N$ until it reaches $\beta$.

At a high level, our algorithm proceeds in three steps:

1. Initialize edge policies $\Pi \leftarrow \emptyset$.
2. While $\text{true}$ do:
   a. $u \leftarrow \text{NEARESTVERTEX}(U \setminus U_p, \Gamma, \Pi)$
   b. $\rho_u \leftarrow \text{SHORTESTPATH}(\Gamma_u)$
   c. $\eta_u \leftarrow \text{REACHEDISTRIBUTION}(\rho_u, \Pi)$
   d. If $u \in F$ then return $\text{PATHPOLICY}(\rho_u, \Pi)$.
   e. For $e = u \rightarrow u' \in \text{Outgoing}(u)$ do:
      i. $\pi_e \leftarrow \text{LEARNPOLICY}(e, \eta_u)$
      ii. Add $\rho_u \circ e$ to $\Gamma_{u'}$ and $\pi_e$ to $\Pi$
      iii. Add $u$ to $U_p$

Algorithm 1 Compositional reinforcement learning algorithm for solving abstract reachability.

**function** $\text{DiRL}(M, \mathcal{G})$

Initialize processed vertices $U_p \leftarrow \emptyset$
Initialize $\Gamma_{u_0} \leftarrow \{ u_0 \}$, and $\Gamma_u \leftarrow \emptyset$ for $u \neq u_0$
Initialize edge policies $\Pi \leftarrow \emptyset$

While true do:
   u ← NEARESTVERTEX($U \setminus U_p, \Gamma, \Pi$)
   $\rho_u \leftarrow \text{SHORTESTPATH}(\Gamma_u)$
   $\eta_u \leftarrow \text{REACHEDISTRIBUTION}(\rho_u, \Pi)$
   If $u \in F$ then return PATHPOLICY($\rho_u, \Pi$)
   For $e = u \rightarrow u' \in \text{Outgoing}(u)$ do:
      $\pi_e \leftarrow \text{LEARNPOLICY}(e, \eta_u)$
      Add $\rho_u \circ e$ to $\Gamma_{u'}$ and $\pi_e$ to $\Pi$
   Add $u$ to $U_p$

We give a proof in Appendix A. As a consequence, we can solve the reinforcement learning problem for $\phi$ by solving the abstract reachability problem for $\mathcal{G}_\phi$. As described below, we leverage the structure of $\mathcal{G}_\phi$ in conjunction with reinforcement learning to do so.

4 Compositional Reinforcement Learning

In this section, we propose a compositional approach for learning a policy to solve the abstract reachability problem for MDP $M$ (with unknown transition probabilities) and abstract graph $\mathcal{G}$.

4.1 Overview

At a high level, our algorithm proceeds in three steps:

- For each edge $e = u \rightarrow u'$ in $\mathcal{G}$, use RL to learn a neural network (NN) policy $\pi_e$ to try and transition the system from any state $s \in \beta(u)$ to some state $s' \in \beta(u')$ in a safe way according to $P^\text{safe}$. Importantly, this step requires a distribution $\eta_u$ over initial states $s \in \beta(u)$.
- Use sampling to estimate the probability $P(e; \pi_e, \eta_u)$ that $\pi_e$ safely transitions from $\beta(u)$ to $\beta(u')$.
- Use Dijkstra’s algorithm in conjunction with the edge costs $c(e) = -\log(P(e; \pi_e, \eta_u))$ to compute a path $\rho^* = u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_k$ in $\mathcal{G}$ that minimizes $c(\rho) = -\sum_{j=0}^{k-1} \log(P(e_j; \pi_j, \eta_j))$, where $e_j = u_j \rightarrow u_{j+1}$, $\pi_j = \pi_{e_j}$, and $\eta_j = \eta_{u_j}$.

Then, we could choose $\pi$ to be the sequence of policies $\pi_1, \ldots, \pi_{k-1}$—i.e., execute each policy $\pi_j$ until it reaches $\beta(u_{j+1})$, and then switch to $\pi_{j+1}$.

There are two challenges that need to be addressed in realizing this approach effectively. First, it is unclear what distribution to use as the initial state distribution $\eta_u$ to train $\pi_e$. Second, it might be unnecessary to learn all the policies since a subset of the edges might be sufficient for the reachability task. Our algorithm (Algorithm 1) addresses these issues by lazily training $\pi_e$—i.e., only training $\pi_e$ when the edge cost $c(e)$ is needed by Dijkstra’s algorithm.

In more detail, DiRL iteratively processes vertices in $\mathcal{G}$ starting from the initial vertex $u_0$, continuing until it processes a final vertex $u \in F$. It maintains the property that for every $u$ it processes, it has already trained policies for all edges along some path $\rho_u$ from $u_0$ to $u$. This property is satisfied by $u_0$ since there is a path of length zero from $u_0$ to itself. In Algorithm 1, $\Gamma_u$ is the set of all paths from $u_0$ to $u$ discovered so far, $\Gamma = \bigcup_u \Gamma_u$, and $\Pi = \{ \pi_e | e = u \rightarrow u' \in E, u \in U_p \}$ is the set of all edge policies trained so far. In each iteration, DiRL processes an unprocessed vertex $u$ nearest to $u_0$, which it discovers using NEARESTVERTEX, and performs the following steps:

1. **SHORTESTPATH** selects the shortest path from $u_0$ to $u$ in $\Gamma_u$, denoted $\rho_u = u_0 \rightarrow \cdots \rightarrow u_k = u$. 
2. **REACHEDISTS** computes the distribution \( \eta_u \) over states in \( \beta(u) \) induced by using the sequence of policies \( \pi_{e_0}, \ldots, \pi_{e_{k-1}} \in \Pi \), where \( e_j = u_j \rightarrow u_{j+1} \) are the edges in \( \rho_u \).

