Induction of Subgoal Automata for Reinforcement Learning

Daniel Furelos-Blanco,[1] **Mark Law,**[1] **Alessandra Russo,**[1] **Krysia Broda,**[1] **Anders Jonsson**[2]
[1]Imperial College London, United Kingdom, [2]Universitat Pompeu Fabra, Barcelona, Spain
{d.furelos-blanco18, mark.law09, a.russo, k.broda}@imperial.ac.uk, anders.jonsson@upf.edu

Abstract

In this work we present ISA, a novel approach for learning and exploiting subgoals in reinforcement learning (RL). Our method relies on inducing an automaton whose transitions are subgoals expressed as propositional formulas over a set of observable events. A state-of-the-art inductive logic programming system is used to learn the automaton from observation traces perceived by the RL agent. The reinforcement learning and automaton learning processes are interleaved: a new refined automaton is learned whenever the RL agent generates a trace not recognized by the current automaton. We evaluate ISA in several gridworld problems and show that it performs similarly to a method for which automata are given in advance. We also show that the learned automata can be exploited to speed up convergence through reward shaping and transfer learning across multiple tasks. Finally, we analyze the running time and the number of traces that ISA needs to learn an automata, and the impact that the number of observable events have on the learner's performance.

1 Introduction

Reinforcement learning (RL) (Sutton and Barto 1998) is a family of algorithms where an agent acts in an environment with the purpose of maximizing the total amount of reward it receives. These algorithms have played a key role in major breakthroughs like human-level video game playing (Mnih et al. 2015) or mastering complex board games (Silver et al. 2018). However, current methods lack the ability to generalize and transfer between tasks. For example, they cannot learn to play chess using the knowledge of shōgi.

Advances in achieving generalization and transfer in RL are mainly due to abstractions (Ho et al. 2019; Konidaris 2019). Hierarchical reinforcement learning methods, which divide a single task into several subtasks that can be solved separately, are among the most promising approaches to abstracting RL tasks. Common frameworks for hierarchical RL are HAMs (Parr and Russell 1997), options (Sutton, Precup, and Singh 1999) and MAXQ (Dietterich 2000).

Abstract hierarchies can be naturally represented using automata, which have been used effectively in task abstraction, not only in RL (Parr and Russell 1997; Toro Icarte et al. 2018), but also in related areas like automated planning (Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2011; Segovia-Aguas, Jiménez, and Jonsson 2018). However, these RL approaches use handcrafted automata instead of learning them from data. It is largely an open problem in RL how to autonomously decompose a task into an automaton that can be exploited during the RL process.

In this paper, we address this problem and propose ISA (**I**nduction of **S**ubgoal **A**utomata for Reinforcement Learning), a method for learning and exploiting a minimal automaton from observation traces perceived by an RL agent. These automata are called *subgoal automata* since each transition is labeled with a subgoal, which is a boolean formula over a set of observables. Subgoal automata can be expressed in a logic programming format and, thus, be learned using state-of-the-art inductive logic programming (ILP) systems (Law, Russo, and Broda 2015). The resulting subgoal automaton can be exploited by an RL algorithm that learns a different policy for each automaton state, each aiming to achieve a subgoal. This decomposition allows learning multiple subpolicies simultaneously and transferring knowledge across multiple tasks. ISA interleaves the RL and automaton learning processes such that the agent can immediately leverage the new (partially correct) learned subgoal automaton: when a trace is not correctly recognized by the automaton, a new one is learned.

We evaluate ISA in several gridworld problems and show that it learns subgoal automata and policies for each of these. Specifically, it performs similarly to a method for which automata are given in advance. Furthermore, the learned automata can be exploited to speed up convergence through reward shaping and transfer learning.

The paper is organized as follows. Section 2 introduces the background of our work. Section 3 formally presents the problem we address and our method for solving it, while Section 4 describes the results. We discuss related work in Section 5 and conclude in Section 6.

2 Background

In this section we briefly summarize the key background concepts used throughout the paper: reinforcement learning and inductive learning of answer set programs.

Reinforcement Learning

Reinforcement learning (RL) (Sutton and Barto 1998) is a family of algorithms for learning to act in an unknown environment. Typically, this learning process is formulated as a *Markov Decision Process (MDP)*, i.e., a tuple $\mathcal{M} = \langle S, A, p, r, \gamma \rangle$, where S is a finite set of states, A is a finite set of actions, $p : S \times A \to \Delta(S)$ is a transition probability function[1], $r : S \times A \times S \to \mathbb{R}$ is a reward function, and $\gamma \in [0, 1)$ is a discount factor. At time t, the agent observes state $s_t \in S$, executes action $a_t \in A$, transitions to the next state $s_{t+1} \sim p(\cdot|s_t, a_t)$ and receives reward $r(s_t, a_t, s_{t+1})$.

We consider *episodic* MDPs that *terminate* in a given set of terminal states. We distinguish between goal states and undesirable terminal states (i.e., dead-ends). Let $S_T \subseteq S$ be the set of terminal states and $S_G \subseteq S_T$ the set of goal states. The aim is to find a *policy* $\pi : S \to \Delta(A)$, a mapping from states to probability distributions over actions, that maximizes the expected sum of discounted reward (or *return*), $R_t = \mathbb{E}[\sum_{k=t}^{n} \gamma^{k-t} r_k]$, where n is the last episode step.

In model-free RL the transition probability function p and reward function r are unknown to the agent, and a policy is learned via interaction with the environment. Q-learning (Watkins 1989) computes an *action-value function* $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a]$ that estimates the return from each state-action pair when following an approximately optimal policy. In each iteration the estimates are updated as

$$Q(s, a) \xleftarrow{\alpha} r(s, a, s') + \gamma \max_{a'} Q(s', a'),$$

where $x \xleftarrow{\alpha} y$ is shorthand for $x \leftarrow x + \alpha (y - x)$, α is a learning rate and s' is the state after applying a in s. Usually, an ϵ-greedy policy selects a random action with probability ϵ and the action maximizing $Q(s, a)$ otherwise. The policy is induced by the action that maximizes $Q(s, a)$ in each s.

