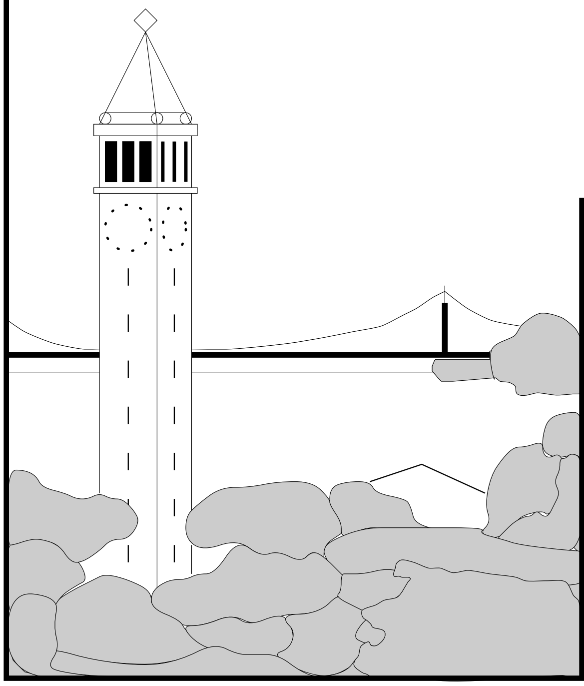
State Abstraction for Programmable Reinforcement Learning Agents

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



Report No. UCB/CSD-1-1156

October 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

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. Like Dietterich's MAXQ framework, this paper develops methods for safe state abstraction in the context of hierarchical reinforcement learning, in which a hierarchical partial program is used to constrain the policies that are considered. We extend techniques from MAXQ to the context of programmable hierarchical abstract machines (PHAMs), which express complex parameterized behaviors using a simple extension of the Lisp language. We show that our methods preserve the property of hierarchical optimality, i.e., optimality among all policies consistent with the PHAM program. We also show how our methods allow safe detachment, encapsulation, and transfer of learned "subroutine" behaviors, and demonstrate our methods on Dietterich's taxi domain.

1 Introduction

The ability to make decisions based on only *relevant* features is a critical part of intelligence and efficient decision making. For example, when faced with a flight of stairs to climb, there is little need to consider the address of the building containing those stairs, let alone the current price of tea in China. *State abstraction* is the process of eliminating aspects of state descriptions to reduce the effective state space; such reductions can speed up dynamic programming and reinforcement learning algorithms considerably. Without state abstraction, each new staircase, each new circumstance involving the existing staircase, and even each step of each staircase requires the agent to relearn a policy 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 [1] for the Missionaries and Cannibals problem. Boutilier *et al.* [5] proposed a general method for deriving safe state abstractions for Markov decision processes (MDPs).

Faster problem solving and learning can also be achieved by providing *prior constraints* on behaviors through some form of partial program. The field of hierarchical reinforcement learning has developed several partial programming formalisms and associated algorithms that construct policies consistent with partial programs. *Hierarchical abstract machines*, or HAMs [11], are hierarchical finite automata with nondeterministic *choice points* within them where learning is to occur. MAXQ programs [7, 8] organize behavior into a hierarchy in which each "subroutine" is simply a repeated choice among a fixed set

* David was supported by the generosity of the Fannie and John Hertz Foundation. The work was also supported by the following grants: NSF ECS-9873474, ONR MURI N00014-00-1-0637

of lower-level subroutines until a termination condition is met. DTGolog [6] allows partial programming in Prolog combined with symbolic dynamic programming as a solution method. Programmable HAMs, or PHAMs [4], are described in Section 2; in short, they augment Lisp with choice points and interrupts to give a very expressive agent language. [1] All these methods construct policies that are "optimal" in some sense. HAMs, PHAMs, and DTGolog use *hierarchical optimality*, i.e., optimality among all policies consistent with the partial program. MAXQ uses *recursive optimality*, in which choices within a subroutine are optimized independently of the calling context, assuming some fixed relative valuation of the possible "exits" from the subroutine.