3. For every edge \( e = u \rightarrow u' \), **LEARNPOLICY** learns a policy \( \pi_e \) for \( e \) using \( \eta_u \) as the initial state distribution, and adds \( \pi_e \) to \( \Pi \) and \( \rho_e \) to \( \Gamma_u' \), where \( \rho_u' = u_0 \rightarrow \cdots \rightarrow u \rightarrow u' \); \( \pi_e \) is trained to ensure that the resulting trajectories from \( \beta(u) \) to \( \beta(u') \) are in \( Z_{\text{safe}} \) with high probability.

### 4.2 Definitions and Notation

**Edge costs.** We begin by defining the edge costs used in Dijkstra's algorithm. Given a policy \( \pi_e \) for edge \( e = u \rightarrow u' \), and an initial state distribution \( \eta_u \) over the subgoal region \( \beta(u) \), the cost \( c(e) \) of \( e \) is the negative log probability that \( \pi_e \) safely transitions the system from \( s_0 \sim \eta_u \) to \( \beta(u') \). First, we say a trajectory \( \xi \) starting at \( s_0 \) achieves an \( e \) if it safely reaches \( \beta(u') \)—formally:

**Definition 4.1.** An infinite trajectory \( \xi = s_0 \rightarrow s_1 \rightarrow \cdots \) achieves edge \( e = u \rightarrow u' \) in \( \mathcal{G} \) (denoted \( \xi \models e \)) if (i) \( s_0 \in \beta(u) \), and (ii) there exists \( i \) (constrained to be positive if \( u \neq u_0 \)) such that \( s_i \in \beta(u') \) and \( \zeta_{0:i} \in Z_{\text{safe}} \); we denote the smallest such \( i \) by \( i(\xi, e) \).

Then, the probability that \( \pi \) achieves \( e \) from an initial state \( s_0 \sim \eta_u \) is

\[
P(e; \pi_e, \eta_u) = \Pr_{s_0 \sim \eta_u, \xi \sim \mathcal{D}_{\pi_e, \pi}} [\xi \models e],
\]

where \( \mathcal{D}_{\pi_e, \pi} \) is the distribution over infinite trajectories induced by using \( \pi_e \) from initial state \( s_0 \).

Finally, the cost of edge \( e \) is \( c(e) = -\log P(e; \pi_e, \eta_u) \). Note that \( c(e) \) is nonnegative for any edge \( e \).

**Path policies.** Given edge policies \( \Pi \) along with a path \( \rho = u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_k = u \) in \( \mathcal{G} \), we define a path policy \( \pi_\rho \) to navigate from \( \beta(u_0) \) to \( \beta(u) \). In particular, \( \pi_\rho \) executes \( \pi_{u_j \rightarrow u_{j+1}} \) (starting from \( j = 0 \)) until reaching \( \beta(u_{j+1}) \), after which it increments \( j \leftarrow j + 1 \) (unless \( j = k \)). That is, \( \pi_\rho \) is designed to achieve the sequence of edges in \( \rho \). Note that \( \pi_\rho \) is stateful since it internally keeps track of the index \( j \) of the current policy.

**Induced distribution.** Let path \( \rho = u_0 \rightarrow \cdots \rightarrow u_k = u \) from \( u_0 \) to \( u \) be such that edge policies for all edges along the path have been trained. The induced distribution \( \eta_\rho \) is defined inductively on the length of \( \rho \). Formally, for the zero length path \( \rho = u_0 \) (so \( u = u_0 \)), we define \( \eta_\rho = \eta_u \) to be the initial state distribution of the MDP \( \mathcal{M} \). Otherwise, we have \( \rho = \rho' \circ e \), where \( e = u' \rightarrow u \). Then, we define \( \eta_\rho \) to be the state distribution over \( \beta(u) \) induced by using \( \pi_e \) from \( s_0 \sim \eta_{\rho'} \) conditioned on \( \xi \models e \). Formally, \( \eta_\rho \) is the probability distribution over \( \beta(u) \) such that for a set of states \( S' \subseteq \beta(u) \), the probability of \( S' \) according to \( \eta_\rho \) is

\[
\Pr_{s \sim \eta_\rho} [s \in S'] = \Pr_{s_0 \sim \eta_{\rho'}, \xi \sim \mathcal{D}_{\pi_e, \pi_\rho}} [s_{i(\xi, e)} \in S' \mid \xi \models e].
\]

**Path costs.** The cost of a path \( \rho = u_0 \rightarrow \cdots \rightarrow u_k = u \) is \( c(\rho) = -\sum_{j=0}^{k-1} \log P(e_j; \pi_e, \eta_{\rho_0:j}) \) where \( e_j = u_j \rightarrow u_{j+1} \) is the \( j \)-th edge in \( \rho \), and \( \rho_{0:j} = u_0 \rightarrow \cdots \rightarrow u_j \) is the \( j \)-th prefix of \( \rho \).

### 4.3 Algorithm Details

**DLIRL** interleaves Dijkstra's algorithm with using RL to train policies \( \pi_e \). Note that the edge weights to run Dijkstra's are not given \textit{a priori} since the edge policies and initial state/distributed distributions are unknown. Instead, they are computed on-the-fly beginning from the subgoal region \( u_0 \) using Algorithm 1. We describe each subroutine below.

**Processing order (NEARESTVERTEX).** On each iteration, **DLIRL** chooses the vertex \( u \) to process next to be an unprocessed vertex that has the shortest path from \( u_0 \)—i.e., \( u \in \arg \min_{u \in \mathcal{V} \setminus \mathcal{U}_u} \min_{e \in \mathcal{G}_u} c(\rho) \). This choice is an important part of Dijkstra's algorithm. For a graph with fixed costs, it ensures that the computed path \( \rho_u \) to each vertex \( u \) is minimized. While the costs in our setting are not fixed since they depend on \( \eta_u \), this strategy remains an effective heuristic.

**Shortest path computation (SHORTESTPATH).** This subroutine returns a path of minimum cost, \( \rho_u \in \arg \min_{\rho \in \Gamma_u} c(\rho) \). These costs can be estimated using Monte Carlo sampling.