Options (Sutton, Precup, and Singh 1999) address temporal abstraction in RL. Given an MDP $\mathcal{M} = \langle S, A, p, r, \gamma \rangle$, an *option* is a tuple $\omega = \langle I_\omega, \pi_\omega, \beta_\omega \rangle$ where $I_\omega \subseteq S$ is the option's initiation set, $\pi_\omega : S \to \Delta(A)$ is the option's policy, and $\beta_\omega : S \to [0, 1]$ is the option's termination condition. An option is available in state $s \in S$ if $s \in I_\omega$. If the option is started, the actions are chosen according to π_ω. The option terminates at a given state $s \in S$ with probability $\beta_\omega(s)$. The action set of an MDP can be augmented with options, which can be either handcrafted or automatically discovered. An MDP extended with options is a Semi-Markov Decision Process (SMDP); the learning methods for MDPs still apply to SMDPs. To improve sample-efficiency, the experiences (s, a, r, s') generated by an option's policy can be used to update other options' policies. This transfer learning method is called *intra-option learning* (Sutton, Precup, and Singh 1998).

Inductive Learning of Answer Set Programs

In this section we describe answer set programming (ASP) and the ILASP system for learning ASP programs.

[1]For any finite set X, $\Delta(X) = \{\mu \in \mathbb{R}^X : \sum_x \mu(x) = 1, \mu(x) \geq 0\ (\forall x)\}$ is the probability simplex over X.

Answer Set Programming (ASP) (Gelfond and Kahl 2014) is a declarative programming language for knowledge representation and reasoning. An ASP problem is expressed in a logical format and the models (called answer sets) of its representation provide the solutions to that problem.

A *literal* is an atom `a` or its negation `not a`. Given an atom `h` and a set of literals $\texttt{b}_1, \ldots, \texttt{b}_n$, a *normal rule* is of the form $\texttt{h :- b}_1, \ldots, \texttt{b}_n$, where `h` is the *head* and $\texttt{b}_1, \ldots, \texttt{b}_n$ is the *body* of the rule. Rules of the form $\texttt{:- b}_1, \ldots, \texttt{b}_n$ are called *constraints*. In this paper, we assume an ASP program P to be a set of normal rules and constraints. Given a set of ground atoms (or *interpretation*) I, a ground normal rule is satisfied if the head is satisfied by I when the body literals are satisfied by I. A ground constraint is satisfied if the body is not satisfied by I. The *reduct* P^I of a program P with respect to I is built by removing all rules including `not a` such that $\texttt{a} \in I$ from P. I is an *answer set* of P iff (1) I satisfies the rules of P^I, and (2) no subset of I satisfies the rules of P^I.

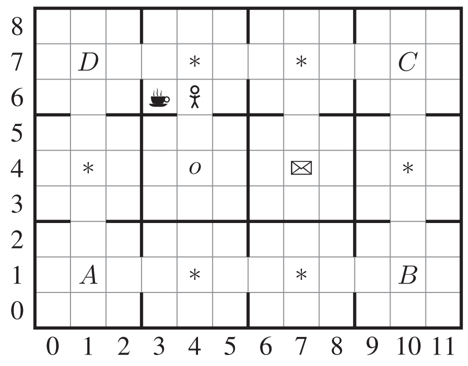
ILASP (Law, Russo, and Broda 2015) is a system for learning ASP programs from partial answer sets. A *context-dependent partial interpretation (CDPI)* (Law, Russo, and Broda 2016) is a pair $\langle \langle e^{inc}, e^{exc} \rangle, e^{ctx} \rangle$, where $\langle e^{inc}, e^{exc} \rangle$ is a partial interpretation and e^{ctx} is an ASP program, called *context*. A program P accepts e iff there is an answer set A of $P \cup e^{ctx}$ such that $e^{inc} \subseteq A$ and $e^{exc} \cap A = \emptyset$. An *ILASP task* (Law, Russo, and Broda 2016) is a tuple $T = \langle B, S_M, E \rangle$ where B is the ASP background knowledge, S_M is the set of rules allowed in the hypotheses, and E is a set of CDPIs, called examples. A hypothesis $H \subseteq S_M$ is a *solution* of T iff $\forall e \in E$, $B \cup H$ accepts e.

3 Methodology

In this section we describe ISA, our method that interleaves the learning of subgoal automata with the learning of policies for achieving these subgoals. The tasks we consider are *episodic MDPs* $\mathcal{M} = \langle S, A, p, r, \gamma, S_T, S_G \rangle$ where $r(s, a, s') = 1$ if $s' \in S_G$ and 0 otherwise.

The automaton transitions are defined by a logical formula over a set of *observables* $\mathcal{O}$. A *labeling function* $L : S \to 2^{\mathcal{O}}$ maps an MDP state into a subset of observables $O \subseteq \mathcal{O}$ (or *observations*) perceived by the agent at that state.

We use the OFFICEWORLD environment (Toro Icarte et al. 2018) to explain our method. It consists of a grid (see Figure 1a) where an agent (🚶) can move in the four cardinal directions, and the set of observables is $\mathcal{O} = \{$☕, ✉$, o, A, B, C, D, *\}$. The agent picks up coffee and the mail at locations ☕ and ✉ respectively, and delivers them to the office at location o. The decorations $*$ are broken if the agent steps on them. There are also four locations labeled A, B, C and D. The observables A, B, C and D, and decorations $*$ do not share locations with other elements. The agent observes these labels when it steps on their locations. Three tasks with different goals are defined on this environment: COFFEE (deliver coffee to the office), COFFEEMAIL (deliver coffee and mail to the office), and VISITABCD (visit A, B, C and D in order). The tasks terminate when the goal is achieved or a $*$ is broken (this is a dead-end state).



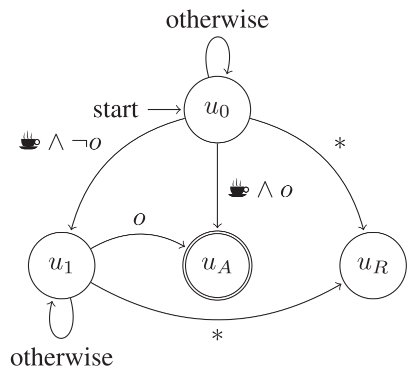
(a)

Execution trace
$T = \langle s_{4,6}, \leftarrow, 0, s_{3,6}, \rightarrow, 0,$
$s_{4,6}, \downarrow, 0, s_{4,5}, \downarrow, 1, s_{4,4}\rangle$

Observation trace
$T_{L,\mathcal{O}} = \langle \{\}, \{☕\}, \{\}, \{\}, \{o\}\rangle$

Compressed observation trace
$\hat{T}_{L,\mathcal{O}} = \langle \{\}, \{☕\}, \{\}, \{o\}\rangle$

(b)



(c)

Figure 1: The OFFICEWORLD environment (Toro Icarte et al. 2018). Figure (a) is an example grid, (b) shows a positive execution trace in the example grid for the COFFEE task and its derived observation traces, and (c) shows the COFFEE automaton.

