State Abstraction for Programmable Reinforcement Learning Agents

David Andre and Stuart J. Russell
Computer Science Division, UC Berkeley, CA 94720
{dandre,russell}@cs.berkeley.edu

Abstract

Safe state abstraction in reinforcement learning allows an agent to ignore aspects of its current state that are irrelevant to its current decision, and therefore speeds up dynamic programming and learning. This paper explores safe state abstraction in hierarchical reinforcement learning, where learned behaviors must conform to a given partial, hierarchical program. Unlike previous approaches to this problem, our methods yield significant state abstraction while maintaining *hierarchical optimality*, i.e., optimality among all policies consistent with the partial program. We show how to achieve this for a partial programming language that is essentially Lisp augmented with nondeterministic constructs. We demonstrate our methods on two variants of Dietterich's taxi domain, showing how state abstraction and hierarchical optimality result in faster learning of better policies and enable the transfer of learned skills from one problem to another.

Introduction

The ability to make decisions based on only *relevant* features is a critical aspect of intelligence. For example, if one is driving a taxi from A to B, decisions about which street to take should not depend on the current price of tea in China; when changing lanes, the traffic conditions matter but not the name of the street; and so on. *State abstraction* is the process of eliminating features to reduce the effective state space; such reductions can speed up dynamic programming and reinforcement learning (RL) algorithms considerably. Without state abstraction, every trip from A to B is a new trip; every lane change is a new task to be learned from scratch.

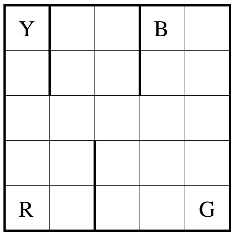
An abstraction is called *safe* if optimal solutions in the abstract space are also optimal in the original space. Safe abstractions were introduced by Amarel (1968) for the Missionaries and Cannibals problem. In our example, the taxi driver can safely omit the price of tea in China from the state space for navigating from A to B. More formally, the value of every state (or of every state-action pair) is independent of the price of tea, so the price of tea is irrelevant in selecting optimal actions. Boutilier *et al.* (1995) developed a general method for deriving such irrelevance assertions from the formal specification of a decision problem.

It has been noted (Dietterich 2000) that a variable can be irrelevant to the optimal decision in a state *even if it affects the value of that state*. For example, suppose that the taxi is driving from A to B to pick up a passenger whose destination is C. Now, C is part of the state, but is not relevant to navigation decisions between A and B. This is because the value (sum of future rewards or costs) of each state between A and B can be *decomposed* into a part dealing with the cost of getting to B and a part dealing with the cost from B to C. The latter part is unaffected by the choice of A; the former part is unaffected by the choice of C.

This idea—that a variable can be irrelevant to *part* of the value of a state—is closely connected to the area of *hierarchical reinforcement learning*, in which learned behaviors must conform to a given partial, hierarchical program. The connection arises because the partial program naturally divides state sequences into parts. For example, the task described above may be achieved by executing two subroutine calls, one to drive from A to B and one to deliver the passenger from B to C. The partial programmer may state (or a Boutilier-style algorithm may derive) the fact that the navigation choices in the first subroutine call are independent of the passenger's final destination. More generally, the notion of *modularity* for behavioral subroutines is precisely the requirement that decisions internal to the subroutine be independent of all external variables other than those passed as arguments to the subroutine.

Several different partial programming languages have been proposed, with varying degrees of expressive power. Expressiveness is important for two reasons: first, an expressive language makes it possible to state complex partial specifications concisely; second, it enables irrelevance assertions to be made at a high level of abstraction rather than repeated across many instances of what is conceptually the same subroutine. The first contribution of this paper is an agent programming language, ALisp, that is essentially Lisp augmented with nondeterministic constructs; the language subsumes MAXQ (Dietterich 2000), options (Precup & Sutton 1998), and the PHAM language (Andre & Russell 2001).

Given a partial program, a hierarchical RL algorithm finds a policy that is consistent with the program. The policy may be *hierarchically optimal*—i.e., optimal among all policies consistent with the program; or it may be *recursively optimal*, i.e., the policy within each subroutine is optimized *ig-*




```
(defun root () (if (not (have-pass)) (get)) (put))
(defun get () (choice get-choice
                (action 'load)
                (call navigate (pickup))))
(defun put () (choice put-choice
                (action 'unload)
                (call navigate (dest))))
(defun navigate(t)
       (loop until (at t) do
             (choice nav (action 'N)
                         (action 'E)
                         (action 'S)
                         (action 'W))))
```