**Initial state distribution (REACHEDISTRIBUTION).** A key choice **DLIRL** makes is what initial state distribution \( \eta_u \) to choose to train policies \( \pi_e \) for outgoing edges \( e = u \rightarrow u' \). **DLIRL** chooses the
initial state distribution $\eta_n = \eta_{\rho_n}$ to be the distribution of states reached by the path policy $\pi_{\rho_n}$ from a random initial state $s_0 \sim \eta_{\overline{U}}$

Learning an edge policy (LEARNPOLICY). Now that the initial state distribution $\eta_n$ is known, we describe how DiRL learns a policy $\pi_e$ for a single edge $e = u \rightarrow u'$. At a high level, it trains $\pi_e$ using a standard RL algorithm, where the rewards $\mathbb{1}(\xi \models e)$ are designed to encourage $\pi_e$ to safely transition the system to a state in $\beta(u')$. To be precise, DiRL uses RL to compute $\pi_e \in \arg\max_{\pi} P(e; \pi, \eta_n)$. Shaped rewards can be used to improve learning; see Appendix B.

Constructing a path policy (PATHPOLICY). Given edge policies $\Pi$ along with a path $\rho = u_0 \rightarrow \cdots \rightarrow u_n$, where $u_i \in F$ is a final vertex, DiRL returns the path policy $\pi_{\rho}$.

Theoretical Guarantee. We have the following guarantee (we give a proof in Appendix C).

Theorem 4.2. Given a path policy $\pi_{\rho}$ corresponding to a path $\rho = u_0 \rightarrow \cdots \rightarrow u_k = u$, where $u \in F$, we have $\Pr_{\xi \sim \pi_{\rho}}[\xi \models G] \geq \exp(-c(\rho))$.

In other words, we guarantee that minimizing the path cost $c(\rho)$ corresponds to maximizing a lower bound on the objective of the abstract reachability problem.

5 Experiments

We empirically evaluate our approach on several continuous control environments; details are in Appendix D, E and F.

Rooms environment. We consider the 9-Rooms environment shown in Figure 1 and a similar 16-Rooms environment. They have states $(x, y) \in \mathbb{R}^2$ encoding 2D position, actions $(v, \theta) \in \mathbb{R}^2$ encoding speed and direction, and transitions $s' = s + (v \cos(\theta), v \sin(\theta))$. For 9-Rooms, we consider specifications similar to $\phi_{\text{ex}}$ in Figure 1. For 16-Rooms, we consider a series of increasingly challenging specifications $\phi_1, \ldots, \phi_5$, each $\phi_i$ encodes a sequence of $i$ sub-specifications, each of which has the same form as $\phi_{\text{ex}}$ (see Appendix F). We learn policies using ARS [32] with shaped rewards (see Appendix B); one each is a fully connected NN with 2 hidden layers of 30 neurons each.

Fetch environment. We consider the Fetch-Pick-And-Place environment in OpenAI Gym [8], consisting of a robotic arm that can grasp objects and a block to manipulate. The state space is $\mathbb{R}^{25}$, which includes components encoding the gripper position, the (relative) position of the object, and the distance between the gripper fingers. The action space is $\mathbb{R}^4$, where the first 3 components encode the target gripper position and the last encodes the target gripper width. The block’s initial position is a random location on a table. We consider predicates $\text{NearObj}$ (indicates if the gripper is close to the block), $\text{HoldingObj}$ (indicates if the gripper is holding the block), $\text{LiftedObj}$ (indicates if the block is above the table), and $\text{ObjAt}(g)$ (indicates if the block is close to a goal $g \in \mathbb{R}^3$).

We consider three specifications. First, PickAndPlace is

$$\phi_1 = \text{NearObj}; \text{HoldingObj}; \text{LiftedObj}; \text{ObjAt}(g),$$

where $g$ is a random goal location. Second, PickAndPlaceStatic is similar to the previous one, except the goal location is fixed. Third, PickAndPlaceChoice involves choosing between two tasks, each of which is a sequence of two subtasks similar to PickAndPlaceStatic. We learn policies using TD3 [14] with shaped rewards; each one is a fully connected NN with 2 hidden layers of 256 neurons each.

Baselines. We compare our approach to four state-of-the-art algorithms for learning from specifications, SPECTRL [24], QRM [19], HRM [35], and a TLTL [29] based approach, as well as a state-of-the-art hierarchical RL algorithm, R-AVI [25], that leverages state abstractions. We used publicly available implementations of SPECTRL, QRM, HRM and R-AVI. For QRM and HRM, we manually encoded the tasks as reward machines with continuous rewards. The variants QRM+CR and HRM+CR use counterfactual reasoning to reuse samples during training. Our implementation of TLTL uses the quantitative semantics defined in Li et al. [29] with ARS to learn a single policy for each task. We used the subgoal regions and the abstract graph generated by our algorithm as inputs to R-AVI. Since R-AVI only supports disjoint subgoal regions and furthermore assumes the ability to sample from any subgoal region, we only ran R-AVI on supported benchmarks. The learning curves

---

4This choice is the distribution of states reaching $u$ by the path policy $\pi_{\rho}$, eventually returned by DiRL. Thus, it ensures that the training and test distributions for edge policies in $\pi_{\rho}$ are equal.
We summarize the scalability of DiRL in Figure 3 where we show the average number of steps needed to achieve a given success probability $z$ as a function of the number of edges in $G_\phi$ (denoted by $|G_\phi|$). As can be seen, the sample complexity of DiRL scales roughly linearly in the graph size. Intuitively, each subtask takes a constant number of steps to learn, so the total number of steps required is proportional to $|G_\phi|$. In the supplement, we show learning curves for 9-Rooms (Figure 6) for a variety of specifications, and learning curves for a variant of 16-Rooms with many blocked doors with the same specifications described above (Figure 7). These experiments demonstrate the robustness of our tool on different specifications and environments. For instance, in the 16-Rooms environment with blocked doors, fewer policies satisfy the specification, which makes learning more challenging but DiRL is still able to learn high-quality policies for all the specifications.

