Concurrent Hierarchical Reinforcement Learning

Bhaskara Marthi, Stuart Russell, David Latham
Computer Science Division
University of California
Berkeley, CA 94720
{bhaskara,russell,latham}@cs.berkeley.edu

Carlos Guestrin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
guestrin@cs.cmu.edu

Abstract

We consider applying hierarchical reinforcement learning techniques to problems in which an agent has several effectors to control simultaneously. We argue that the kind of prior knowledge one typically has about such problems is best expressed using a *multithreaded partial program*, and present concurrent ALisp, a language for specifying such partial programs. We describe algorithms for learning and acting with concurrent ALisp that can be efficient even when there are exponentially many joint choices at each decision point. Finally, we show results of applying these methods to a complex computer game domain.

1 Introduction

Hierarchical reinforcement learning (HRL) is an emerging subdiscipline in which reinforcement learning methods are augmented with prior knowledge about the high-level structure of behaviour. Various formalisms for expressing this prior knowledge exist, including HAMs [Parr and Russell, 1997], MAXQ [Dietterich, 2000], options [Precup and Sutton, 1998], and ALisp [Andre and Russell, 2002]. The idea is that the prior knowledge should enormously accelerate the search for good policies. By representing the hierarchical structure of behaviour, these methods may also expose additive structure in the value function [Dietterich, 2000].

All of these HRL formalisms can be viewed as expressing constraints on the behavior of a single agent with a single thread of control; for example, ALisp extends the Lisp language with nondeterministic constructs to create a single-threaded partial programming language (see Section 2). Experience with programming languages in robotics suggests, however, that the behaviour of complex physical agents in real domains is best described in terms of *concurrent* activities. Such agents often have several *effectors*, so that the total action space available to the agent is the Cartesian product of the action spaces for each effector.

Consider, for example, the *Resource domain* shown in Figure 1. This is a subproblem of the much larger Stratagus domain (see stratagus.sourceforge.net). Each peasant in the Resource domain can be viewed as an effector, part of a multibody agent. Controlling all the peasants with a single control thread is difficult. First, different peasants may be involved in different activities and may be at different stages within those activities; thus, the thread has to do complex bookkeeping that essentially implements multiple control stacks. Second, with N peasants, there are 8^N possible joint actions, so action selection is combinatorially nontrivial even when exact value functions are given. Finally, there are interactions among all the peasants' actions—they may collide with each other and may compete for access to the mines and forests.

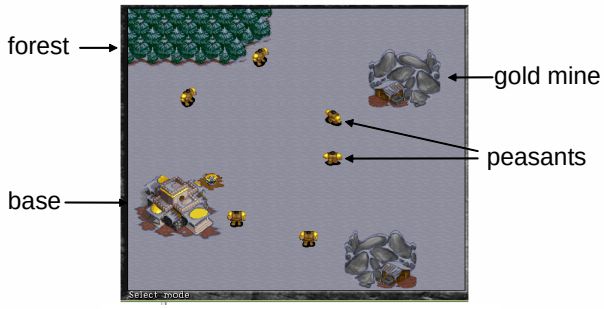

Figure 1: A resource-gathering subgame within Stratagus. Peasants can move one step in the N, S, E, W directions, or remain stationary (all transitions are deterministic). They can pick up wood or gold if they are at a forest or goldmine and drop these resources at the base. A reward of 1 is received whenever a unit of a resource is brought back to the base. A cost of 1 is paid per timestep, and an additional cost of 5 is paid when peasants collide. The game ends when the player's reserves of gold and wood reach a predefined threshold.

To handle all of these issues, we introduce *concurrent ALisp*, a language for HRL in multieffector problems. The language extends ALisp to allow *multithreaded partial programs*, where each thread corresponds to a "task". Threads can be created and destroyed over time as new tasks are initiated and terminated. At each point in time, each effector is controlled by some thread, but this mapping can change over time. Prior knowledge about coordination between threads can be included in the partial program, but even if it is not, the threads will coordinate their choices at runtime to maximize the global utility function.

We begin by briefly describing ALisp (Section 2) and its theoretical foundations in semi-Markov decision processes (SMDPs). Section 3 defines the syntax of concurrent ALisp and its semantics—i.e., how the combination of a concur-

rent ALisp program and a physical environment defines an SMDP. We obtain a result analogous to that for ALisp: the optimal solution to the SMDP is the optimal policy for the original environment that is consistent with the partial program. Section 4 describes the methods used to handle the very large SMDPs that we face: we use linear function approximators combined with relational feature templates [Guestrin *et al.*, 2003], and for action selection, we use coordination graphs [Guestrin *et al.*, 2002]. We also define a suitable SMDP Q-learning method. In Section 5, we show experimentally that a suitable concurrent ALisp program allows the agent to control a large number of peasants in the resource-gathering domain; we also describe a much larger Stratagus subgame in which very complex concurrent, hierarchical activities are learned effectively. Finally, in Section 6, we show how the additive reward decomposition results of Russell and Zimdars [2003] may be used to recover the three-part Q-decomposition results for single-threaded ALisp within the more complex concurrent setting.