Figure 1: The taxi world. It is a 5x5 world with 4 special cells (RGBY) where passengers are loaded and unloaded. There are 4 features, `x,y,pickup,dest`. In each episode, the taxi starts in a randomly chosen square, and there is a passenger at a random special cell with a random destination. The taxi must travel to, pick up, and deliver the passenger, using the commands `N,S,E,W,load,unload`. The taxi receives a reward of -1 for every action, +20 for successfully delivering the passenger, -10 for attempting to `load` or `unload` the passenger at incorrect locations. The discount factor is 1.0. The partial program shown is an ALisp program expressing the same constraints as Dietterich's taxi MAXQ program. It breaks the problem down into the tasks of getting and putting the passenger, and further isolates navigation.

noring the calling context. Recursively optimal policies may be worse than hierarchically optimal policies if the context is relevant. Dietterich 2000 shows how a two-part decomposition of the value function allows state abstractions that are safe with respect to recursive optimality, and argues that "State abstractions [of this kind] cannot be employed without losing hierarchical optimality." The second, and more important, contribution of our paper is a three-part decomposition of the value function allowing state abstractions that are safe with respect to hierarchical optimality.

The remainder of the paper begins with background material on Markov decision processes and hierarchical RL, and a brief description of the ALisp language. Then we present the three-part value function decomposition and associated Bellman equations. We explain how ALisp programs are annotated with (ir)relevance assertions, and describe a model-free hierarchical RL algorithm for annotated ALisp programs that is guaranteed to converge to hierarchically optimal solutions[1]. Finally, we describe experimental results for this algorithm using two domains: Dietterich's original taxi domain and a variant of it that illustrates the differences between hierarchical and recursive optimality.

Background

Our framework for MDPs is standard (Kaelbling, Littman, & Moore 1996). An MDP is a 4-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ a set of actions, $\mathcal{T}$ a probabilistic transition function mapping $\mathcal{S}\times\mathcal{A}\times\mathcal{S} \to [0,1]$, and $\mathcal{R}$ a reward function mapping $\mathcal{S}\times\mathcal{A}\times\mathcal{S}$ to the reals. We focus on infinite-horizon MDPs with a discount factor β. A solution to an MDP is an optimal policy π^* mapping from $\mathcal{S} \to \mathcal{A}$ and achieves the maximum expected discounted reward. An SMDP (semi-MDP) allows for actions that take more than one time step. $\mathcal{T}$ is now a mapping from $\mathcal{S}\times\mathbf{N}\times\mathcal{S}\times\mathcal{A} \to [0,1]$, where $\mathbf{N}$ is the natural numbers; i.e., it specifies a distribution over both outcome states and action durations. $\mathcal{R}$ then maps from $\mathcal{S}\times\mathbf{N}\times\mathcal{S}\times\mathcal{A}$ to the reals. The expected discounted reward for taking action a in state s and then following policy π is known as the Q value, and is defined as $Q^\pi(s,a) = E[r_0 + \beta r_1 + \beta^2 r_2 + \ldots]$. Q values are related to one another through the Bellman equations (Bellman 1957):

$$Q^\pi(s,a) = \sum_{s',N} \mathcal{T}(s',N,s,a)[\mathcal{R}(s',N,s,a) + \beta^N Q^\pi(s',\pi(s'))].$$

Note that $\pi \in \pi^*$ iff $\forall_s \pi(s) \in \arg\max_a Q^\pi(s,a)$.

In most languages for partial reinforcement learning programs, the programmer specifies a program containing choice points. A *choice point* is a place in the program where the learning algorithm must choose among a set of provided options (which may be primitives or subroutines). Formally, the program can be viewed as a finite state machine with state space Θ (consisting of the stack, heap, and program pointer). Let us define a joint state space Y for a program $\mathcal{H}$ as the cross product of Θ and the states, S, in an MDP $\mathcal{M}$. Let us also define Ω as the set of *choice states*, that is, Ω is the subset of Y where the machine state is at a choice point. With most hierarchical languages for reinforcement learning, one can then construct a joint SMDP $\mathcal{H} \circ \mathcal{M}$ where $\mathcal{H} \circ \mathcal{M}$ has state space Ω and the actions at each state in Ω are the choices specified by the partial program $\mathcal{H}$. For several simple RL-specific languages, it has been shown that policies optimal under $\mathcal{H}\circ\mathcal{M}$ correspond to the best policies achievable in $\mathcal{M}$ given the constraints expressed by $\mathcal{H}$ (Andre & Russell 2001; Parr & Russell 1998).

The ALisp language

The ALisp programming language consists of the Lisp language augmented with three special macros:

- `(choice` *<label> <form0> <form1>* `...)` takes 2 or more arguments, where *<formN>* is a Lisp S-expression. The agent learns which form to execute.
- `(call` *<subroutine> <arg0> <arg1>*`)` calls a subroutine with its arguments and alerts the learning mechanism that a subroutine has been called.
- `(action` *<action-name>*`)` executes a "primitive" action in the MDP.