Next, we show results for the Fetch environment in Figure 4. The trends are similar to before—DiRL leverages compositionality to quickly learn effective policies, whereas the baselines are ineffective. The last task is especially challenging, taking DiRL somewhat longer to solve, but it ultimately achieves similar effectiveness. These results demonstrate that DiRL can scale to complex specifications even in challenging environments with high-dimensional state spaces.
Figure 4: Learning curves for Fetch environment; $x$-axis denotes the total number of samples (steps) and $y$-axis denotes the estimated probability of success. Results are averaged over 5 runs with error bars indicating ± standard deviation.

6 Conclusions

We have proposed DiRL, a reinforcement learning approach for logical specifications that leverages the compositional structure of the specification to decouple high-level planning and low-level control. Our experiments demonstrate that DiRL can effectively solve complex continuous control tasks, significantly improving over existing approaches. Logical specifications are a promising approach to enable users to more effectively specify robotics tasks; by enabling more scalable learning of these specifications, we are directly enabling users to specify more complex objectives through the underlying specification language. While we have focused on SPECTRL specifications, we believe our approach can also enable the incorporation of more sophisticated features into the underlying language, such as conditionals (i.e., only perform a subtask upon observing some property of the environment) and iterations (i.e., repeat a subtask until some objective is met).

Limitations

DiRL assumes the ability to sample trajectories starting at any state $s \in S$ that has been observed before, whereas in some cases it might only be possible to obtain trajectories starting at some initial state. One way to overcome this limitation is to use the learnt path policies for sampling—i.e., in order to sample a state from a subgoal region $\beta(u)$ corresponding to a vertex $u$ in the abstract graph, we could sample an initial state $s_0 \sim \eta$ from $\beta(u_0)$ and execute the path policy $\pi_{\rho_u}$ corresponding to the shortest path $\rho_u$ from $u_0$ to $u$ starting at $s_0$. Upon successfully reaching $\beta(u)$ (we can restart the sampling procedure if $\beta(u)$ is not reached), the system will be in a state $s \sim \eta_u$ in $\beta(u)$ from which we can simulate the system further.

Another limitation of our approach is that we only consider path policies. It is possible that an optimal policy must follow different high-level plans from different states within the same subgoal region. We believe this limitation can be addressed in future work by modifying our algorithm appropriately.

Societal impacts

Our work seeks to improve reinforcement learning for complex long-horizon tasks. Any progress in this direction would enable robotics applications both with positive impact—e.g., flexible and general-purpose manufacturing robotics, robots for achieving agricultural tasks, and robots that can be used to perform household chores—and with negative or controversial impact—e.g., military applications. These issues are inherent in all work seeking to improve the abilities of robots.

Acknowledgements and Funding

We thank the anonymous reviewers for their helpful comments. Funding in direct support of this work: CRA/NSF Computing Innovations Fellow Award, DARPA Assured Autonomy project under Contract No. FA8750-18-C-0090, ONR award N00014-20-1-2115, NSF grant CCF-1910769, and ARO grant W911NF-20-1-0080.
References

[1] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In International Conference on Artificial Intelligence and Statistics, pages 1639–1650. PMLR, 2020.

[2] Derya Aksaray, Austin Jones, Zhaodan Kong, Mac Schwager, and Calin Belter. Q-learning for robust satisfaction of signal temporal logic specifications. In Conference on Decision and Control (CDC), pages 6565–6570. IEEE, 2016.

[3] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. arXiv preprint arXiv:1606.06565, 2016.

[4] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In Advances in Neural Information Processing Systems, 2020.

[5] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In International Conference on Machine Learning, pages 166–175, 2017.

[6] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob Mccrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. The International Journal of Robotics Research, 39(1):3–20, 2020.

[7] Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. Ltlf/Ildf non-markovian rewards. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Alberto Camacho, Rodrigo Toro Icarte, Torey Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Ltfl and beyond: Formal languages for reward function specification in reinforcement learning. In International Joint Conference on Artificial Intelligence, pages 6065–6073, 2019.

[10] Yevgen Chebotar, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine. Combining model-based and model-free updates for trajectory-centric reinforcement learning. In International conference on machine learning, pages 703–711. PMLR, 2017.

[11] Steve Collins, Andy Ruina, Russ Tedrake, and Martijn Wisse. Efficient bipedal robots based on passive-dynamic walkers. Science, 307(5712):1082–1085, 2005.

[12] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Foundations for restraining bolts: Reinforcement learning with ltfl/ldf restraining specifications. In Proceedings of the International Conference on Automated Planning and Scheduling, volume 29, pages 128–136, 2019.

[13] Benjamin Eysenbach, Ruslan Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In NeurIPS, 2019.

[14] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In International Conference on Machine Learning, pages 1587–1596, 2018.

[15] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Omega-regular objectives in model-free reinforcement learning. In Tools and Algorithms for the Construction and Analysis of Systems, pages 395–412, 2019.

[16] Lewis Hammond, Alessandro Abate, Julian Gutierrez, and Michael Wooldridge. Multi-agent reinforcement learning with temporal logic specifications. In International Conference on Autonomous Agents and MultiAgent Systems, page 583–592, 2021.

[17] M. Hasanbeig, Y. Kantaros, A. Abate, D. Kroening, G. J. Pappas, and I. Lee. Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In Conference on Decision and Control (CDC), pages 5338–5343, 2019.
[18] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained reinforcement learning. *arXiv preprint arXiv:1801.08099*, 2018.

[19] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

[20] León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A. McIlraith. Symbolic plans as high-level instructions for reinforcement learning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):540–550, Jun. 2020.

[21] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[22] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. Neurosymbolic transformers for multi-agent communication. *arXiv preprint arXiv:2101.03238*, 2021.

[23] Yuqian Jiang, Sudarshanah Bharadwaj, Bo Wu, Rishi Shah, Ufuk Topcu, and Peter Stone. Temporal-logic-based reward shaping for continuing learning tasks, 2020.