Traces

An *execution trace* T is a finite state-action-reward sequence $T = \langle s_0, a_0, r_1, s_1, a_1, \dots, a_{n-1}, r_n, s_n\rangle$ induced by a (changing) policy during an episode. An execution trace is *positive* if $s_n \in S_G$, *negative* if $s_n \in S_T \setminus S_G$, and *incomplete* if $s_n \notin S_T$, denoted by T^+, T^- and T^I respectively.

An *observation trace* $T_{L,\mathcal{O}}$ is a sequence of observation sets $O_i \subseteq \mathcal{O}, 0 \leq i \leq n$, obtained by applying a labeling function L to each state s_i in an execution trace T. A *compressed observation trace* $\hat{T}_{L,\mathcal{O}}$ is the result of removing contiguous duplicated observation sets from $T_{L,\mathcal{O}}$. Figure 1b shows an example of a positive execution trace for the COFFEE task together with the derived observation trace and the resulting compressed trace.

A set of execution traces is a tuple $\mathcal{T} = \langle \mathcal{T}^+, \mathcal{T}^-, \mathcal{T}^I\rangle$, where $\mathcal{T}^+$, $\mathcal{T}^-$ and $\mathcal{T}^I$ are sets of positive, negative and incomplete traces, respectively. The sets of observation and compressed observation traces are denoted $\mathcal{T}_{L,\mathcal{O}}$ and $\hat{\mathcal{T}}_{L,\mathcal{O}}$.

Subgoal Automata

A *subgoal automaton* is a tuple $\mathcal{A} = \langle U, \mathcal{O}, \delta, u_0, u_A, u_R\rangle$ where U is a finite set of states, $\mathcal{O}$ is a set of observables (or alphabet), $\delta : U \times 2^{\mathcal{O}} \to U$ is a deterministic transition function that takes as arguments a state and a subset of observables and returns a state, $u_0 \in U$ is a start state, $u_A \in U$ is the unique accepting state, and $u_R \in U$ is the unique rejecting state.

A subgoal automaton $\mathcal{A}$ *accepts* an observation trace $T_{L,\mathcal{O}} = \langle O_0, \dots, O_n\rangle$ if there exists a sequence of automaton states $u_0, \dots, u_{n+1}$ in U such that (1) $\delta(u_i, O_i) = u_{i+1}$ for $i = 0, \dots, n$, and (2) $u_{n+1} = u_A$. Analogously, $\mathcal{A}$ *rejects* $T_{L,\mathcal{O}}$ if $u_{n+1} = u_R$.

When a subgoal automaton is used together with an MDP, the actual states are (s, u) pairs where $s \in S$ and $u \in U$. Therefore, actions are selected according to a policy $\pi : S \times U \to \Delta(A)$ where $\pi(a|s, u)$ is the probability for taking action $a \in A$ at the MDP state $s \in S$ and the automaton state $u \in U$. At each step, the agent transitions to (s', u') where s' is the result of applying an action $a \in A$ in s, while $u' = \delta(u, L(s'))$. Then, the agent receives reward 1 if $u' = u_A$ and 0 otherwise.

Figure 1c shows the automaton for the COFFEE task of the OFFICEWORLD domain. Each transition is labeled with a logical condition $\varphi \in 3^{\mathcal{O}}$ that expresses the subgoal to be achieved[2]. The sequence of visited automaton states for the trace in Figure 1b would be $\langle u_0, u_1, u_1, u_1, u_A\rangle$.

Relationship with options. Each state $u \in U$ in a subgoal automaton encapsulates an option $\omega_u = \langle I_u, \pi_u, \beta_u\rangle$ whose policy π_u attempts to satisfy an outgoing transition's condition. Formally, the option termination condition β_u is:

$$\beta_u(s) = \begin{cases} 1 \text{ if } \exists u' \neq u, \varphi \in 3^{\mathcal{O}} | L(s') \models \varphi, \delta(u, \varphi) = u' \\ \qquad 0 \text{ otherwise} \end{cases}.$$

That is, the option at automaton state $u \in U$ finishes at MDP state $s \in S$ if there is a transition to a different automaton state $u' \in U$ such that the transition's condition φ is satisfied by the next observations $L(s')$. Note that at most one transition can be made true since the automaton is deterministic.

The initiation set I_u is formed by all those states that satisfy the incoming conditions:

$$I_u = \{s \in S \mid \delta(u', \varphi) = u, L(s) \models \varphi, u \neq u', u \neq u_0\}.$$

In the particular case of the initial automaton state $u_0 \in U$, its initiation set $I_{u_0} = S$ is the whole state space since there is no restriction imposed by any previous automaton state.

Note that we do not add options to the set of primitive actions, which would make the decision process more complex since more alternatives are available. Instead, subgoal automata keep the action set unchanged, and which option to apply is determined by the current automaton state.

Learning Subgoal Automata from Traces

This section describes our approach for learning an automaton. We formalize the automaton learning task as a

[2]Note that $\varphi \in 3^{\mathcal{O}}$ because each observable can appear as positive or negative, or not appear in the condition.

tuple $T_{\mathcal{A}} = \langle U, \mathcal{O}, \mathcal{T}_{L,\mathcal{O}} \rangle$, where $U \supseteq \{u_0, u_A, u_R\}$ is a set of states, $\mathcal{O}$ is a set of observables, and $\mathcal{T}_{L,\mathcal{O}} = \langle \mathcal{T}^{+}_{L,\mathcal{O}}, \mathcal{T}^{-}_{L,\mathcal{O}}, \mathcal{T}^{I}_{L,\mathcal{O}} \rangle$ is a set of (possibly compressed) observation traces (abbreviated as *traces* below). An automaton $\mathcal{A}$ is a solution of $T_{\mathcal{A}}$ if and only if accepts all positive traces $\mathcal{T}^{+}_{L,\mathcal{O}}$, rejects all negative traces $\mathcal{T}^{-}_{L,\mathcal{O}}$, and neither accepts nor rejects incomplete traces $\mathcal{T}^{I}_{L,\mathcal{O}}$.

The automaton learning task $T_{\mathcal{A}}$ is mapped into an ILASP task $M(T_{\mathcal{A}}) = \langle B, S_M, E \rangle$. Then, the ILASP system is used to learn the smallest set of transitions (i.e., a minimal hypothesis) that covers the example traces.