An ALisp program consists of an arbitrary Lisp program that is allowed to use these macros and obeys the constraint that

[1] Proofs of all theorems are omitted and can be found in an accompanying technical report (Andre & Russell 2002).

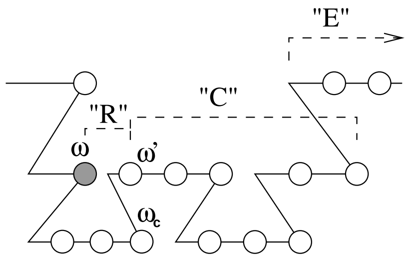


Figure 2: Decomposing the value function for the shaded state, ω. Each circle is a choice state of the SMDP visited by the agent, where the vertical axis represents depth in the hierarchy. The trajectory is broken into 3 parts: the reward "R" for executing the macro action at ω, the completion value "C", for finishing the subroutine, and "E", the external value.

all subroutines that include the choice macro (either directly, or indirectly, through nested subroutine calls) are called with the `call` macro. An example ALisp program is shown in Figure 1 for Dietterich's Taxi world (Dietterich 2000). It can be shown that, under appropriate restrictions (such as that the number of machine states Y stays bounded in every run of the environment), that optimal policies for the joint SMDP $\mathcal{H} \circ \mathcal{M}$ for an ALisp program $\mathcal{H}$ are optimal for the MDP $\mathcal{M}$ among those policies allowed by $\mathcal{H}$ (Andre & Russell 2002).

Value Function Decomposition

A value function decomposition splits the value of a state/action pair into multiple additive components. Modularity in the hierarchical structure of a program allows us to do this decomposition along subroutine boundaries. Consider, for example, Figure 2. The three parts of the decomposition correspond to executing the current action (which might itself be a subroutine), completing the rest of the current subroutine, and all actions outside the current subroutine. More formally, we can write the Q-value for executing action a in $\omega \in \Omega$ as follows:

$$\begin{aligned} Q^\pi(\omega,a) = &\quad E\left[\sum_{t=0}^{\infty}\beta^t r_t\right] \\ &= E\left[\sum_{t=0}^{N_1-1}\beta^t r_t\right] + E\left[\sum_{t=N_1}^{N_2-1}\beta^t r_t\right] + E\left[\sum_{t=N_2}^{\infty}\beta^t r_t\right] \\ &= \quad Q_r^\pi(\omega,a) \quad + \quad Q_c^\pi(\omega,a) \quad + \quad Q_e^\pi(\omega,a) \end{aligned}$$

where N_1 is the number of primitive steps to finish action a, N_2 is the number of primitive steps to finish the current subroutine, and the expectation is over trajectories starting in ω with action a and following π. N_1, N_2, and the rewards, r_t, are defined by the trajectory. Q_r thus expresses the expected discounted reward for doing the current action ("R" from Figure 2), Q_c for completing rest of the current subroutine ("C"), and Q_e for all the reward external to the current subroutine ("E").

It is important to see how this three-part decomposition allows greater state abstraction. Consider the taxi domain,

x	y	pickup	dest	Q	Q_r	Q_c	Q_e
3	3	R	G	0.23	-7.5	-1.0	8.74
3	3	R	B	1.13	-7.5	-1.0	9.63
3	2	R	G	1.29	-6.45	-1.0	8.74

Table 1: Table of Q values and decomposed Q values for 3 states and action $a =$ `(nav pickup)`, where the machine state is equal to {`get-choice`}. The first four columns specify the environment state. Note that although none of the Q values listed are identical, Q_c is the same for all three cases, and Q_e is the same for 2 out of 3, and Q_r is the same for 2 out of 3.

where there are many opportunities for state abstraction (as pointed out by Dietterich (2000) for his two-part decomposition). While completing the `get` subroutine, the passenger's destination is not relevant to decisions about getting to the passenger's location. Similarly, when navigating, only the current x/y location and the target location are important – whether the taxi is carrying a passenger is not relevant. Taking advantage of these intuitively appealing abstractions requires a value function decomposition, as Table 1 shows.

Before presenting the Bellman equations for the decomposed value function, we must first define transition probability measures that take the program's hierarchy into account. First, we have the SMDP transition probability $p(\omega', N|\omega, a)$, which is the probability of an SMDP transition to ω' taking N steps given that action a is taken in ω. Next, let S be a set of states, and let $F_S^\pi(\omega', N|\omega, a)$ be the probability that ω' is the first element of S reached and that this occurs in N primitive steps, given that a is taken in ω and π is followed thereafter. Two such distributions are useful, $F^\pi_{SS(\omega)}$ and $F^\pi_{EX(\omega)}$, where $SS(\omega)$ are those states in the same subroutine as ω and $EX(\omega)$ are those states that are exit points for the subroutine containing ω. We can now write the Bellman equations using our decomposed value function, as shown in Equations 1, 2, and 3 in Figure 3, where $o(\omega)$ returns the next choice state at the parent level of the hierarchy, $i_a(\omega)$ returns the first choice state at the child level, given action a [2], and A_p is the set of actions that are not `calls` to subroutines. With some algebra, we can then prove the following results.