[24] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. In *Advances in Neural Information Processing Systems*, volume 32, pages 13041–13051, 2019.

[25] Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. Abstract value iteration for hierarchical reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 1162–1170. PMLR, 2021.

[26] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. Safety-constrained reinforcement learning for mdps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 130–146. Springer, 2016.

[27] Yen-Ling Kuo, B. Katz, and A. Barbu. Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of ltl formulas. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5604–5610, 2020.

[28] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[29] Xiao Li, Cristian-Ioan Vasile, and Calin Belta. Reinforcement learning with temporal logic rewards. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3834–3839. IEEE, 2017.

[30] Michael L. Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via gltl, 2017.

[31] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275*, 2017.

[32] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1805–1814, 2018.

[33] Cyrus Neary, Zhe Xu, Bo Wu, and Ufuk Topcu. Reward machines for cooperative multi-agent reinforcement learning, 2021.

[34] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

[35] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *arXiv preprint arXiv:2010.03950*, 2020.
[36] Pashootan Vaezipoor, Andrew Li, Rodrigo Toro Icarte, and Sheila McIlraith. Ltl2action: Generalizing ltl instructions for multi-task rl. *arXiv preprint arXiv:2102.06858*, 2021.

[37] Abhinav Verma, Vijayraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054, 2018.

[38] Abhinav Verma, Hoang M Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

[39] Zhe Xu and Ufuk Topcu. Transfer of temporal logic formulas in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 4010–4018, 7 2019.

[40] Lim Zun Yuan, Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Modular deep reinforcement learning with temporal logic specifications. *arXiv preprint arXiv:1909.11591*, 2019.
A Reduction to Abstract Reachability

In this section, we detail the construction of the abstract graph $G_\phi$ from a SPECTRL specification $\phi$. Given two sets of finite trajectories $Z_1, Z_2 \subseteq Z_f$, let us denote by $Z_1 \circ Z_2$ the concatenation of the two sets—i.e.,

$$Z_1 \circ Z_2 = \left\{ \zeta \in Z_f \mid \exists i < t \ . \ \zeta_{0:i} \in Z_1 \land \zeta_{(i+1):t} \in Z_2 \right\}.$$ 

In addition to the abstract graph $G = (U, E, u_0, F, \beta, Z_{safe})$ we also construct a set of safe terminal trajectories $Z_{term} = \bigcup_{u \in F} Z_{term}^u$ where $Z_{term}^u \subseteq Z_f$ is the set of terminal trajectories for the final vertex $u \in F$. Now, we define what it means for a finite trajectory $\zeta$ to satisfy the pair $(G, Z_{term})$.

**Definition A.1.** A finite trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_t} s_t$ in $M$ satisfies the pair $(G, Z_{term})$ (denoted $\zeta \models (G, Z_{term})$) if there is a sequence of indices $0 = i_0 < i_1 < \cdots < i_k = t$ and a path $\rho = u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_k$ in $G$ such that

- $u_k \in F$,
- for all $j \in \{0, \ldots, k\}$, we have $s_{i_j} \in \beta(u_j)$,
- for all $j < k$, letting $e_j = u_j \rightarrow u_{j+1}$, we have $\zeta_{i_j:i_{j+1}} \in Z_{safe}^{e_j}$, and
- $\zeta_{i_0:i_k} \in Z_{term}^{u_k}$.

We now outline the inductive construction of the pair $(G_\phi, Z_{term, \phi})$ from a specification $\phi$ such that any finite trajectory $\zeta \in Z_f$ satisfies $\phi$ if and only if $\zeta$ satisfies $(G_\phi, Z_{term, \phi})$.

**Objectives ($\phi = achieve\ b$).** The abstract graph is $G_\phi = (U, E, u_0, F, \beta, Z_{safe})$ where

- $U = \{u_0, u_b\}$ with $\beta(u_0) = S$ and $\beta(u_b) = S_b = \{s \mid s \models b\}$,
- $E = \{u_0 \rightarrow u_b\}$,
- $F = \{u_b\}$ and,
- $Z_{safe}^{u_0,u_b} = Z_{term} = Z_f$.

**Constraints ($\phi = \phi_1\ ensuring\ b$).** Let the abstract graph for $\phi_1$ be $G_{\phi_1} = (U_1, E_1, u_0^1, F_1, \beta_1, Z_{safe,1})$ and the terminal trajectories be $Z_{term,1}$. Then, the abstract graph for $\phi$ is $G_\phi = (U, E, u_0, F, \beta, Z_{safe})$ where

- $U = U_1$, $u_0 = u_0^1$, $E = E_1$ and $F = F_1$,
- $\beta(u) = \beta_1(u) \cap S_b$ for all $u \in U \setminus \{u_0\}$ where $S_b = \{s \mid s \models b\}$, and $\beta(u_0) = S$.
- $Z_{safe}^e = Z_{safe,1}^e \cap Z_b$ for all $e \in E$ where $Z_b = \{\zeta \in Z_f \mid \forall i . s_i \models b\}$.
- $Z_{term}^u = Z_{term,1}^u \cap Z_b$ for all $u \in F$.

**Sequencing ($\phi = \phi_1 ; \phi_2$).** Let the abstract graph for $\phi_i$ be $G_{\phi_i} = (U_i, E_i, u_0^i, F_i, \beta_i, Z_{safe,i})$ and the terminal trajectories be $Z_{term,i}$ for $i \in \{1, 2\}$. The abstract graph $G_\phi = (U, E, u_0, F, \beta, Z_{safe})$ is constructed as follows.