The combination of state abstraction and hierarchical reinforcement learning is natural, because the notion of "subroutine" is predicated on the idea that decisions "internal" to the subroutine ought to be made based on little or no "outside" information; any relevant outside information can be passed in through arguments. DTGolog [6] derives abstractions that are safe with respect to hierarchical optimality by computing logical descriptions of state sets with constant value. Abstractions in MAXQ preserve recursive optimality, which is a weaker condition and therefore allows stronger abstractions. Additionally, Dietterich [8] makes a crucial observation: a state variable can be irrelevant to a decision in a particular state *even if the variable affects the value of the state*. For example, suppose a taxi driver is on her way to pick up a passenger at location A. The value of the current state depends on the passenger's destination, B, because B influences the fare that will be paid. Yet B has no bearing on the current decision about how to get to A. Dietterich obtains this additional form of abstraction by developing a two-part decomposition of the value function, reflecting the structure of the partial MAXQ program, and shows that it yields substantial additional speedup.

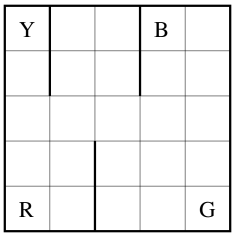
In comparing recursive and hierarchical optimality, Dietterich [8] remarks that "State abstractions [of this kind] cannot be employed without losing hierarchical optimality," which may be true for a two-part value decomposition. Section 3 of this paper develops a three-part decomposition that allows safe state abstraction with respect to hierarchical optimality. We also give a decomposed dynamic programming formulation for deriving hierarchically optimal solutions. Section 4 derives a set of conditions for identifying safe abstractions, and Section 5 describes a convergent reinforcement learning algorithm for PHAMs with state abstraction. Finally, Section 6 describes experimental results for this algorithm using Dietterich's taxi domain. Detailed proofs of all theorems are omitted for space reasons.

2 Background

Our framework for MDPs is standard [8, 9]. An MDP is a 4-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{T}$ is a probabilistic transition function mapping $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and $\mathcal{R}$ is a reward function mapping $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ to the reals. In this paper, we focus on infinite-horizon MDPs with a discount factor β. A solution to an MDP is an optimal policy π^* that maps from $\mathcal{S} \to \mathcal{A}$ and achieves maximum expected discounted reward for the agent. An SMDP (semi-Markov decision process) allows for actions that take more than one time step. $\mathcal{T}$ is modified to be a mapping from $\mathcal{S}, \mathcal{A}, \mathcal{S}, \mathbf{N} \to [0, 1]$, where $\mathbf{N}$ is the natural numbers; i.e., it specifies a distribution over both output states and action durations. $\mathcal{R}$ is then a mapping from $\mathcal{S}, \mathcal{A}, \mathcal{S}, \mathbf{N}$ to the reals. The expected discounted reward for taking an action a in state s and then following a policy π is known as the Q value of that state/action pair, and is defined as $Q^\pi(s, a) = E[r_0 + \beta r_1 + \beta^2 r_2 + ...]$. Note that $\pi = \pi^*$ if $\pi(s) = \arg\max_a Q^\pi(s, a)$.

The PHAM programming language consists of the LISP language augmented with three special macros that enable reinforcement learning to be performed:

[1] Options [12] augment the set of primitive actions with user-written complex behaviors without restricting the possible policies considered; although they speed learning, they are not truly partial policies and are thus not directly comparable to the other methods.




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

Figure 1: The taxi world. It is a 5x5 world with 4 special cells (RGBY) where the passenger is picked up and dropped off. In each episode, the taxi starts in a randomly chosen square, and there is a passenger at a random one of the 4 special cells with random destination. The taxi must pick up the passenger and deliver her, using the commands `N,S,E,W,Pickup,Putdown`. The taxi receives a reward of -1 for every action, +20 for successfully delivering the passenger, -10 for attempting to `pickup` or `putdown` the passenger at incorrect locations. The discount factor is 1.0. The partial program shown on the right is a PHAM for this problem that expresses 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 the navigation component.