Theorem 1 *If Q_r^*, Q_c^*, and Q_e^* are solutions to Equations 1, 2, and 3 for π^*, then $Q^* = Q_r^* + Q_c^* + Q_e^*$ is a solution to the standard Bellman equation.*

Theorem 2 *Decomposed value iteration and policy iteration algorithms (Andre & Russell 2002) derived from Equations 1, 2, and 3 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

Extending policy iteration and value iteration to work with these decomposed equations is straightforward, but it does require that the full model is known – including $F^\pi_{SS(\omega)}(\omega', N|\omega, a)$ and $F^\pi_{EX(\omega)}(\omega', N|\omega, a)$, which can be found through dynamic programming. After explaining how

[2] We make a trivial assumption that calls to subroutines are surrounded by choice points with no intervening primitive actions at the calling level. $i_a(\omega)$ and $o(\omega)$ are thus simple deterministic functions, determined from the program structure.

$$Q_r^\pi(\omega,a) = \begin{cases} \sum_{\omega',N'} p(\omega',N|\omega,a) r(\omega',N,\omega,a) & \text{if } a \in A_p \\ Q_r^\pi(i_a(\omega),\pi(i_a(\omega))) + Q_c^\pi(i_a(\omega),\pi(i_a(\omega))) & \text{otherwise.} \end{cases} \tag{1}$$

$$Q_c^\pi(\omega,a) = \sum_{(\omega',N)} F^\pi_{SS(\omega)}(\omega',N|\omega,a)\beta^N [Q_r^\pi(\omega',\pi(\omega')) + Q_c^\pi(\omega',\pi(\omega'))] \tag{2}$$

$$Q_e^\pi(\omega,a) = \sum_{(\omega',N)} F^\pi_{EX(\omega)}(\omega',N|\omega,a)\beta^N [Q^\pi(o(\omega'),\pi(o(\omega')))] \tag{3}$$

$$\forall_{a\in A_p} Q_r^*(z_p(\omega,a),a) = \sum_{(\omega',N)} p(\omega',N|\omega,a) r(\omega',N,\omega,a) \tag{4}$$

$$\forall_{a\notin A_p} Q_r^*(z_r(\omega,a),a) = Q_r^*(z_r(\omega'),a') + Q_c^*(z_c(\omega'),a'), \text{ where } \omega' = i_a(\omega) \text{ and } a' = \arg\max_b Q^*(\omega',b) \tag{5}$$

$$\forall_a Q_c^*(z_c(\omega,a),a) = \sum_{(\omega',N)} F^*_{SS(\omega)}(\omega',N|\omega,a)\beta^N [Q_r^*(z_r(\omega',a),a') + Q_c^*(z_c(\omega',a),a')] \text{ where } a' = \arg\max_b Q^*(\omega',b) \tag{6}$$

$$\forall_a Q_e^*(z_e(\omega,a),a) = \sum_{(\omega',N)} F^*_{EX(z_e(\omega,a))}(\omega',N|z_e(\omega,a),a)\beta^N [Q^*(o(\omega'),a')] \text{ where } a' = \arg\max_b Q^*(o(\omega'),b) \tag{7}$$

Figure 3: Top: Bellman equations for the three-part decomposed value function. Bottom: Bellman equations for the abstracted case.

the decomposition enables state abstraction, we will present an online learning method which avoids the problem of having to specify or determine a complex model.

State Abstraction

One method for doing learning with ALisp programs would be to flatten the subroutines out into the full joint state space of the SMDP. This has the result of creating a copy of each subroutine for every place where it is called with different parameters. For example, in the Taxi problem, the flattened program would have 8 copies (4 destinations, 2 calling contexts) of the `navigate` subroutine, each of which have to be learned separately. Because of the three-part decomposition discussed above, we can take advantage of state abstraction and avoid flattening the state space.

To do this, we require that the user specify which features matter for each of the components of the value function. The user must do this for each action at each choice point in the program. We thus annotate the language with `:depends-on` keywords. For example, in the `navigate` subroutine, the `(action 'N)` choice is changed to

```
((action 'N)
  :reward-depends-on nil
  :completion-depends-on '(x y t)
  :external-depends-on '(pickup dest))
```

Note that `t` is the parameter for the target location passed into `navigate`. The Q_r-value for this action is constant – it doesn't depend on any features at all (because all actions in the Taxi domain have fixed cost). The Q_c value only depends on where the taxi currently is and on the target location. The Q_e value only depends on the passenger's location (either in the Taxi or at R,G,B, or Y) and the passenger's destination. Thus, whereas a program with no state abstraction would be required to store 800 values, here, we only must store 117.