- $U = U_1 \cup U_2 \setminus \{u_0^2\}$.
- $E = E_1 \cup E_2 \cup E_{1\rightarrow 2}$ where
  $$E' = \{u \rightarrow u' \in E_2 \mid u \neq u_0^2\} \quad \text{and} \quad E_{1\rightarrow 2} = \{u^1 \rightarrow u^2 \mid u^1 \in F_1 \land u_0^2 \rightarrow u^2 \in E_2\}.$$
- $u_0 = u_0^1$ and $F = F_2$.
- $\beta(u) = \beta_i(u)$ for all $u \in U_i$ and $i \in \{1, 2\}$.
- The safe trajectories are given by
Lemma A.2. For any the Boolean semantics, we assume that the quantitative semantics is associated with a function \( \|p\|_q : S \rightarrow \mathbb{R} \). To ensure compatibility with the Boolean semantics, we assume that
\[
\|p\|_q(s) = (\|p\|_q(s) > 0).
\]
The safety reward is defined by

\[ [\text{reach } s]_q(s') = 1 - \|s' - s\| \]

indicates whether the system is in a state near \( s \) w.r.t. some norm \( \| \cdot \| \). In addition, we can extend the quantitative semantics to predicates \( b \in \mathcal{P} \) by recursively defining \([b_1 \land b_2]_q(s) = \min\{[b_1]_q(s),[b_2]_q(s)\}\) and \([b_1 \lor b_2]_q(s) = \max\{[b_1]_q(s),[b_2]_q(s)\}\). These definitions are a standard extension of Boolean logic to real values. In particular, they preserve \([1] \)—i.e., \( b \models s \) if and only if \([b]_q(s) > 0\).

In addition to quantitative semantics, we make use of the following property to define shaped rewards.

**Lemma B.1.** The abstract graph \( G_\phi = (U, E, u_0, F, \beta, Z_{\text{safe}}) \) of a specification \( \phi \) satisfies the following:

- For every non-initial vertex \( u \in U \setminus \{u_0\} \), there is a predicate \( b \in \mathcal{P} \) such that \( \beta(u) = S_b = \{s \mid s \models b\} \).
- For every \( e \in E \), either \( Z_{\text{safe}}^e = Z_b = \{\zeta \in Z \mid \forall i . s_i \models b\} \) for some \( b \in \mathcal{P} \) or \( Z_{\text{safe}}^e = Z_{b_1} \circ Z_{b_2} \) for some \( b_1, b_2 \in \mathcal{P} \).

**Proof sketch.** We prove a stronger property that, in addition to the above, requires that for any \( e = u_0 \rightarrow u \in E \), \( Z_{\text{safe}}^e = Z_b \) for some \( b \in \mathcal{P} \) and for any final vertex \( u \), \( Z_{\text{term}, \phi}^u = Z_b \) for some \( b \in \mathcal{P} \). This stronger property follows from a straightforward induction on \( \phi \).

Next, we describe the shaped rewards we use to learn an edge \( e = u \rightarrow u' \) in \( G_\phi \), which have the form

\[ R_{\text{step}}(s, a, s') = R_{\text{reach}}(s, a, s') + R_{\text{safe}}(s, a, s'). \]

Intuitively, the first term encodes a reward for reaching \( \beta(u') \), and the second term encodes a reward for maintaining safety. By Lemma B.1, \( \beta(u') = S_b \) for some \( b \in \mathcal{P} \). Then, we define

\[ R_{\text{reach}}(s, a, s') = [b]_q(s'). \]

The safety reward is defined by

\[ R_{\text{safe}}(s, a, s') = \begin{cases} \min\{0, [b]_q(s')\} & \text{if } Z_{\text{safe}}^e = Z_b \\ \min\{0, [b \lor b']_q(s')\} & \text{if } Z_{\text{safe}}^e = Z_b \circ Z_{b'} \land \psi_b \\ \min\{0, [b']_q(s')\} & \text{if } Z_{\text{safe}}^e = Z_b \circ Z_{b'} \land \neg \psi_b. \end{cases} \]

Here, \( \psi_b \) is internal state keeping track of whether \( b \) has held so far—i.e., \( \psi_b \leftarrow \psi_b \land [b]_q(s) \) at state \( s \). Intuitively, the first case is the simpler case, which checks if every state in the trajectory satisfies \( b \), and the latter two cases handle a sequence where \( b \) should hold for the first part of the trajectory, and \( b' \) should hold for the remainder.

**C Proof of Theorem 4.2**

**Proof.** Let the abstract graph be \( G = (U, E, u_0, F, \beta, Z_{\text{safe}}) \). Let us first define what it means for a rollout to achieve a path in \( G \).

**Definition C.1.** We say that an infinite trajectory \( \zeta \) achieves the path \( \rho \) (denoted \( \zeta \models \rho \)) if \( \zeta \models G_\rho \) where \( G_\rho = (U_\rho, E_\rho, u_0, \{u_k\}, \beta \downarrow \rho, Z_{\text{safe}} \downarrow \rho) \) with \( U_\rho = \{u_j \mid 0 \leq j \leq k\} \), \( E_\rho = \{u_j \rightarrow u_{j+1} \mid 0 \leq j \leq k\} \) and \( \beta \downarrow \rho \) and \( Z_{\text{safe}} \downarrow \rho \) are \( \beta \) and \( Z_{\text{safe}} \) restricted to the vertices and the edges of \( G_\rho \), respectively.

From the definition it is clear that for any infinite trajectory \( \zeta \), if \( \zeta \models \rho \) then \( \zeta \models G \) and therefore

\[ \Pr_{\zeta \sim D_{\ast, \rho}} [\zeta \models G] \geq \sum_{\rho \sim D_{\ast, \rho}} \Pr_{\zeta \sim D_{\ast, \rho}} [\zeta \models \rho]. \]

Let us now define a slightly stronger notion of achieving an edge.
Definition C.2. An infinite trajectory \( \zeta = s_0 \rightarrow s_1 \rightarrow \cdots \) is said to greedily achieve the path \( \rho \) (denoted \( \zeta \models g \rho \)) if there is a sequence of indices \( 0 = i_0 \leq i_1 < \cdots < i_k \) such that for all \( j < k \),

\[ i_j : \zeta_{i_j : \infty} \models e_j = u_j \rightarrow u_{j+1} \text{ and,} \]

\[ i_{j+1} = i(\zeta_{i_j : \infty}, e_j), \]

where \( \zeta_{i_j : \infty} = s_{i_j} \rightarrow s_{i_j+1} \rightarrow \cdots \).