- `(choice` <*label*> <*form0*> <*form1*> `...)` takes 2 or more arguments, where the forms are LISP S-expressions. The agent must learn which form to execute.
- `(call-subpham` <*subroutine*> <*arg0*> <*arg1*>`)` calls a subroutine with its arguments and alerts the learning mechanism that a subroutine has been called.
- `(pham-action` <*action-name*>`)` executes a "primitive" action in the MDP.

A PHAM program consists of an arbitrary LISP program that is allowed to use these macros and obeys the constraint that all subroutines that include choice points (either directly, or indirectly, through nested subroutine calls) are called with the *call-subpham* macro. Define C as the set of choice points (one for each `choice` macro) in a PHAM program and Θ as the set of possible machine states achievable by the program (where a machine state includes the program counter, all memory variables, and the call stack). In previous work [2, 4], we showed that, under appropriate restrictions (such as that the number of machine states $|\Theta|$ stays bounded in every run in the environment), the problem of finding the optimal choices for each choice-point c in C is equivalent to finding a solution for the joint SMDP created by executing the PHAM program in an MDP. We presented a hierarchically optimal learning algorithm (based on [11]), and demonstrated the advantages of using the PHAM language in terms of both expressive power and learning rate. An example PHAM is shown in Figure 1 which also describes the Taxi world domain from Dietterich's experiments [8], on which we illustrate our techniques.

3 Value Function Decomposition

A value function decomposition splits the value of a state/action pair into mulitple additive components. In Dietterich's decomposition, for example, the expected discounted return for executing action a and then following π until the end of the current subroutine h is written as $Q^\pi(h, s, a)$ and is split into two parts: $V^\pi(s, a)$ (the expected reward from executing a in world state s) and $C^\pi(h, s, a)$ (the expected reward for finishing subroutine h after a is executed), where $Q^\pi(h, s, a) = V^\pi(s, a) + C^\pi(h, s, a)$. This two-part decomposition allows only for recursive optimality precisely because the expected reward *after* subroutine h is executed is not a component of the value function for h, s, a.

To achieve hierarchical optimality for a value function decomposition in the PHAM framework, we must redefine these two components slightly and add a new component for the expected reward outside the current subroutine. First, note that we will express value functions in the joint SMDP space, where each state ω is comprised of an environment state s and an internal state θ. The actions A consist of the choices at the choice points in the PHAM program. Define A_p as the set of actions where the corresponding choice is either not a sub-PHAM call or calls only subroutines containing no choice points.

The Q-value for executing action a in ω is written $Q^\pi(\omega, a)$ and is decomposed as follows:

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

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 tracjectories starting in w with action a and then following π. Q_r thus expresses the expected discounted reward for doing the current action, Q_c for completing rest of the current subroutine, and Q_e for all the reward external to the current subroutine.

Before presenting the Bellman equations for the decomposed value function, we must first define transition probability measures that take the hierarchy of the program into account. First, we have the standard 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. Two such distributions are useful, $F_{SS(\omega)}^\pi$ and $F_{EX(\omega)}^\pi$, 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. Using these probabilities, we can write the Bellman equations using our decomposed value function, as follows:

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

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

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

where $o(\omega)$ returns the next choice state at the parent level of the hierarchy, and $i_a(\omega)$ returns the first choice state at the child level, given action a. [2]

Theorem 1 *If Q_r^*, Q_c^*, and Q_e^* are solutions to Equation 1, Equation 2, and Equation 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 (omitted here) derived from Equation 1, Equation 2, and Equation 3 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

4 State Abstraction and Transfer

There are several opportunities for state abstraction in the taxi task (Figure 1). For example, while completing the `Get` subroutine, the 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. The key idea of state abstraction is that we want to treat certain sets of states as equivalent for the different components of our decomposition. We first require some notation to set up our theoretical results. Let $z(c)$ be an abstraction function specifying the set of relevant machine and environment features for each choice point c. For the example shown in Table 1 where the x and y locations do

[2] We make a trivial assumption that calls to subroutines are surrounded by choice points with no intervening primitive actions at the calling level. When this isn't the case, single-choice choice points are inserted, which allows simpler analysis. $i_a(\omega)$ and $o(\omega)$ are thus simple deterministic functions, determined from the program structure. See [3] for more details.