Safe state abstraction

Now that we have the programmatic machinery to define abstractions, we'd like to know when a given set of abstraction functions is safe for a given problem. To do this, we first need a formal notation for defining abstractions. Let $z_p[\theta,a]$, $z_r[\theta,a]$, $z_c[\theta,a]$, and $z_e[\theta,a]$ be abstraction functions specifying the set of relevant machine and environment features for each choice point θ and action a for the primitive reward, non-primitive reward, completion cost, and external cost respectively. In the example above, $z_c[nav,N] = \{\mathtt{x},\mathtt{y},\mathtt{t}\}$. Note that this function z groups states together into equivalence classes (for example, all states that agree on assignments to `x`, `y`, and `t` would be in an equivalence class). Let $z(\omega,a)$ be a mapping from a state-action pair to a canonical member of the equivalence class to which it belongs under the abstraction z. We must also discuss how policies interact with abstractions. We will say that a policy π and an abstraction z are consistent iff $\forall_{\omega,a}\pi(\omega) = \pi(z(\omega,a))$ and $\forall_{a,b} z(\omega,a) = z(\omega,b)$. We will denote the set of such policies as Π_z.

Now, we can begin to examine when abstractions are safe. To do this, we define several notions of equivalence.

Definition 1 (P-equivalence) *z_p is P-equivalent (Primitive equivalent) iff* $\forall_{\omega,a\in A_p}$, $Q_r(\omega,a) = Q_r(z_p(\omega,a),a)$.

Definition 2 (R-equivalence) *z_r is R-equivalent iff* $\forall_{\omega,a\notin A_p,\pi\in\Pi_{z_r}}$, $Q_r(\omega,a) = Q_r(z_r(\omega),a)$.

These two specify that states are abstracted together under z_p and z_r only if their Q_r values are equal. C-equivalence can be defined similarly.

For the E component, we can be more aggressive. The exact value of the external reward isn't what's important, rather, it's the behavior that it imposes on the subroutine. For example, in the Taxi problem, the external value after reaching the end of the `navigate` subroutine will be very different when the passenger is in the taxi and when she's not – but the optimal behavior for `navigate` is the same in both cases. Let h be a subroutine of a program $\mathcal{H}$, and let Ω_h be the set of choice states reachable while control remains in h. Then, we can define E-equivalence as follows:

Definition 3 (E-equivalence) *z_e is E-equivalent iff*

1. $\forall_{h\in\mathcal{H}}\forall_{\omega_1,\omega_2\in\Omega_h} z_e[\omega_1] = z_e[\omega_2]$ *and*

2. $\forall_\omega \arg\max_a Q_r^*(\omega,a) + Q_c^*(\omega,a) + Q_e^*(\omega,a) = \arg\max_a Q_r^*(\omega,a) + Q_c^*(\omega,a) + Q_e^*(z_e(\omega,a),a)$.

The last condition says that states are abstracted together only if they have the same set of optimal actions in the set of optimal policies. It could also be described as "passing in enough information to determine the policy". This is the critical constraint that allows us to maintain hierarchical optimality while still performing state abstraction.

We can show that if abstraction functions satisfy these four properties, then the optimal policies when using these abstractions are the same as the optimal policies without them. To do this, we first express the abstracted Bellman equations as shown in Equations 4 - 7 in Figure 3. Now, if z_p z_r, z_c, and z_e are P-, R-,C-, and E-equivalent, respectively, then we can show that we have a safe abstraction.

Theorem 3 *If z_p is P-equivalent, z_r is R-equivalent, z_c is C-equivalent, and z_e is E-equivalent, then, if Q_r^*, Q_c^*, and Q_e^* are solutions to Equations 4 - 7, for MDP $\mathcal{M}$ and ALisp program $\mathcal{H}$, then π such that $\pi(\omega) \in \arg\max_a Q^*(\omega,a)$ is an optimal policy for $\mathcal{H} \circ \mathcal{M}$.*

Theorem 4 *Decomposed abstracted value iteration and policy iteration algorithms (Andre & Russell 2002) derived from Equations 4 - 7 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

Proving these various forms of equivalence might be difficult for a given problem. It would be easier to create abstractions based on conditions about the model, rather than conditions on the value function. Dietterich (2000) defines four conditions for safe state abstraction under recursive optimality. For each, we can define a similar condition for hierarchical optimality and show how it implies abstractions that satisfy the equivalence conditions we've defined. These conditions are leaf abstraction (essentially the same as P-equivalence), subroutine irrelevance (features that are totally irrelevant to a subroutine), result-distribution irrelevance (features are irrelevant to the F_{SS} distribution for all policies), and termination (all actions from a state lead to an exit state, so Q_c is 0). We can encompass the last three conditions into a strong form of equivalence, defined as follows.