That is, \( \zeta \models g \rho \) if a partition of \( \zeta \) realizing \( \rho \) can be be constructed greedily by picking \( i_{j+1} \) to be the smallest index \( i \geq i_j \) (strictly bigger if \( j > 0 \)) such that \( s_i \in \beta(u_{j+1}) \) and \( \zeta_{i,j;i} \in Z_{\text{safe}} \). Since \( \zeta \models g \rho \) implies \( \zeta \models \rho \), we have

\[
\Pr_{\zeta \sim D_{\pi_e}} [\zeta \models g \rho] \geq \Pr_{\zeta \sim D_{\pi_e'}} [\zeta \models g \rho]. \tag{3}
\]

Let \( \rho_{j:k} \) denote the \( j \)-th suffix of \( \rho \). We can decompose the probability \( \Pr_{\zeta \sim D_{\pi_e}} [\zeta \models g \rho] \) as follows.

\[
\Pr_{\zeta \sim D_{\pi_e}} [\zeta \models g \rho] = \Pr_{\zeta \sim D_{\pi_e}} [\zeta \models e_0 \land \zeta_{i((\zeta,e_0) : \infty} \models g \rho_{1:k}]
\]

\[ = \Pr_{\zeta \sim D_{\pi_e}} [\zeta \models e_0] \cdot \Pr_{\zeta_{i((\zeta,e_0) : \infty} \models g \rho_{1:k} | \zeta \models e_0]
\]

\[ = P(e_0; \pi_{e_0}, \eta_0) \cdot \Pr_{\zeta_{i=0} \sim \eta_{p_{0:i}} : \zeta_{i=1:k} \sim D_{\pi_{e_{1:k}}}} [\zeta \models g \rho_{1:k}]
\]

where the last equality followed from the definition of \( \eta_{p_{0:i}} \) and the Markov property of \( M \). Applying the above decomposition recursively, we get

\[
\Pr_{\zeta \sim D_{\pi_e}} [\zeta \models g \rho] = \prod_{j=0}^{k-1} P(e_j; \pi_{e_j}, \eta_{p_{0:j}})
\]

\[ = \exp(\log(\prod_{j=0}^{k-1} P(e_j; \pi_{e_j}, \eta_{p_{0:j}})))
\]

\[ = \exp\left(-(-\sum_{j=0}^{k-1} \log P(e_j; \pi_{e_j}, \eta_{p_{0:j}}))\right)
\]

\[ = \exp\left(-c(\rho)\right).
\]

Therefore, from Equations (2) and (3) we get the required bound. \( \square \)

D Experimental Methodology

Our tool learns the low-level NN policies for edges using an off-the-shelf RL algorithm. For the Rooms and Fetch environments, we learn policies using ARS \([32]\) and TD3 \([14]\) with shaped rewards, respectively.

For each specification on an environment, we first construct its abstract graph. In DiRL, each edge policy \( \pi_e \) is trained using \( k \) episodes of interactions with the environment. For the purpose of generating a learning curve, we run DiRL for each specification with several values of \( k \). For each \( k \) value, we plot the sum total of the samples taken to train all edge policies against the probability with which the computed policy reaches a final subgoal region.

For a fair comparison with the baselines, if each episode for learning an edge policy in DiRL is run for \( m \) steps, we run the episodes of the baselines for \( m \cdot d + c \) steps, where \( d \) is the maximum path length to reach a final vertex in the abstract graph of the specification and \( c > 0 \) is a buffer. Intuitively, this approach ensures that all tools get a similar number of steps in each episode to learn the specification.
We consider environments with several interconnected rooms. The rooms are separated by thick walls and are connected through bi-directional doors.

The environments are a 9-Rooms environment, a 16-Rooms environment with all doors open, and 16-Rooms environment with some doors open. The red blocks indicate obstacles. A robot can pass through those rooms by moving around the red blocks. The robot is initially placed randomly in the center of the room with the blue box (bottom-left corner).

Rooms are identified by the tuple \((r, c)\) denoting the room in the \(r\)-th row and \(c\)-th column. We use the convention that the bottom-left corner is room \((0,0)\). Predicate \(\text{reach} (r, c)\) is interpreted as reaching the center of the \((r, c)\)-th room and predicate \(\text{avoid} (r, c)\) is interpreted as avoiding the center of the \((r, c)\)-th room. For clarity, we omit the word \(\text{achieve}\) from specifications of the form \(\text{achieve} b\) denoting such a specification using just the predicate \(b\).

### E.1 9-Rooms Environment

Specifications.

1. \(\phi_1 := \text{reach} (2, 0) \land \text{reach} (0, 0)\)
   - Go to the top-left corner and then return to the bottom-left corner (initial room); red blocks not considered obstacles.
   - This specification is difficult for standard RL algorithms that do not store whether the first sub-task has been achieved. In these cases, a stateless policy will not be able to determine whether to move upwards or downwards. In contrast, DiRL (as well as SPECTRL and RM based approaches) augment the state space to automatically keep track of which sub-tasks have been achieved so far.

2. \(\phi_2 := \text{reach} (2, 0) \lor \text{reach} (0, 2)\)
   - Either go to the top-left corner or to the bottom-right corner (obstacles are not considered).

3. \(\phi_3 := \phi_2 \land \text{reach} (2, 2)\)
   - After completing \(\phi_2\), go to the top-right corner (obstacles not considered).
   - This specification combines two choices of similar difficulty yet only one is favorable to fulfilling the specification since the direct path to the top-right corner from the bottom-right one is obstructed by walls.

4. \(\phi_4 := \text{reach} (2, 0) \land \text{avoid} (1, 0)\)
   - Reach the top-left (while considering the obstacles).