We define the components of the ILASP task $M(T_{\mathcal{A}})$ below. An ILASP task specifies the maximum size of a rule body. To allow for an arbitrary number of literals in the bodies, we learn the negation $\bar{\delta}$ of the actual transitions δ. Thus, we do not limit the number of conjuncts, but the number of disjuncts (i.e., the number of edges between two states). We denote the maximum number of disjuncts by $\max|\delta(x,y)|$.

Hypothesis space. The hypothesis space S_M is formed by two kinds of rules:

1. Facts of the form $\mathtt{ed(X,Y,E)}$. indicating there is a transition from state $\mathtt{X} \in U \setminus \{u_A, u_R\}$ to state $\mathtt{Y} \in U \setminus \{\mathtt{X}\}$ using edge $\mathtt{E} \in [1, \max|\delta(x,y)|]$.
2. Normal rules whose *head* is of the form $\bar{\delta}(\mathtt{X,Y,E,T})$ stating that the conditions of the transition from state $\mathtt{X} \in U \setminus \{u_A, u_R\}$ to state $\mathtt{Y} \in U \setminus \{\mathtt{X}\}$ in edge $\mathtt{E} \in [1, \max|\delta(x,y)|]$ *do not* hold at time $\mathtt{T}$. These conditions are specified in the *body*, which is a conjunction of $\mathtt{obs(O,T)}$ literals indicating that observable $\mathtt{O} \in \mathcal{O}$ is seen at time $\mathtt{T}$. The atom $\mathtt{step(T)}$ expresses that $\mathtt{T}$ is a timestep. The body must contain at least one $\mathtt{obs}$ literal.

Note that the hypothesis space does not include (i) loop transitions, and (ii) transitions from u_A and u_R. Later, we define (i) in the absence of external transitions.

Given a subgoal automaton $\mathcal{A}$, we denote the set of ASP rules that describe it by $M(\mathcal{A})$. The rules below correspond to the (u_0, u_1) and (u_0, u_A) transitions in Figure 1c:

$\mathtt{ed(u_0,u_1,1).\ ed(u_0,u_A,1).}$
$\bar{\delta}(\mathtt{u_0,u_1,1,T}) \mathtt{:-\,not\ obs(☕,T),step(T).}$
$\bar{\delta}(\mathtt{u_0,u_1,1,T}) \mathtt{:-\,obs(}o\mathtt{,T),step(T).}$
$\bar{\delta}(\mathtt{u_0,u_A,1,T}) \mathtt{:-\,not\ obs(}o\mathtt{,T),step(T).}$
$\bar{\delta}(\mathtt{u_0,u_A,1,T}) \mathtt{:-\,not\ obs(☕,T),step(T).}$

Examples. Given $\mathcal{T}_{L,\mathcal{O}} = \langle \mathcal{T}^{+}_{L,\mathcal{O}}, \mathcal{T}^{-}_{L,\mathcal{O}}, \mathcal{T}^{I}_{L,\mathcal{O}} \rangle$, the example set is defined as $E = \{\langle e^*, C_{T_{L,\mathcal{O}}} \rangle \mid * \in \{+,-,I\}, T_{L,\mathcal{O}} \in \mathcal{T}^{*}_{L,\mathcal{O}}\}$, where $e^+ = \langle \{\mathtt{accept}\}, \{\mathtt{reject}\} \rangle$, $e^- = \langle \{\mathtt{reject}\}, \{\mathtt{accept}\} \rangle$ and $e^I = \langle \{\}, \{\mathtt{accept}, \mathtt{reject}\} \rangle$ are the partial interpretations for positive, negative and incomplete examples. The $\mathtt{accept}$ and $\mathtt{reject}$ atoms express whether a trace is accepted or rejected by the automaton; hence, positive traces must only be accepted, negative traces must only be rejected, and incomplete traces cannot be accepted or rejected.

Given a trace $T_{L,\mathcal{O}} = \langle O_0, \ldots, O_n \rangle$, a context is defined as: $C_{T_{L,\mathcal{O}}} = \{\mathtt{obs(O,T).} \mid \mathtt{O} \in O_{\mathtt{T}}, O_{\mathtt{T}} \in T_{L,\mathcal{O}}\} \cup \{\mathtt{last(}n\mathtt{).}\}$, where $\mathtt{last(}n\mathtt{)}$ indicates that the trace ends at time n. We denote by $M(T_{L,\mathcal{O}})$ the set of ASP facts that describe the trace $T_{L,\mathcal{O}}$. For example, $M(T_{L,\mathcal{O}}) = \{\mathtt{obs(a,0).\ obs(b,2).\ obs(c,2).\ last(2).}\}$ for the trace $T_{L,\mathcal{O}} = \langle \{a\}, \{\}, \{b,c\} \rangle$.

Background knowledge. The next paragraphs describe the background knowledge B components: $B = B_U \cup B_{\mathtt{step}} \cup B_\delta \cup B_{\mathtt{st}}$. First, B_U is a set of facts of the form $\mathtt{state(u)}$. for each $\mathtt{u} \in U$. The set $B_{\mathtt{step}}$ defines a fluent $\mathtt{step(T)}$ for $0 \leq \mathtt{T} \leq \mathtt{T}' + 1$ where $\mathtt{T}'$ is the step for which $\mathtt{last(T')}$ is defined.

The subset B_δ defines rules for the automaton transition function. The first rule defines all the possible edge identifiers, which is limited by the maximum number of edges between two states. The second rule states that there is an external transition from state $\mathtt{X}$ at time $\mathtt{T}$ if there is a transition from $\mathtt{X}$ to a different state $\mathtt{Y}$ at that time. The third rule is a frame axiom: state $\mathtt{X}$ transitions to itself at time $\mathtt{T}$ if there are no external transitions from it at that time. The fourth rule defines the positive transitions in terms of the learned negative transitions $\bar{\delta}$ and $\mathtt{ed}$ atoms. The fifth rule preserves determinism: two transitions from $\mathtt{X}$ to two different states $\mathtt{Y}$ and $\mathtt{Z}$ cannot hold at the same time. The sixth rule forces all non-terminal states to have an edge to another state.