Definition 4 (SSR-equivalence) *An abstraction function z_c is strongly subroutine (SSR) equivalent for an ALisp program $\mathcal{H}$ iff the following conditions hold for all ω and policies π that are consistent with z_c.*

1. *Equivalent states under z_c have equivalent transition probabilities: $\forall_{\omega',a,a',N}$*
   $F_{SS}(\omega',N|\omega,a) = F_{SS}(z_c(\omega',a'),N|z_c(\omega,a),a)$ [3]
2. *Equivalent states have equivalent rewards: $\forall_{\omega',a,a',N}$*
   $r(\omega',N,\omega,a) = r(z_c(\omega',a'),N,z_c(\omega,a),a)$
3. *The variables in z_c are enough to determine the optimal policy: $\forall_a \pi^*(\omega) = \pi^*(z_c(\omega,a))$*

The last condition is the same sort of condition as the last condition of E-equivalence, and is what enables us to maintain hierarchical optimality. Note that SSR-equivalence implies C-equivalence.

[3]We can actually use a weaker condition: Dietterich's (2000) factored condition for subtask irrelevance

The ALispQ learning algorithm

We present a simple model-free state abstracted learning algorithm based on MAXQ (Dietterich 2000) for our three-part value function decomposition. We assume that the user provides the three abstraction functions z_p, z_c, and z_e. We store and update $\hat{Q}_c(z_c(\omega,a),a)$ and $\hat{Q}_e(z_e(\omega,a),a)$ for all $a \in A$, and $\hat{r}(z_p(\omega,a),a)$ for $a \in A_p$. We calculate

$$\hat{Q}(\omega,a) = \hat{Q}_r(\omega,a) + \hat{Q}_c(z_c(\omega),a) + \hat{Q}_e(z_e(\omega,a),a).$$

Note that, as in Dietterich's work, $\hat{Q}_r(\omega,a)$ is recursively calculated as $\hat{r}(z_p(\omega,a),a)$ if $a \in A_p$ for the base case and otherwise as

$$\hat{Q}_r(\omega,a) = \hat{Q}_r(i_a(\omega),a') + \hat{Q}_c(z_c(i_a(\omega)),a'),$$

where $a' = \arg\max_b \hat{Q}(i_a(\omega),b)$. This means that that the user need not specify z_r. We assume that the agent uses a GLIE (Greedy in the Limit with Infinite Exploration) exploration policy.

Imagine the situation where the agent transitions to ω' contained in subroutine h, where the most recently visited choice state in h was ω, where we took action a, and it took N primitive steps to reach ω'. Let ω_c be the last choice state visited (it may or may not be equal to ω, see Figure 2 for an example), let $a' = \arg\max_b \hat{Q}(\omega',b)$, and let r_s be the discounted reward accumulated between ω and ω'. Then, ALispQ learning performs the following updates:

- if $a \in A_p$, $\hat{r}(z_p(\omega,a),a) \leftarrow (1-\alpha)\hat{r}(z_p(\omega,a),a) + \alpha r_s$
- $\hat{Q}_c(z_c(\omega),a) \leftarrow (1-\alpha)\hat{Q}_c(z_c(\omega),a) + \alpha\beta^N\,[\hat{Q}_r(z_c(\omega'),a') + \hat{Q}_c(z_c(\omega'),a')]$
- $\hat{Q}_e(z_e(\omega,a),a) \leftarrow (1-\alpha)\hat{Q}_e(z_e(\omega,a),a) + \alpha\beta^N\hat{Q}_e(z_e(\omega',a'),a')$
- if ω_c is an exit state and $z_c(\omega_c) = \omega_c$ (and let a be the sole action there) then $\hat{Q}_e(z_e(\omega_c,a),a) \leftarrow (1-\alpha)\hat{Q}_e(z_e(\omega_c,a),a) + \arg\max_b \hat{Q}(\omega',b)$ [4]

Theorem 5 (Convergence of ALispQ-learning) *If z_r, z_s, and z_e are R-,SSP-, and E- Equivalent, respectively, then the above learning algorithm will converge (with appropriately decaying learning rates and exploration method) to a hierarchically optimal policy.*

Experiments

Figure 5 shows the performance of five different learning methods on Dietterich's taxi-world problem. The learning rates and Boltzmann exploration constants were tuned for each method. Note that standard Q-learning performs better than "ALisp program w/o SA" – this is because the problem is episodic, and the ALisp program has joint states that

[4]Note that this only updates the Q_e values when the exit state is the distinguished state in the equivalence class. Two algorithmic improvements are possible: using all exits states and thus basing Q_e on an average of the exit states, and modifying the distinguished state so that it is one of the most likely to be visited.