θ	x	y	pass	dest	$Q(\omega,a)$	$Q_r(\omega,a)$	$Q_c(\omega,a)$	$Q_e(\omega,a)$
{`get-choice`}	3	3	R	G	0.23	-7.5	-1.0	8.74
{`get-choice`}	3	3	R	B	1.13	-7.5	-1.0	9.63
{`get-choice`}	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 pass)`. Note that {get-choice} is sufficient to describe the stack space for the simple taxi domain, and 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.

not matter for the Q_e value, z(`get-choice`) = { θ, `pass`, `dest`}. Note that this function z groups states together into equivalence classes (for example, all states that agree on assignments to θ, `pass`, and `dest` would be in an equivalence class). Let $\chi_z(\omega)$ be a mapping from states to a canonical member of the equivalence class to which they belong for the abstraction z. For our example, χ_z would map all states $< get{-}choice, *, *, R, G >$ to some state in the class, say, $< get - choice, 0, 0, R, G >$. An abstraction can also be dependent on the action taken at ω, and is written $z(\omega, a)$, where the corresponding mapping function is $\chi_z(\omega, a)$. Finally, define the recursive closure of ω, $rc(\omega)$, as the set of all states contained in any subroutine in the call subtree rooted by the subroutine containing ω. We now define 3 types of equivalence that provide safe state abstractions.

Definition 1 (R-equivalence) *z_r is R-equivalent iff for all ω_1, ω_2, and a,*

$$\chi_{z_r}(\omega_1,a) = \chi_{z_r}(\omega_2,a) \Rightarrow \sum_{\omega',N} p(\omega',N|\omega_1,a)r(\omega',N,\omega_1,a) = \sum_{\omega',N} p(\omega',N|\omega_2,a)r(\omega',N,\omega_2,a)$$

Definition 2 (E-equivalence) *z_e is E-equivalent iff $\forall_{\omega,\pi,a} Q_e^\pi(\omega,a) = Q_e^\pi(\chi_{z_e}(\omega,a),a)$*

Definition 3 (SSP-equivalence) *An abstraction function z_s is strongly sub-PHAM (SSP) equivalent iff the following 4 conditions hold for all PHAM consistent policies:*

1. *If z_s ignores a feature (e.g. f_k) at one level, it must do so at all lower levels in the hierarchy: $\forall_{\omega,k} f_k \notin z_s(\omega) \Rightarrow \forall \omega' \in rc(\omega) f_k \notin z_s(\omega')$.*
2. *Equivalent states have equivalent transition probabilities: $\forall_{\omega,\omega',a,N}$ $p(\omega',N|\omega',a) = p(\chi_{z_s}(\omega'),N|\chi_{z_s}(\omega),a)$ [3]*
3. *Equivalent states have equivalent rewards: $\forall_{\omega,\omega',A,N}$ $r(\omega',N,\omega,a) = r(\chi_{z_s}(\omega'),N,\chi_{z_s}(\omega),a)$*
4. *The variables in z_s are enough to determine the optimal policy: $\forall_\omega \pi(\omega) = \pi(\chi_{z_s}(\omega))$*

The last condition states that in the optimal policy, the choices of action are the same for any states that are abstracted together. This is related to the notion that states can only be abstracted together if the contexts of those states is similar enough that the policy is the same. 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.

Now, we can express the abstracted Bellman equations, using the shorthand of $\chi_{z_r} = \chi_r$, $\chi_{z_s} = \chi_s$, and $\chi_{z_e} = \chi_e$.

