

Concurrent Hierarchical Reinforcement Learning

Bhaskara Marthi, David Latham, Stuart Russell

Dept of Computer Science
UC Berkeley
Berkeley, CA 94720

{bhaskara|latham|russell}@cs.berkeley.edu

Carlos Guestrin

Intel Berkeley Research Lab
Berkeley, CA 94720
guestrin@cs.stanford.edu

Abstract

We describe a language for partially specifying policies in domains consisting of multiple subagents working together to maximize a common reward function. The language extends ALisp with constructs for concurrency and dynamic assignment of subagents to tasks. During learning, the subagents learn a distributed representation of the Q-function for this partial policy. They then coordinate at runtime to find the best joint action at each step. We give examples showing that programs in this language are natural and concise. We also describe online and batch learning algorithms for learning a linear approximation to the Q-function, which make use of the coordination structure of the problem.

Introduction

The field of hierarchical reinforcement learning (Precup & Sutton 1998; Parr & Russell 1997; Dietterich 1998; Andre 2003) attempts to reduce the complexity of solving large Markov decision processes by making use of the temporal structure of good policies. One way of viewing the latter three of these frameworks is that they constrain the set of policies using a partial program. The learning or planning task then becomes that of finding the optimal completion of this program. The partial program is a convenient way for the designer to incorporate prior knowledge about the problem into the solution algorithm. Also, it imposes additive structure on the value function, which in turn allows for state abstraction and faster learning.

However, for some large domains it is difficult to write a concise partial program that expresses one's prior knowledge. Consider, for example, an agent that plays the computer game *Stratagus* (formerly known as *Freecraft*). This game simulates a battle between two opposing armies, each controlled by a player. The armies consist of several units or "subagents", such as peasants, footmen, and knights. Playing the game requires coordinating the subagents to perform various activities like gathering resources, constructing buildings, and attacking the enemy. This domain can be viewed as an MDP with a multidimensional action vector, but a single reward signal.

There is a lot of prior knowledge that we may want to incorporate into our agent using a partial program. For exam-

ple, building a barracks may require first sending some peasants to gather gold and wood, accompanied by some footmen for protection, and having some of the peasants actually build the barracks once enough resources are gathered. We might also have more low-level prior knowledge about actions of individual subagents, for example about navigation. It is not easy to express all this knowledge using a single-threaded program. For example, it is natural to represent each task using a subroutine, but it is then not possible for different subagents to be carrying out different tasks. One might try to have a separate partial program executing for each subagent. However, this makes it hard to specify coordinated behaviour, and the environment of each subagent is no longer Markovian.

In this paper, we present Concurrent ALisp, a language for expressing prior knowledge about such domains using a *multithreaded program* where each task is represented by a thread of execution. Threads can be created and destroyed over time, as new tasks are initiated and terminated. At any point, each task controls some subset of the subagents. Subagents may be reassigned between tasks, and the set of subagents is allowed to change over time. Threads may exchange information using shared memory, but even if they do not, they will coordinate their actions at runtime to maximize the joint Q-function. We first describe the syntax of this language, and then give a formal semantics for it using a semi-Markov decision process. We then describe algorithms for learning and acting in this SMDP. Finally, we show example partial programs and experimental results for a multiagent domain.

Many problems in manufacturing, search-and-rescue, and computer games involve several subagents acting in parallel to satisfy a common goal or maximize a global reward. Work on this problem in deterministic domains includes ConGolog (Giacomo, Lesperance, & Levesque 2000) which builds on situation calculus and (Boutilier & Brafman 1997) which uses STRIPS to represent concurrent actions. The latter makes use of structure in the transition distributions and reward functions, and could possibly be combined with this work, which imposes structure on the policies.

Most work on multiple-agent scenarios in MDPs has focussed only on primitive actions. One exception is (Mahadevan *et al.* 2004), in which each subagent has its own MAXQ hierarchy, and Q-functions at the higher levels of this hier-

archy also depend on the machine states of the other sub-agents.

Background

Our language is an extension of ALisp, and so we begin with a brief overview of ALisp. A complete description can be found in (Andre 2003). We will assume an MDP $M = (S, A, T, R)$ where S is a set of states, A is a set of actions, $T(s, a, s')$ is the transition distribution, and $R(s, a, s')$ is the reward function.