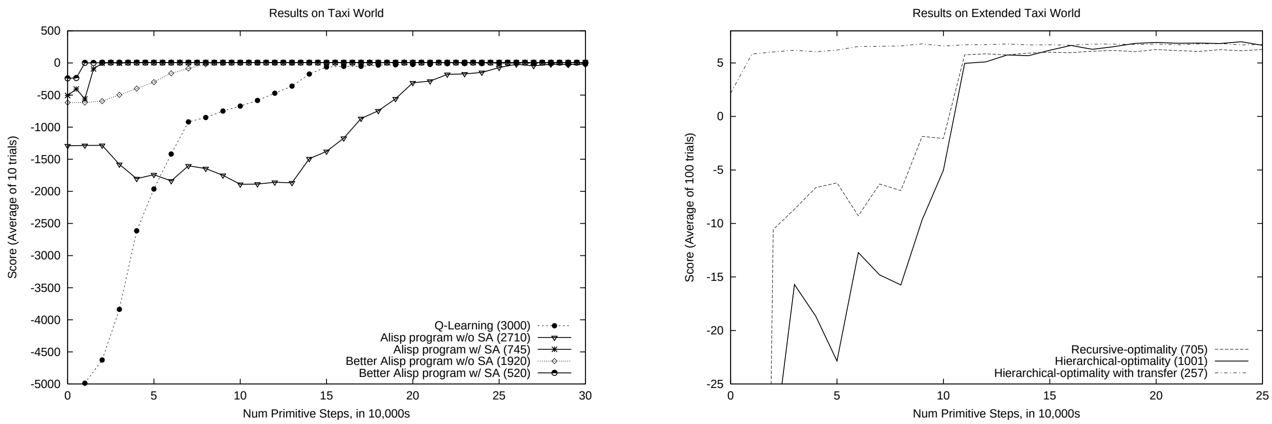


Figure 5: Learning curves for the taxi domain (left) and the extended taxi domain with time and arrival distribution (right), averaged over 50 training runs. Every 10000 primitive steps (x-axis), the greedy policy was evaluated for 10 trials, and the score (y-axis) was averaged. The number of parameters for each method is shown in parentheses after its name.

```
(defun root () (progn (guess) (wait)  (get) (put)))
(defun guess () (choice guess-choice
                        (nav R)
                        (nav G)
                        (nav B)
                        (nav Y)))
(defun wait () (loop while (not (pass-exists)) do
                        (action  'wait)))
```

Figure 4: New subroutines added for the extended taxi domain.

are only visited once per episode, whereas Q learning can visit states multiple times per run. Performing better than Q learning is the "Better ALisp program w/o SA", which is an ALisp program where extra constraints have been expressed, namely that the `load` (`unload`) action should only be applied when the taxi is co-located with the passenger (destination). Second best is the "ALisp program w/ SA" method, and best is the "Better ALisp program w/SA" method. We also tried running our algorithm with recursive optimality on this problem, and found that the performance was essentially unchanged, although the hierarchically optimal methods used 745 parameters, while recursive optimality used 632. The similarity in performance on this problem is due to the fact that every subroutine has a deterministic exit state given the input state.

We also tested our methods on an extension of the taxi problem where the taxi initially doesn't know where the passenger will show up. The taxi must guess which of the primary destinations to go to and wait for the passenger. We also add the concept of time to the world: the passenger will show up at one of the four distinguished destinations with a distribution depending on what time of day it is (morning or afternoon). We modify the root subroutine for the domain and add two new subroutines, as shown in Figure 4.

The right side of Figure 5 shows the results of running our algorithm with hierarchical versus recursive optimality. Because the arrival distribution of the passengers is not known in advance, and the effects of this distribution on reward are delayed until after the guess subroutine finishes, the recursively optimal solution cannot take advantage of the different distributions in the morning and afternoon to choose the best place to wait for the arrival, and thus cannot achieve the optimal score. None of the recursively optimal solutions achieved a policy having value higher than 6.3, whereas every run with hierarchical optimality found a solution with value higher than 6.9.

The reader may wonder if, by rearranging the hierarchy of a recursively optimal program to have "morning" and 'afternoon"`guess` functions, one could avoid the deficiencies of recursive optimality. Although possible for the taxi domain, in general, this method can result in adding an exponential number of subroutines (essentially, one for each possible subroutine policy). Even if enough copies of subroutines are added, with recursive optimality, the programmer must still precisely choose the values for the pseudo-rewards in each subroutine. Hierarchical optimality frees the programmer from this burden.