$$B_\delta = \left\{ \begin{array}{l} \mathtt{edge\_id(1..}\max|\delta(\mathtt{x,y})|\mathtt{).} \\ \mathtt{ext\_}\delta\mathtt{(X,T)\,:-\,}\delta\mathtt{(X,Y,\_,T),X!{=}Y.} \\ \delta\mathtt{(X,X,1,T)\,:-\,not\ ext\_}\delta\mathtt{(X,T),state(X),step(T).} \\ \delta\mathtt{(X,Y,E,T)\,:-\,ed(X,Y,E),not\ }\bar{\delta}\mathtt{(X,Y,E,T),step(T).} \\ \mathtt{:-\,}\delta\mathtt{(X,Y,\_,T),}\delta\mathtt{(X,Z,\_,T),Y!{=}Z.} \\ \mathtt{:-\,not\ ed(X,\_,\_),state(X),X!{=}u_A,X!{=}u_R.} \end{array} \right\}$$

The subset $B_{\mathtt{st}}$ uses $\mathtt{st(T,X)}$ atoms, indicating that the agent is in state $\mathtt{X}$ at time $\mathtt{T}$. The first rule says that the agent is in $\mathtt{u_0}$ at time $\mathtt{0}$. The second rule determines that at time $\mathtt{T{+}1}$ the agent will be in state $\mathtt{Y}$ if it is in a non-terminal state $\mathtt{X}$ at time $\mathtt{T}$ and a transition between them holds. The third (resp. fourth) rule defines that the example is accepted (resp. rejected) if the state at the trace's last timestep is $\mathtt{u_A}$ (resp. $\mathtt{u_R}$).

$$B_{\mathtt{st}} = \left\{ \begin{array}{l} \mathtt{st(0,u_0).} \\ \mathtt{st(T{+}1,Y)\,:-\,st(T,X),}\delta\mathtt{(X,Y,\_,T),X!{=}u_A,X!{=}u_R.} \\ \mathtt{accept\,:-\,last(T),st(T{+}1,u_A).} \\ \mathtt{reject\,:-\,last(T),st(T{+}1,u_R).} \end{array} \right\}$$

Lemma 1 and Theorem 1 capture the correctness of the encoding. We omit the proofs for brevity.

Lemma 1 (Correctness of the ASP encoding)**.** *Given an automaton $\mathcal{A}$ and a finite trace $T^*_{L,\mathcal{O}}$, where $* \in \{+,-,I\}$, $M(\mathcal{A}) \cup B \cup M(T^*_{L,\mathcal{O}})$ has a unique answer set S and (1)* $\mathtt{accept} \in S$ *iff $* = +$, and (2)* $\mathtt{reject} \in S$ *iff $* = -$.*

Theorem 1. *Given an automaton learning task $T_{\mathcal{A}} = \langle U, \mathcal{O}, \mathcal{T}_{L,\mathcal{O}} \rangle$, an automaton $\mathcal{A}$ is a solution of $T_{\mathcal{A}}$ iff $M(\mathcal{A})$ is an inductive solution of $M(T_{\mathcal{A}}) = \langle B, S_M, E \rangle$.*

Interleaved Automata Learning Algorithm

This section describes ISA (**I**nduction of **S**ubgoal **A**utomata for Reinforcement Learning), a method that combines reinforcement and automaton learning. First, we describe the RL algorithm that exploits the automaton structure. Second, we explain how these two learning components are mixed.

Reinforcement learning algorithm. The RL algorithm we use to exploit the automata structure is QRM (Q-learning for Reward Machines) (Toro Icarte et al. 2018). QRM maintains a Q-function for each automaton state, which are updated with Q-learning updates of the form:

$$Q_u(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a'} Q_{u'}(s',a'),$$

where, in our case, $r = 1$ if $u' = u_A$ and 0 otherwise. Note that the bootstrapped action-value depends on the next automaton state u'.

QRM performs this update for all the automaton states, so all policies are simultaneously updated based on the (s, a, s') experience. Note this is a form of intra-option learning (Sutton, Precup, and Singh 1998): we update the policies of all the states from the experience generated by a single state's policy. In the tabular case, QRM is guaranteed to converge to an optimal policy in the limit. Note that QRM (and thus, ISA) is still applicable in domains with large state spaces by having a Deep Q-Network (Mnih et al. 2015) in each automaton state instead of a Q-table.

Algorithm 1 ISA algorithm for a single task

```
 1: $\mathcal{A} \leftarrow$ INITAUTOMATON($\{u_0, u_A, u_R\}$)
 2: $\mathcal{T}_{L,\mathcal{O}} \leftarrow \{\}$                          ▷ Set of counterexamples
 3: INITQFUNCTIONS($\mathcal{A}$)
 4: for $l = 0$ to num_episodes do
 5:     $s \leftarrow$ ENVINITIALSTATE()
 6:     $u_p \leftarrow \delta(u_0, L(s))$
 7:     $T_{L,\mathcal{O}} \leftarrow \langle L(s) \rangle$                      ▷ Initialize trace
 8:     if ISCOUNTEREXAMPLE($s, u_p$) then
 9:         ONCOUNTEREXAMPLEFOUND($T_{L,\mathcal{O}}$)
10:         $u_p \leftarrow \delta(u_0, L(s))$
11:     for $t = 0$ to length_episode do
12:         $a, s' \leftarrow$ ENVSTEP($s, u_p$)
13:         $u_q \leftarrow \delta(u_p, L(s'))$
14:         UPDATEOBSTRACE($L(s'), T_{L,\mathcal{O}}$)
15:         if ISCOUNTEREXAMPLE($s', u_q$) then
16:             ONCOUNTEREXAMPLEFOUND($T_{L,\mathcal{O}}$)
17:             break
18:         else
19:             UPDATEQFUNCTIONS($s$, $a$, $s'$, $L(s')$)
20:             $s \leftarrow s'; u_p \leftarrow u_q$
21: function ONCOUNTEREXAMPLEFOUND($T_{L,\mathcal{O}}$)
22:     $\mathcal{T}_{L,\mathcal{O}} \leftarrow \mathcal{T}_{L,\mathcal{O}} \cup \{T_{L,\mathcal{O}}\}$
23:     $\mathcal{A} \leftarrow$ FINDMINIMALAUTOMATON($\mathcal{T}_{L,\mathcal{O}}$)
24:     INITQFUNCTIONS($\mathcal{A}$)
```

ISA algorithm. Algorithm 1 is the ISA pseudocode for a single task and is explained below:

1. The initial automaton (line 1) has initial state u_0, accepting state u_A and rejecting state u_R. The automaton does not accept nor reject anything. The set of counterexample traces and the Q-functions are initialized (lines 2-3).
2. When an episode starts, the current automaton state u_p is u_0. One transition is applied depending on the agent's initial observations $L(s)$ (lines 5-6). In Figure 1c, if the agent initially observes {☕}, the actual initial state is u_1.
3. At each step, we select an action a in state s using an ϵ-greedy policy (line 12), and update the automaton state u_p and observation trace $T_{L,\mathcal{O}}$ (lines 13-14). If no counterexample is found (line 18), the Q-functions of all automaton states are updated (line 19) and the episode continues.
4. Let u be the current automaton state and s the MDP state. A counterexample trace is found (lines 8, 15) if (a) multiple outgoing transitions from u hold, or (b) the automaton does not correctly recognize s (e.g., $s \in S_G \wedge u \neq u_A$).
5. If a counterexample trace $T_{L,\mathcal{O}}$ is found (lines 21-24):
   (a) Add it to $\mathcal{T}^{+}_{L,\mathcal{O}}$ if $s \in S_G$, to $\mathcal{T}^{-}_{L,\mathcal{O}}$ if $s \in S_T \setminus S_G$ and to $\mathcal{T}^{I}_{L,\mathcal{O}}$ if $s \notin S_T$ (line 22).
   (b) Run the automaton learner (line 23), using iterative deepening to select the number of automaton states.
   - When a new automaton is learned, we reset all the Q-functions (e.g., setting all Q-values to 0) (line 24).
   - If a counterexample is detected at the beginning of the episode (line 8), the automaton state is reset (line 10), else the episode ends (line 17).

ISA does not start learning automata until we find a positive example (i.e., the goal is achieved). Resetting all the Q-functions causes the agent to forget everything it learned. To mitigate the forgetting effect and further exploit the automata structure, we employ reward shaping.

Reward shaping. Ng, Harada, and Russell (1999) proposed a function that provides the agent with additional reward to guide its learning process while guaranteeing that optimal policies remain unchanged:

$$F(s,a,s') = \gamma\Phi(s') - \Phi(s),$$

where γ is the MDP's discount factor and $\Phi : S \rightarrow \mathbb{R}$ is a real-valued function. The automata structure can be exploited by defining $F : U \times U \rightarrow \mathbb{R}$ in terms of the automaton states instead of the MDP states:

$$F(u,u') = \gamma\Phi(u') - \Phi(u),$$

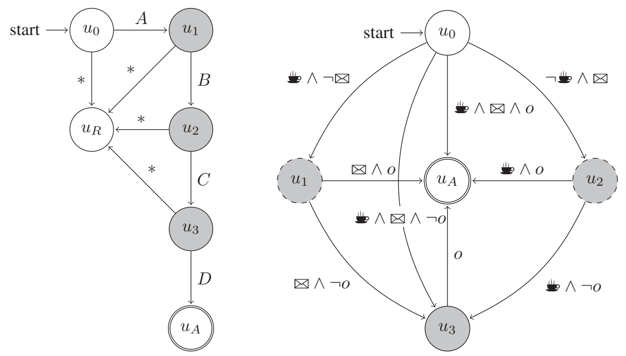
where $\Phi : U \rightarrow \mathbb{R}$. Intuitively, we want F's output to be high when the agent gets closer to u_A. Thus, we define Φ as

$$\Phi(u) = |U| - d(u, u_A),$$

where $|U|$ is the number of states in the automaton (an upper bound for the maximum finite distance between u_0 and u_A), and $d(u, u_A)$ is the distance between state u and u_A. If u_A is unreachable from a state u, then $d(u, u_A) = \infty$.

Theorem 2 shows that if the target automata is in the hypothesis space, there will only be a finite number of learning steps in the algorithm before it converges on the target automata (or an equivalent automata).

Theorem 2. *Given a target finite automaton $\mathcal{A}_*$, there is no infinite sequence σ of automaton-counterexample pairs $\langle \mathcal{A}_i, e_i \rangle$ such that $\forall i$: (1) $\mathcal{A}_i$ covers all examples $e_1, \ldots, e_{i-1}$, (2) $\mathcal{A}_i$ does not cover e_i, and (3) $\mathcal{A}_i$ is in the finite hypothesis space S_M.*



(a) VISITABCD (b) COFFEEMAIL

Figure 2: Automata for two OFFICEWORLD tasks. Self-loops and transitions to u_R in (b) are omitted. The shaded state IDs can be interchanged without B_{sym}. The dashed state IDs are still interchangeable even when using B_{sym}.

Proof. By contradiction. Assume that σ is infinite. Given that S_M is finite, the number of possible automata is finite. Hence, some automaton $\mathcal{A}$ must appear in σ at least twice, say as $\mathcal{A}_i = \mathcal{A}_j, i < j$. By definition, $\mathcal{A}_i$ does not cover e_i and $\mathcal{A}_j$ covers e_i. This is a contradiction. □

4 Experimental Results

We evaluate[3] ISA using the OFFICEWORLD domain and the three tasks introduced in Section 3. The automata we compute using our method are forced to be *acyclic*. Besides, we add a simple *symmetry breaking* constraint B_{sym} to the task's background knowledge to avoid considering isomorphic automata. Figure 2 shows two automata whose state IDs u_1, u_2 and u_3 can be used indifferently. Our symmetry breaking method (1) assigns an integer index to each state[4] and (2) imposes that states must be visited in increasing order of indices. For example, if we assign indices 0…3 to $u_0, \ldots, u_3$, positive traces in Figure 2a always yield the sequence $\langle u_0, u_1, u_2, u_3, u_A \rangle$. However, this is not enough to break all symmetries in Figure 2b: u_1 and u_2 can still be switched since they cannot be in the same path to u_A or u_R.

Tabular Q-learning is used to learn the Q-function at each automaton state with parameters $\alpha = 0.1$, $\epsilon = 0.1$, and $\gamma = 0.99$. The agent's state is its position. ISA receives a set of 100 randomly generated grids[5]. One episode is run per grid in sequential order until reaching 20,000 episodes for each grid. The maximum episode length is 100 steps.

For some experiments we consider the multitask setting, where an automaton is learned for every task from the set of grids. Thus, there is a Q-table for each task-grid-automaton state triplet updated at every step. One episode is run per task-grid until 20,000 episodes are performed for each pair.

[3]Code: github.com/ertsiger/induction-subgoal-automata-rl.

[4]u_0 has the lowest value, while u_A and u_R have the highest.