The ALisp programming language consists of standard Lisp (Steele 1990) augmented with three operations:

- **choice** $form_1 \dots form_n$ represents a choice point, where learning takes place.
- **call** $subroutine\ arg_0 \dots arg_n$ calls a subroutine and notifies the learning mechanism that a call has occurred.
- **action** $action-name$ executes a primitive action in the environment.

An ALisp program for M is a Lisp program that may also refer to the state of M and use the above operations. To give this a precise semantics, imagine an agent executing the program as it moves through the environment. At each step it has a joint state $\omega = (s, a)$, where $s \in S$ is the environment state and θ is the *machine state*, consisting of the program counter $\theta.\rho$ (the next statement to execute), the runtime stack state $\theta.s$, and the global memory state $\theta.m$. If ρ is at an **action** statement, the corresponding action is done in the environment and ρ moves to the next statement in the program. If ρ is at a **call** statement, the subroutine is called, θ is updated appropriately, and the call information is noted (for use during learning). If ρ is at a **choice** statement, the agent must choose between the forms to execute. If none of these conditions hold, ρ is updated according to standard Lisp semantics.

Combining an MDP with an ALisp program results in an SMDP over the joint states, and the actions correspond to the choices at choice points of the program. Let a *completion* of an ALisp program in an environment consist of a *choice function* at each choice point, which specifies which choice to take as a function of the current environment state and machine state. We then have

Theorem 1 *The natural correspondence between stationary policies for the SMDP and completions of the Alisp program preserves the value function. In particular, the optimal policy for the SMDP corresponds to the optimal completion of the program.*

Our approach

The language

We assume an MDP as before, where

- Each state s includes a list of subagents that are present in this state
- The set of actions available at s is of the form $A_1 \times \dots \times A_M$ where there is a one-one correspondence between subagents present in this state and components m of the action set.

Our language builds on common Lisp, together with the Allegro implementation of multithreading, which provides standard constructs like locks and waiting. These constructs introduce the possibility of deadlock. We leave it up to the programmer to write programs that always eventually do an action in the environment, and all our results assume this condition.

At any point during execution, each subagent is “assigned” to a unique thread. The operation

(my – subagents)

when executed in a thread returns the list of subagents currently assigned to that thread. There is a single thread at the beginning of execution, and new threads are started using a statement of the form

(spawn $l\ s\ a\ e$)

where l is a label, s is a function¹, a is an argument list, and e is a subset of the subagents currently assigned to the calling thread. This statement asynchronously spawns off a new thread of execution at the given function call, with the subagents in e assigned to it. The label is used by other threads to refer to the newly created thread.

A thread can also reassign some of its subagents to another thread. This is done with the statement

(reassign $e\ label$)

where *label* is the label of the thread to which subagents in e should be reassigned.

The **action** operation needs to be extended to handle multiple subagents. We use the syntax

(action ($e_1\ a_1$) ... ($e_n\ a_n$))

to indicate that subagent e_i should do a_i .

The **choice** statement is as in ALisp. We also include a **choose-from-list** statement for cases when the number of choices is not known in advance.

The presence of multiple simultaneously executing threads means that there is in general no simple hierarchical decomposition of the value function as in ALisp. For this reason, our current implementation of the language does not include a separate **call** statement - function calls can just be done using standard Lisp syntax.

A *concurrent partial program* for an MDP is a program that may use the above constructs, and contains a function **run**, where the root thread begins its execution. If the MDP is one in which the set of subagents may change over time, then the partial program must also provide a function **assign-new-effectors** which decides the thread the newly added effectors are assigned to after an environment step.