Figure 5 also shows the results of transferring the `navigate` subroutine from the simple version of the problem to the more complex version. By analyzing the domain, we were able to determine that an optimal policy for `navigate` from the simple taxi domain would have the correct local sub-policy in the new domain, and thus we were able to guarantee that transferring it would be safe.

Discussion and Future Work

This paper has presented ALisp, shown how to achieve safe state abstraction for ALisp programs while maintaining hierarchical optimality, and demonstrated that doing so speeds learning considerably. Although policies for (discrete, finite) fully observable worlds are expressible in principle as lookup tables, we believe that the the expressiveness of a full programming language enables abstraction and modularization that would be difficult or impossible to create otherwise.

There are several directions in which this work can be extended.

- *Partial observability* is probably the most important outstanding issue. The required modifications to the theory are relatively straightforward and have already been investigated, for the case of MAXQ and recursive optimality, by Makar *et al*. (2001).

- *Average-reward learning* over an infinite horizon is more appropriate than discounting for many applications. Ghavamzadeh and Mahadevan (2002) have extended our three-part decomposition approach to the average-reward case and demonstrated excellent results on a real-world manufacturing task.

- *Shaping* (Ng, Harada, & Russell 1999) is an effective technique for adding "pseudorewards" to an RL problem to improve the rate of learning without affecting the final learned policy. There is a natural fit between shaping and hierarchical RL, in that shaping rewards can be added to each subroutine for the completion of subgoals; the structure of the ALisp program provides a natural scaffolding for the insertion of such rewards.

- *Function approximation* is essential for scaling hierarchical RL to very large problems. In this context, function approximation can be applied to any of the three value function components. Although we have not yet demonstrated this, we believe that there will be cases where componentwise approximation is much more natural and accurate. We are currently trying this out on an extended taxi world with continuous dynamics.

- *Automatic derivation of safe state abstractions* should be feasible using the basic idea of backchaining from the utility function through the transition model to identify relevant variables (Boutilier *et al.* 2000). It is straightforward to extend this method to handle the three-part value decomposition.

In this paper, we demonstrated that transferring entire subroutines to a new problem can yield a significant speedup. However, several interesting questions remain. Can subroutines be transferred only partially, as more of a shaping suggestion than a full specification of the subroutine? Can we automate the process of choosing what to transfer and deciding how to integrate it into the partial specification for the new problem? We are presently exploring various methods for partial transfer and investigating using logical derivations based on weak domain knowledge to help automate transfer and the creation of partial specifications.

Acknowledgments

The first author was supported by the generosity of the Fannie and John Hertz Foundation. The work was also supported by the following two grants: ONR MURI N00014-00-1-0637 "Decision Making under Uncertainty", NSF ECS-9873474 "Complex Motor Learning". We would also like to thank Tom Dietterich, Sridhar Mahadevan, Ron Parr, Mohammad Ghavamzadeh, Rich Sutton, Anders Jonsson, Andrew Ng, Mike Jordan, and Jerry Feldman for useful conversations on the work presented herein.

References

[1] Amarel, S. 1968. On representations of problems of reasoning about actions. In Michie, D., ed., *Machine Intelligence 3*, volume 3. Elsevier. 131–171.

[2] Andre, D., and Russell, S. J. 2001. Programmatic reinforcement learning agents. In Leen, T. K.; Dietterich, T. G.; and Tresp, V., eds., *Advances in Neural Information Processing Systems 13*. Cambridge, Massachusetts: MIT Press.

[3] Andre, D., and Russell, S. J. 2002. State abstraction in programmable reinforcement learning. Technical Report UCB//CSD-02-1177, Computer Science Division, University of California at Berkeley.

[4] Bellman, R. E. 1957. *Dynamic Programming*. Princeton, New Jersey: Princeton University Press.

[5] Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*. Austin, Texas: AAAI Press.

[6] Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*. Montreal, Canada: Morgan Kaufmann.

[7] Dietterich, T. G. 2000. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.

[8] Ghavamzadeh, M., and Madadevan, S. 2002. Hierarchically optimal average reward reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*. Sydney, Australia: Morgan Kaufmann.

[9] Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

[10] Makar, R.; Mahadevan, S.; and Ghavamzadeh, M. 2001. Hierarchical multi-agent reinforcement learning. In *Fifth International Conference on Autonomous Agents*.

[11] Ng, A.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*. Bled, Slovenia: Morgan Kaufmann.

[12] Parr, R., and Russell, S. 1998. Reinforcement learning with hierarchies of machines. In Jordan, M. I.; Kearns, M. J.; and Solla, S. A., eds., *Advances in Neural Information Processing Systems 10*. Cambridge, Massachusetts: MIT Press.

[13] Precup, D., and Sutton, R. 1998. Multi-time models for temporally abstract planning. In Kearns, M., ed., *Advances in Neural Information Processing Systems 10*. Cambridge, Massachusetts: MIT Press.