[5]Each grid has the same size and walls as Figure 1a. The observables and the agent are randomly placed.

When reward shaping is on, the shaping function's output is set to -100 in case it is $-\infty$. This occurs when the next automaton state is u_R since there is no path from it to u_A.

The different settings are referenced as **S** (single task), **M** (multitask) and **R** (reward shaping), all using the same set of 100 random grids. We execute 10 runs for each setting.

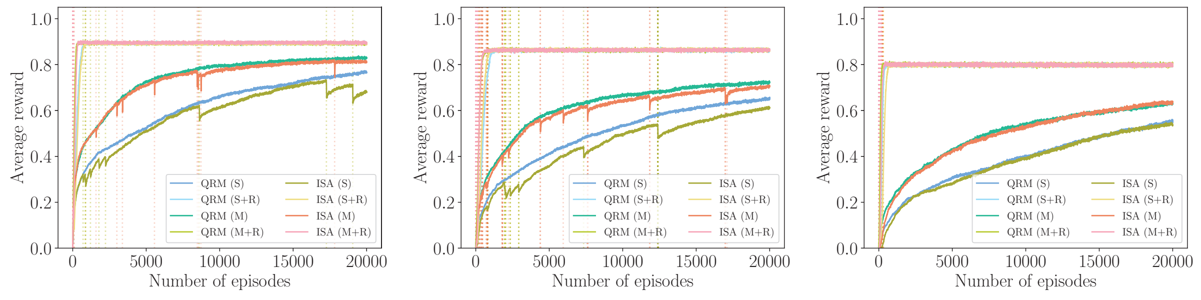
We use ILASP to learn the automata from compressed observation traces and set $\max |\delta(x, y)|$ to 1. All experiments ran on 3.40GHz Intel® Core$^{\text{TM}}$ i7-6700 processors.

ISA performs similarly to QRM. Figure 3 shows the tasks' average learning curve for ISA and QRM (where the automaton is given beforehand) across 10 runs. The ISA curves converge similarly to the analogous QRM's. When reward shaping is used, convergence speeds up dramatically. The multitask settings converge faster since an agent is also trained from other agents' experiences in different tasks.

The vertical lines are episodes where an automaton is learned, and often occur during the first episodes: this is the main reason why the learning and non-learning curves are similar. Less frequently, automata learning also happens at intermediate phases in the COFFEE and COFFEEMAIL tasks which make the agent forget what it learned. In these cases, recovery from forgetting happens faster in the multitask settings because of the reason above. Reward shaping has a positive effect on the learner: not only is convergence faster, it also helps the agent to discover helpful traces earlier.

ISA's automata learning results. Table 1 shows the average number of examples needed to learn the final automata for setting **S** (the results for other settings are similar). Table 2 shows the average time needed for computing *all* the automata, which is negligible with respect to ISA's total running time. For both tables, the standard error is shown in brackets. First, we observe that the most complex tasks (COFFEEMAIL and VISITABCD) need a higher number of examples and more time to compute their corresponding automata. However, while the total number of examples for these tasks are similar, the time is higher for VISITABCD possibly because the longest path from u_0 to u_A is longer than in COFFEEMAIL. Second, we see that the number of positive and incomplete examples are usually the smallest and the largest respectively. Note that the number of positive examples is approximately equal to the number of paths from u_0 to u_A. The paths described by the positive examples are refined through the negative and incomplete ones. Finally, Table 2 shows slight variations of time across settings. The different experimental settings and the exploratory nature of the agent are responsible for coming up with different counterexamples and, thus, cause these variations. There is not a clear setting which leads to faster automata learning across domains. The design of exploratory strategies that speed up automata learning is possible future work.

ISA learns automata faster with few observables. To test the effect of the observable set $\mathcal{O}$ on ILASP's performance, we run the experiments using setting **S** but employing only the observables that each task needs, e.g., $\mathcal{O} = \{☕, o, *\}$ in the COFFEE task. The biggest changes occur in COFFEEMAIL, where the total number of examples is 46%



(a) COFFEE (b) COFFEEMAIL (c) VISITABCD

Figure 3: Learning curves for different OFFICEWORLD tasks. The vertical lines are episodes where an automaton is learned.

	All	+	-	I
COFFEE	6.6 (0.5)	2.2 (0.2)	2.3 (0.2)	2.1 (0.3)
COFFEEMAIL	34.5 (2.9)	5.5 (0.4)	9.9 (0.9)	19.1 (2.2)
VISITABCD	32.5 (2.1)	1.7 (0.2)	11.6 (0.8)	19.2 (1.7)

Table 1: Average number of examples needed for setting **S**.

	S	**S+R**	**M**	**M+R**
COFFEE	0.5 (0.0)	0.4 (0.0)	0.3 (0.0)	0.4 (0.0)
COFFEEMAIL	43.3 (12.1)	36.9 (6.0)	24.8 (3.6)	24.6 (2.7)
VISITABCD	63.0 (11.4)	68.5 (13.0)	48.4 (8.8)	69.6 (8.1)

Table 2: Average ILASP running time.

smaller. The sets of positive, negative and incomplete examples are 27%, 42% and 53% smaller. Besides, the automata are computed 92% faster. Thus, we see that the number of observables has an impact on the performance: the RL process is halted less frequently and automata are learned faster. This performance boost in COFFEEMAIL can be due to the fact that it has more paths to u_A; thus, irrelevant symbols do not need to be discarded for each of these. This is confirmed by the fact that while the number of positives is roughly the same, the number of incomplete examples greatly decreases.

5 Related Work

Hierarchical RL (HRL). Our method is closely related to the options framework for HRL, and indeed we define one option per automaton state. The key difference from other HRL approaches, like HAMs (Parr and Russell 1997), MAXQ (Dietterich 2000) and the common way of using options, is that we do not learn a high-level policy for selecting among options. Rather, the high-level policy is implicitly represented by the automaton, and the option to execute is fully determined by the current automaton state. Consequently, while HRL policies may be suboptimal in general, the QRM algorithm we use converges to the optimal policy.

Our approach is similar to HAMs in that they also use an automaton. However, HAMs are non-deterministic automata whose transitions can invoke lower level machines and are not labeled by observables (the high-level policy consists in deciding which transition to fire). Leonetti, Iocchi, and Patrizi (2012) synthesize a HAM from the set of shortest solutions to a non-deterministic planning problem, and use it to refine the choices at non-deterministic points through RL.