Example

We illustrate the language with an example for an extension to Dietterich’s taxi domain (Dietterich 1998) in which the

¹this is easy to implement in Lisp, where functions are first class objects

```

(defun top ()
  (loop
    for i below num-taxis
    do (spawn i #'taxi-top
              (list i) (list i)))
  (loop
    do (wait exists-unserved-passenger)
        (wait exists-idle-taxi)
        (setf i
              (choose-list 'top-choice
                           idle-taxis))
        (setf (aref tasks i)
              (first-unserved-passenger)))
  ))

(defun taxi-top (i)
  (loop
    do (wait (has-task i))
        (serve i (aref tasks i))))

(defun serve (i j)
  (get-pass i j)
  (put-pass i j))

(defun get-pass (i j)
  (nav i (pass-source j))
  (action i 'pickup))

(defun put-pass (i j)
  (nav i (pass-dest j))
  (action i 'putdown))

(defun nav (i loc)
  (loop
    until (equal (taxi-pos i) loc)
    (choice (action (i 'north))
            (action (i 'south))
            (action (i 'west))
            (action (i 'east))
            (action (i 'wait')))))

```

Figure 1: Program for the multiple taxi domain

task is to control multiple taxis moving around in a world with multiple passengers appearing from time to time. The taxis have noisy move actions, and a taxi may also choose to wait for an environment step. A reward is gained whenever a passenger is successfully picked up from their source and delivered to their destination. Also, a negative reward results when two taxis collide.

Our partial program for this domain in Figure 1 is quite similar to programs for the single taxi domain. The main difference is the top level function, which spawns off threads for each taxi, and then chooses which taxi to assign each new passenger to. The individual taxi threads make no mention of other taxis or multithreading constructs. This is intentional - we would like the existence of multiple threads to be as transparent as possible to make it easy to write partial programs. However, we do want the taxis to coordinate their behaviour in the world - for example, the penalty for collisions means that taxis should avoid moving to the same location. In the next section, we will describe the semantics for our language, which makes such coordination happen automatically.

Semantics

The semantics for a concurrent ALisp program is defined in terms of an SMDP it induces on the set of environment-machine states. A machine state θ consists of a global memory state m which includes the contents of all shared variables, a set T of currently active threads, and for each $i \in T$,

- A unique identifier ν ;
- A program counter ρ which is a location in the partial program;
- The set E of subagents currently assigned to this thread;
- The runtime stack σ .

A joint state ω consists of an environment state s and a machine state θ . A *choice state* is a joint state in which each running thread is at a choice or action statement, and at least one thread is at a choice. An *action state* is a joint state in which all running threads are at action statements. Choice statements are the only controllable states of the SMDP, and the set of actions available at a choice state equals the set of joint choices for the threads that are at a choice point. Thus, although the threads may run independently, they must wait for each other at choice points. The reason for this is that we would like decisions to be made together to allow coordination - we believe that making decisions sequentially would lead to Q functions that are more difficult to represent and learn. Also, threads wait for each other at actions. This prevents the overall behaviour from depending on the speed of execution of the different threads.

To define transitions, we first define the semantics of a single thread executing a statement which is not an action or a choice in a joint state. If the statement does not use any of the new constructs, then it has the standard (multi-threaded) Lisp semantics. It is also easy to formalize the descriptions of `spawn`, `die`, and `reassign` from the previous section. However, defining the transition distribution of the SMDP using these components runs into problems,

because it depends on which thread is selected for execution by the scheduler.

We instead define an SMDP over the set of choice states. First, given a particular scheduling algorithm, we define a transition distribution procedurally as follows : after making a joint choice c at a choice state, update the program counters of the threads that made the choice according to c . Then repeatedly do the following :

- If at an action state, perform the joint action in the environment and get the new environment state. Assign any new effectors to threads by calling the partial program's `assign-new-effectors` function. Update the program counters of all the threads.
- Otherwise, use the scheduler to pick a thread that's not at a choice or action, and execute the statement pointed to by this thread's program counter.

until we are at a choice state. Assuming we're guaranteed to eventually terminate or reach a choice state, this gives a well-defined next-state distribution. Similarly, define the SMDP reward function to be the sum of the rewards received at environment actions between the two choice states. A priori, these two functions will depend on the choice of scheduler. However, from now on we restrict attention to partial programs in which these functions are scheduler-independent. This is analogous to requiring that a standard program contain no race conditions. The following theorem justifies the use of this SMDP.