5. \(\phi_5 := \phi_4 \lor \text{reach} (0, 2) \land \text{reach} (2, 2)\)
   - Either go to the top-left corner or bottom-right corner enroute to the top-right corner (while considering the obstacles).
   - This specification is similar to \(\phi_3\) except that the choices are of unequal difficulty due to the placement of the red obstacle. In this case, the non-greedy choice is favorable for completing the task.
Figure 6: Learning curves for 9-Rooms environment with different specifications. x-axis denotes the number of samples (steps) and y-axis denotes the estimated probability of success. Results are averaged over 10 runs with error bars indicating ± standard deviation.

**Hyperparameters.** The edge policies are learned using ARS [32] (version V2-t) with neural network policies and the following hyperparameters.

- Step-size $\alpha = 0.3$.
- Standard deviation of exploration noise $\nu = 0.05$.
- Number of directions sampled per iteration is 30.
- Number of top performing directions to use $b = 15$.

To plot the learning curve, we use values of $k \in \{3000, 6000, 12000, 18000, 24000, 30000\}$ where each episode consists of $m = 20$ steps.

**Results.** The learning curves for these specifications are shown in Figure 6. While most tools perform reasonably well on specifications $\phi_2$ (Figure 6b) and $\phi_4$ (Figure 6d), the baselines are unable to learn to satisfy $\phi_3$ (Figure 6c) and $\phi_5$ (Figure 6e) except for R-AVI which learns to satisfy $\phi_3$ as well.

### E.2 16-Rooms Environment

**Specifications.** We describe the five specifications used for the 16-rooms environment, which are designed to increase in difficulty. First, we define a segment as the following specification: Given the current location of the agent, the goal is to reach a room diagonally opposite to it by visiting at least one of the rooms at the remaining two corners of the rectangle formed by the current room and the goal room—e.g., in the 9-Rooms environment, to visit $S_3$ from the initial room, the agent must visit either $S_1$ or $S_2$ first.

Then, we design specifications of varying sizes by sequencing several segments one after the other. In the first segment, the agent’s current location is the initial room. In subsequent segments, the current location is the goal room of the previous segment. In addition, the agent must always avoid the obstacles in the environment. We create five such specifications, one half-segment and specifications up to four segments ($\phi_1$ to $\phi_5$), as illustrated in Figure 5a and described below:

1. $\phi_1$ corresponds to the half-segment enroute (2,2) from (0,0). Thus $\phi_1$ is a choice between (0,2) and (2,0).
2. $\phi_2$ is the first segment that goes from (0,0) to (2,2).
We denote by $|G_2|$ the number of edges in the abstract graph corresponding to the specification $\phi$.

**Hyperparameters.** We use the same hyperparameters of ARS as the ones used for the 9-Rooms environment. We run experiments for

$$k \in \{6000, 12000, 24000, 48000, 60000, 72000\}.$$

**Results.** The learning curves for the environment with all open doors and the constrained environment with some open doors are shown in Figure 3 and Figure 7 respectively.

## F Case Study: Fetch Environment

The fetch robotic arm from OpenAI Gym is visualized in Figure 8. Let us denote by $s_r = (s_r^x, s_r^y, s_r^z) \in \mathbb{R}^3$ the position of the gripper, $s_o \in \mathbb{R}^3$ the relative position of the object (black block) w.r.t. the gripper, $s_g \in \mathbb{R}^3$ the goal location (red sphere) and $s_w \in \mathbb{R}$ the width of the gripper. Let $c$ denote the width of the object and $z_\epsilon = (0, 0, \epsilon + c)$ for $\epsilon > 0$. Then, we define the following predicates.

- **NearObj** holds true in states in which the gripper is wide open, aligned with the object and is slightly above the object:
  
  $$\text{NearObj}(s) = (\|s_o + z_\epsilon\|^2 + (s_w - 2c)^2 < \delta_1)$$

- **HoldingObj** holds true in states in which the gripper is close to the object and its width is close to the object’s width:
  
  $$\text{HoldingObj}(s) = (\|s_o\|^2 + (s_w - c)^2 < \delta_2)$$

3. $\phi_3$ augments $\phi_2$ with a second segment to $(3,1)$.
4. $\phi_4$ augments $\phi_3$ with a segment to $(1,3)$
5. $\phi_5$ augments $\phi_4$ with a segment to $(0,1)$
Figure 8: Fetch robotic arm.

- LiftedObj holds true in states in which the object is above the surface level of the table:
  \[ \text{LiftedObj}(s) = (s_r^z + s_o^z > \delta_3) \]

- ObjAt\[g\] holds true in states in which the object is close to \(g\):
  \[ \text{ObjAt}[g](s) = (\|s_r + s_o - g\|_2^2 < \delta_4) \]

Then the specifications we use are the following:\footnote{We denote achieve \(b\) using just the predicate \(b\).}

- PickAndPlace: \(\phi_1 = \text{NearObj}; \text{HoldingObj}; \text{LiftedObj}; \text{ObjAt}[s_g]\).
- PickAndPlaceStatic: NearObj; HoldingObj; LiftedObj; ObjAt\[{g_1}\] where \(g_1\) is a fixed goal.
- PickAndPlaceChoice: (\text{NearObj}; \text{HoldingObj}; \text{LiftedObj}); ((\text{ObjAt}[g_1]; \text{ObjAt}[g_2]) \lor (\text{ObjAt}[g_3]; \text{ObjAt}[g_4]))

**Hyperparameters.** We use TD3 \cite{fujimoto2018 td3} for learning edge policies with the following hyperparameters.

- Discount \(\gamma = 0.95\).
- Adam optimizer; actor learning rate 0.0001; critic learning rate 0.001.
- Soft update targets \(\tau = 0.005\).
- Replay buffer of size 200000.
- 100 training steps performed every 100 environment steps.
- A minibatch of 256 steps used per training step.
- Exploration using gaussian noise with \(\sigma = 0.15\).

We run experiments for \(k \in \{1000, 2000, 4000\}\) and each episode consists of \(m = 40\) steps.