Option discovery. ISA is similar to bottleneck option discovery methods, which find "bridges" between regions of the state space. In particular, ISA finds conditions that connect two of these regions. McGovern and Barto (2001) use diverse density to find landmark states in state traces that achieve the task's goal. This approach is similar to ours because (1) it learns from traces; (2) it classifies traces into different categories; and (3) it interleaves option discovery and learning.

Just like some option discovery methods (McGovern and Barto 2001; Stolle and Precup 2002), our approach requires the task to be solved at least once. Other methods (Menache, Mannor, and Shimkin 2002; Şimşek and Barto 2004; Şimşek, Wolfe, and Barto 2005; Machado, Bellemare, and Bowling 2017) discover options without solving the task.

Grammars are an alternative to automata for expressing formal languages. Lange and Faisal (2019) induce a straight-line grammar from action traces to discover macro-actions.

Reward machines (RMs). Subgoal automata are similar to RMs (Toro Icarte et al. 2018). There are two differences: (1) RMs do not have explicit accepting and rejecting states, and (2) RMs use a reward-transition function $\delta_r : U \times U \to \mathbb{R}$ that returns the reward for taking a transition between two automaton states. Note that our automata are a specific case of the latter where $\delta_r(\cdot, u_A) = 1$ and 0 otherwise.

Recent work has focused on learning RMs from experience using discrete optimization (Toro Icarte et al. 2019) and grammatical inference algorithms (Xu et al. 2019). While these approaches can get stuck in a local optima, ISA returns a minimal automaton each time it is called. Just as our method, they use an 'a priori' specified set of observables.

In contrast to the other learning approaches, in the future we can leverage ILP to (1) transfer knowledge between automata learning tasks (e.g., providing a partial automaton as the background knowledge), (2) support examples generated by a noisy labeling function, and (3) easily express more complex conditions (e.g., using first-order logic).

Camacho et al. (2019) convert reward functions expressed in various formal languages into RMs, and propose a reward shaping method that runs value iteration on the RM states.

6 Conclusions and Future Work

In this paper we have proposed ISA, an algorithm for learning subgoals by inducing a deterministic finite automaton from observation traces seen by the agent. The automaton structure can be exploited by an existing RL algorithm to increase sample efficiency and transfer learning. We have shown that our method performs comparably to an algorithm where the automaton is given beforehand.

Improving the scalability is needed to handle more complex tasks requiring automata with cycles or longer examples. In the future, we will further explore symmetry breaking techniques to reduce the hypothesis space (Drescher, Tifrea, and Walsh 2011) and other approaches automata learning, like RNNs (Weiss, Goldberg, and Yahav 2018; Michalenko et al. 2019). Discovering the observables used by the automata is also an interesting path for future work.

Acknowledgments

Anders Jonsson is partially supported by the Spanish grants TIN2015-67959 and PCIN-2017-082.

References

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *ICAPS*.

Camacho, A.; Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2019. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *IJCAI*.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *JAIR* 13:227–303.

Drescher, C.; Tifrea, O.; and Walsh, T. 2011. Symmetry-breaking Answer Set Solving. *AI Commun.* 24(2):177–194.

Gelfond, M., and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press.

Ho, M. K.; Abel, D.; Griffiths, T. L.; and Littman, M. L. 2019. The value of abstraction. *Current Opinion in Behavioral Sciences* 29:111 – 116. SI: 29: Artificial Intelligence (2019).

Hu, Y., and De Giacomo, G. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *IJCAI*.

Konidaris, G. 2019. On the necessity of abstraction. *Current Opinion in Behavioral Sciences* 29:1 – 7. SI: 29: Artificial Intelligence (2019).

Lange, R. T., and Faisal, A. 2019. Semantic RL with Action Grammars: Data-Efficient Learning of Hierarchical Task Abstractions . *CoRR* abs/1907.12477.

Law, M.; Russo, A.; and Broda, K. 2015. The ILASP System for Learning Answer Set Programs.

Law, M.; Russo, A.; and Broda, K. 2016. Iterative Learning of Answer Set Programs from Context Dependent Examples. *TPLP* 16(5-6):834–848.

Leonetti, M.; Iocchi, L.; and Patrizi, F. 2012. Automatic Generation and Learning of Finite-State Controllers. In *AIMSA*.

Machado, M. C.; Bellemare, M. G.; and Bowling, M. H. 2017. A Laplacian Framework for Option Discovery in Reinforcement Learning. In *ICML*.

McGovern, A., and Barto, A. G. 2001. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *ICML*.

Menache, I.; Mannor, S.; and Shimkin, N. 2002. Q-Cut - Dynamic Discovery of Sub-goals in Reinforcement Learning. In *ECML*.

Michalenko, J. J.; Shah, A.; Verma, A.; Baraniuk, R. G.; Chaudhuri, S.; and Patel, A. B. 2019. Representing Formal Languages: A Comparison Between Finite Automata and Recurrent Neural Networks. *CoRR* abs/1902.10297.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Ng, A. Y.; Harada, D.; and Russell, S. J. 1999. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *ICML*.

Parr, R., and Russell, S. J. 1997. Reinforcement Learning with Hierarchies of Machines. In *NeurIPS*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2018. Computing Hierarchical Finite State Controllers with Classical Planning. *JAIR* 62:755–797.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Şimşek, Ö., and Barto, A. G. 2004. Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning. In *ICML*.

Şimşek, Ö.; Wolfe, A. P.; and Barto, A. G. 2005. Identifying Useful Subgoals in Reinforcement Learning by Local Graph Partitioning. In *ICML*.

Stolle, M., and Precup, D. 2002. Learning Options in Reinforcement Learning. In *SARA*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An introduction*. MIT Press.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1998. Intra-Option Learning about Temporally Abstract Actions. In *ICML*.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1999. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.* 112(1-2):181–211.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2018. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In *ICML*.

Toro Icarte, R.; Waldie, E.; Klassen, T. Q.; Valenzano, R.; Castro, M. P.; and McIlraith, S. A. 2019. Learning Reward Machines for Partially Observable Reinforcement Learning. In *NeurIPS*.

Watkins, C. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge, UK.

Weiss, G.; Goldberg, Y.; and Yahav, E. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *ICML*.

Xu, Z.; Gavran, I.; Ahmad, Y.; Majumdar, R.; Neider, D.; Topcu, U.; and Wu, B. 2019. Joint Inference of Reward Machines and Policies for Reinforcement Learning. *CoRR* abs/1909.05912.