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

$$\forall_{a\notin A_p} Q_r^\pi(\chi_s(\omega,a),a) = Q_r^\pi(w',\pi(w')) + Q_c^\pi(w',\pi(w')), \text{ where } w' = i_a(\chi_s(\omega)) \tag{5}$$

$$\forall_a Q_c^\pi(\chi_s(\omega),a) = \sum_{\omega',N} F^\pi_{SS(\chi_s(\omega))}(\omega',N|\chi_s(\omega),a)\beta^N[Q_r^\pi(\chi_s(\omega'),\pi(\chi_s(\omega'))) + Q_c^\pi(\chi_s(\omega'),\pi(\chi_s(\omega')))] \tag{6}$$

$$\forall_a Q_e^\pi(\chi_e(\omega,a),a) = \sum_{\omega',N} F^\pi_{EX(\omega)}(\omega',N|\chi_e(\omega,a),a)\beta^N[Q^\pi(o(\omega'),\pi(o(\omega')))] \tag{7}$$

[3] We can actually use a weaker condition: Dietterich's [8] factored condition for subtask irrelevance; see [3] for details.

Theorem 3 *If z_r is R-equivalent, z_s is SSP-equivalent, and z_e is E-equivalent, then, if Q_r^*, Q_c^*, and Q_e^* are solutions to Equation 4, Equation 5, Equation 6, and Equation 7 for π^*, then $Q^* = Q_r^* + Q_c^* + Q_e^*$ is a solution to the standard Bellman equation.*

Theorem 4 *Decomposed abstracted value iteration and policy iteration algorithms (omitted here) derived from Equation 4, Equation 5, Equation 6, and Equation 7 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

Note that abstraction reduces the size of the system of equations describing the SMDP by an amount dependent on the abstraction functions z_r, z_s, and z_e.

5 The PHAM-SA learning algorithm

We present a simple model-free state abstracted learning algorithm based on MAXQ [8] for our three level value function decomposition. We store and update $\hat{Q}_c(\chi_s(\omega), a)$ and $\hat{Q}_e(\chi_s(\omega, a), a)$ for all $a \in A$, and $\hat{r}(\chi_r(\omega, a), a)$ for those $a \in A_p$. We calculate $\hat{Q}(\omega, a) = \hat{Q}_r(\omega, a) + \hat{Q}_c(\chi_s(\omega), a) + \hat{Q}_e(\chi_e(\omega, a), a)$. Note that as in Dietterich's work, $\hat{Q}_r(\omega, a)$ is recursively calculated as $\hat{r}(\chi_r(\omega, a), a)$ if $a \in A_p$ for the base case and otherwise as $\hat{Q}_r(\omega, a) = \hat{Q}_r(i_a(\omega), a') + \hat{Q}_c(\chi_s(i_a(\omega)), a')$, where $a' = \arg\max_b \hat{Q}(w', b)$.

When transitioning to a state ω' contained in subroutine h, where the last choice point visited in h was ω, where a was executed, and N primitive steps were taken between ω and ω', we do the following updates, where $a' = \arg\max_b \hat{Q}(\omega', b)$.

- if $a \in A_p$, $\hat{r}(\chi_r(\omega, a), a) \leftarrow (1-\alpha)\hat{r}(\chi_r(\omega, a), a) + \alpha r_s$
- $\hat{Q}_c(\chi_s(\omega), a) \leftarrow (1-\alpha)\hat{Q}_c(\chi_s(\omega), a) + \alpha\beta^N \left[\hat{Q}_r(\chi_s(\omega'), a') + \hat{Q}_c(\chi_s(\omega'), a')\right]$
- $\hat{Q}_e(\chi_e(\omega, a), a) \leftarrow (1-\alpha)\hat{Q}_e(\chi_e(\omega, a), a) + \alpha\beta^N \hat{Q}_e(\chi_e(\omega', a'), a')$