Theorem 2 *The natural correspondence between completions of the ALisp partial program and stationary policies for the SMDP preserves the value function. In particular, the optimal completion corresponds to the optimal policy for the SMDP.*

Linear function approximation

The learning task is to find a policy for the SMDP of a partial program, i.e. a mapping from choice states to joint choices. Unfortunately, this is too large to represent directly, so we represent it implicitly using the Q-function $Q(\omega, a)$. Of course, the Q-function is also too large to represent exactly, but we can use a linear approximation

$$Q(\omega, a) \approx \sum_i w_i \phi_i(\omega, a)$$

Apart from the partial program, the programmer must also specify the features ϕ , and they provide another way to incorporate prior knowledge. The features can depend not just on the environment state but also on the machine state, which can be very useful. For example, a partial program may set a certain flag in its global memory upon completing a particular task, and we can have a feature that depends on this flag.

Algorithms

We first describe how to select the best action in a particular joint state given a learnt Q-function. This requires a maximization over the action set, which is exponentially large in the number of threads. We use the solution described

in (Guestrin, Lagoudakis, & Parr 2002), and assume that the features ϕ_i in the linear approximation to Q each depend on a small number of action variables in any given joint state. We then form the *cost network* (Dechter 1999) representation of $\sum_k Q_k$ and solve the maximization problem using nonserial dynamic programming. This takes time exponential in the treewidth of the cost network. This action selection algorithm is also used in the inner loops of the following learning algorithms.

The learning problem is to learn the Q-function in an SMDP. Thus standard SMDP techniques can be applied. One simple method is to use SMDP Q-learning with linear function approximation. Q-learning uses a set of samples of the form $(\omega, a, r, \omega', N)$, representing a transition from ω to ω' under action a , yielding reward r and taking N units of time. Assuming a discount factor γ and learning rate α , the Q-learning update rule is then

$$w' = w + \alpha(r + \gamma^N \max_{a'} Q(\omega', a'; w) - Q(\omega, a; w))\phi(\omega, a)$$

In practice, the performance of Q-learning is quite sensitive to the parameters in the learning rate and exploration policies, and it is susceptible to divergence. Least-squares policy iteration (Lagoudakis & Parr 2001) is a batch algorithm that tries to address these problems. LSPI was originally designed for MDPs, so we now show how to extend it to SMDPs.

Let Φ be the feature matrix with rows indexed by state-action pairs, so $\Phi(x, a) = (\phi_1(x, a) \dots \phi_K(x, a))^T$. Suppose we are trying to evaluate a policy π , and let $P^\pi(\omega', N|\omega, a)$ be the transition distribution of the associated semi-Markov process. We would like to find weights w such that Φw is a fixed point of the Bellman backup

$$Q(\omega, a) \leftarrow R(\omega, a) + \sum_{\omega', N} P(\omega', N|\omega, a)\gamma^N Q(\omega', \pi(\omega'))$$

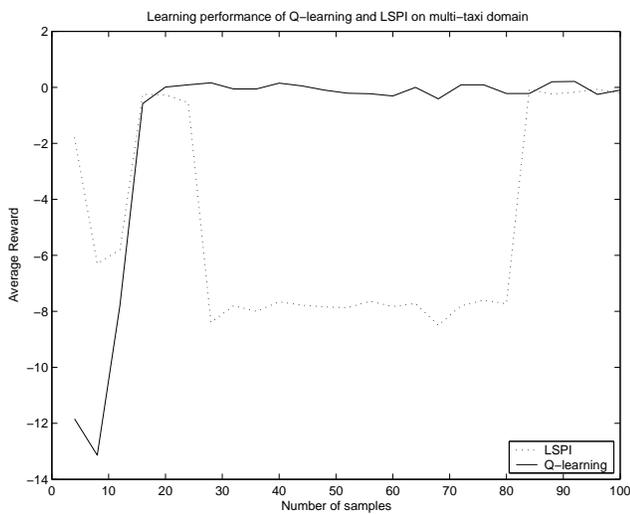
followed by a projection onto the column space of Φ . A little algebra shows that w is a solution of the equation $Aw = b$ where $A = \Phi^T(\Phi - P_\gamma\Phi)$ with

$$P_\gamma(\omega, a, \omega') = \sum_N P(\omega', N|\omega, a)\gamma^N$$

and $b = \Phi^T R$. The matrices Φ and P_γ are too large to handle explicitly, and so as in LSPI, we can form an approximation $\widehat{\Phi}$ from a set of samples $\{(\omega_i, a_i, r_i, \omega'_i, N_i)\}$ by letting the i^{th} row be $\phi(\omega_i, a_i)$. In the MDP case, N is always 1 and so $P_\gamma\Phi(\omega, a)$ is just γ times the expected next-state value of doing a in ω . It can therefore be estimated by the matrix $\widehat{P}\widehat{\Phi}$ whose i^{th} row is $\gamma\phi(\omega'_i, \pi(\omega'_i))$. In the SMDP case, because of the variable discount factor, $P_\gamma\Phi$ can no longer be viewed as an expectation over ω' . However, we can rewrite

$$P_\gamma\Phi(\omega, a) = \sum_{\omega', N} P(\omega', N|\omega, a)\gamma^N \phi(\omega', \pi(\omega'))$$

which is an expectation with respect to the joint distribution $P(\omega', N|\omega, a)$. Thus, given samples $(\omega_i, a_i, r_i, \omega'_i, N_i)$, we may use the unbiased estimate $\widehat{P}\widehat{\Phi}$ whose i^{th} row is $\gamma^{N_i}\phi(\omega_i, a_i)$.



Results

We tested the two learning algorithms on the multiple taxi domain with the partial program described earlier. The linear function approximator included a feature checking for collisions, features that computed the distance of a taxi to its current goal, and a constant feature. Uniform sampling was used to generate the samples for both algorithms. The optimal average reward in this MDP is about 0, so it can be seen from the figure that the two algorithms reach the optimum very fast. Q-learning does better in this situation, probably because the matrix operations in LSPI are numerically unstable with a very small number of samples. Of course, this is a very simple domain. We believe that our approach will scale to much larger problems, and to this end we have implemented an interface to the Stratagus domain described earlier, and are working on applying our algorithms to it.

Conclusions

We have described a language for specifying partial programs in MDPs consisting of multiple cooperating sub-agents. The language is very expressive, including all the functionality of a standard programming language as well as constructs for concurrency, spawning of threads, and dynamic allocation of subagents to tasks. We are currently working on discovering additive decompositions of the type found in (Dietterich 1998; Andre 2003), on improved learning algorithms, and on scaling up our algorithms to handle the full Stratagus domain.

References

- Andre, D. 2003. *Programmable reinforcement learning agents*. Ph.D. Dissertation, UC Berkeley.
- Boutilier, C., and Brafman, R. 1997. Planning with concurrent interacting actions. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 720–729.
- Claus, C., and Boutilier, C. 1998. The dynamics of reinforcement learning in cooperative multiagent systems. In

Proceedings of the Fifteenth National Conference on Artificial Intelligence, 746–752.

Dechter, R. 1999. Bucket elimination : a unifying framework for reasoning. *Artificial Intelligence* 41–85.

Dietterich, T. 1998. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.

Giacomo, G. D.; Lesperance, Y.; and Levesque, H. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 109–169.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing plans to new environments in relational mdps. In *IJCAI 2003*.

Guestrin, C.; Lagoudakis, M.; and Parr, R. 2002. Co-ordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 227–234.

Lagoudakis, M., and Parr, R. 2001. Model free least squares policy iteration. In *Advances in Neural Information Processing Systems*.

Mahadevan, S.; Ghavamzadeh, M.; Rohanimanesh, K.; and Theodorou, G. 2004. Hierarchical approaches to concurrency, multiagency, and partial observability. In Sie, J.; Barto, A.; Powell, W.; and Wunsch, D., eds., *Learning and approximate dynamic programming : scaling up to the real world*. New York: John Wiley.

Parr, R., and Russell, S. 1997. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*.

Precup, D., and Sutton, R. 1998. Multi-time models for temporally abstract planning. In *Advantages in Neural Information Processing Systems 10*.

Steele, G. 1990. *Common Lisp the language*. Digital Press, 2nd edition.