Theorem 5 (Convergence of PHAM-SAQ-learning with State Abstraction) *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.*

6 Experiments

Figure 2 shows the performance of five different learning methods on Dietterich's taxi-world problem. The learning rates and Boltzman exploration constants were tuned for each method. The Q-learning method is just regular Q-learning for the problem. Note that it performs better than the PHAM w/o SA (state abstraction) method – this is because the problem is episodic, and the PHAM has states that are only visited once per episode, whereas Q learning can visit states multiple times per run. Performing better than Q learning is the "Better PHAM w/o SA", which is a PHAM where extra constraints have been expressed, namely that the `pickup` (`putdown`) action should only be applied when the taxi is co-located with the passenger (destination). The top performing methods both use state abstraction. The fact that the "Better PHAM w/ SA" performs essentially the same as the "PHAM w/ SA" method is interesting, and appears to be due to the fact that it is relatively easy for the state abstracted PHAM to learn not to `pickup` and `putdown` unless it is at the right place.

7 Conclusions and Future Work

This paper has shown that it is possible to obtain safe state abstraction while maintaining hierarchical optimality. Although it is possible to use state abstraction in an approximate fashion as a form of function approximation [10], we are investigating the possibility of

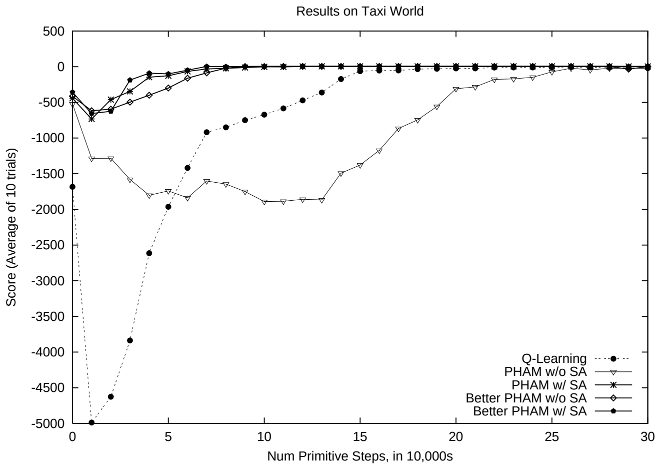


Figure 2: Learning curves for the taxi domain, averaged over 50 training runs. Every 10000 primitive steps (x-axis), the greedy policy was evaluted for 10 trials, and the score (y-axis) was averaged.

starting with safe state abstraction, and then doing function approximation for each component of our three part abstracted and decomposed value function. Additionally, we are investigating the use of shaping and model-based approaches to improve the speed of learning and allow the use of these techniques on real-world domains.

Note that, as in Dietterich's work, our system requires that the user specify the set of state abstractions to use. As previously mentioned, it would be preferable to automatically identify those state abstractions which are warranted by the environment's dynamics. Combining our three part value function decomposition with Boutillier's [6] offline inferential approach to finding state abstractions seems promising.

References

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

[2] D Andre. Programmable hams. tech report: www.cs.berkeley.edu/˜pham.ps, 2000.

[3] D Andre. State abstraction in phams. tech report: www.cs.berkeley.edu/˜sa.ps, 2001.

[4] D. Andre and S.J. Russell. Programmatic reinforcement learning agents. In Dietterich T.G. Tresp V. Leen, T. K., editor, *NIPS 13*. MIT Press, Cambridge, Massachusetts, 2001.

[5] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proc. of the Eleventh National Conf. on Artificial Intelligence*, 1995.

[6] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI-2000*, 2000.

[7] T. G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

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

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

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

[11] R. Parr and S.J. Russell. Reinforcement learning with hierarchies of machines. In M. I. Jordan, M.J. Kearns, and S. A. Solla, editors, *NIPS 10*. MIT Press, Cambridge, Massachusetts, 1998.

[12] R. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, February 1999.