2 Background

ALisp [Andre and Russell, 2002] is a language for writing partial programs. It is a superset of common Lisp, and ALisp programs may therefore use all the standard Lisp constructs including loops, arrays, hash tables, etc. ALisp also adds the following new operations :

- `(choose` choices`)` picks one of the forms from the list choices and executes it. If choices is empty, it does nothing.
- `(action` a`)` performs action a in the environment.
- `(call` f args`)` calls subroutine f with argument list args.
- `(get-state)` returns the current environment state.

Figure 2 is a simple ALisp program for the Resource domain of Figure 1 in the single-peasant case. The agent executes this program as it acts in the environment. Let $\omega = (s, \theta)$ be the *joint state*, which consists of the *environment state* s and the *machine state* θ, which in turn consists of the program counter, call stack, and global memory of the partial program. When the partial program reaches a choice state, i.e., a joint state in which the program counter is at a **choose** statement, the agent must pick one of the choices to execute, and the learning task is to find the optimal choice at each choice point as a function of ω. More formally, an ALisp partial program coupled with an environment results in a semi-Markov decision process (a Markov decision process where actions take a random amount of time) over the joint choice states, and finding the optimal policy in this SMDP is equivalent to finding the optimal completion of the partial program in the original MDP [Andre, 2003].

3 Concurrent ALisp

Now consider the Resource domain with more than one peasant and suppose the prior knowledge we want to incorporate is that each individual peasant behaves as in the single-peasant case, i.e., it picks a resource and a location, navigates to that location, gets the resource, returns to the base and drops it off, and repeats the process.

```
(defun single-peasant-top ()
  (loop do
    (choose
      '((call get-gold) (call get-wood)))))

(defun get-wood ()
  (call nav (choose *forests*))
  (action 'get-wood)
  (call nav *home-base-loc*)
  (action 'dropoff))

(defun get-gold ()
  (call nav (choose *goldmines*))
  (action 'get-gold)
  (call nav *home-base-loc*)
  (action 'dropoff))

(defun nav (l)
  (loop until (at-pos l) do
    (action (choose '(N S E W Rest)))))
```

Figure 2: ALisp program for Resource domain with a single peasant.

```
(defun multi-peasant-top ()
  (loop do
    (loop until (my-effectors) do
      (choose '()))
    (setf p (first (my-effectors)))
    (choose
      (spawn p #'get-gold () (list p))
      (spawn p #'get-wood () (list p)))))
```

Figure 3: New top level of a concurrent ALisp program for Resource domain with any number of peasants. The rest of the program is identical to Figure 2.

To incorporate such prior knowledge, we have developed a concurrent extension of ALisp that allows multithreaded partial programs. For the multiple-peasant Resource domain, the concurrent ALisp partial program is identical to the one in Figure 2, except that the top-level function is replaced by the version in Figure 3. The new top level waits for an unassigned peasant and then chooses whether it should get gold or wood and spawns off a thread for the chosen task. The peasant will complete its assigned task, then its thread will die and it will be sent back to the top level for reassignment. The overall program is not much longer than the earlier ALisp program and doesn't mention coordination between peasants. Nevertheless, when executing this partial program, the peasants will automatically coordinate their decisions and the Q-function will refer to joint choices of all the peasants. So, for example, they will learn not to collide with each other while navigating. The joint decisions will be made efficiently despite the exponentially large number of joint choices. We now describe how all this happens.

3.1 Syntax

Concurrent ALisp includes all of standard Lisp. The **choose** and **call** statements have the same syntax as in ALisp.

Threads and effectors are referred to using unique IDs. At any point during execution, there is a set of threads and each thread is assigned some (possibly empty) subset of the effectors. So the **action** statement now looks like (**action** e_1 a_1 ... e_n a_n) which means that each effector e_i must do a_i. In the special case of a thread with exactly one effector assigned to it, the e_i can be omitted. Thus, a legal program for single-threaded ALisp is also legal for concurrent ALisp.

The following operations deal with threads and effectors.

- (**spawn** thread-id fn args effector-list) creates a new thread with the given id which starts by calling fn with arguments args, with the given effectors.
- (**reassign** effector-list thread-id) reassigns the given effectors (which must be currently assigned to the calling thread) to thread-id.
- (**my-effectors**) returns the set of effectors assigned to the calling thread.

Interthread communication is done through condition variables, using the following operations :

- (**wait** cv-name) waits on the given condition variable (which is created if it doesn't exist).
- (**notify** cv-name) wakes up all threads waiting on this condition variable.

Every concurrent ALisp program should contain a designated top level function where execution will begin. In some domains, the set of effectors changes over time. For example, in Stratagus, existing units may be destroyed and new ones trained. A program for such a domain should include a function **assign-effectors** which will be called at the beginning of each timestep to assign new effectors to threads.

3.2 Semantics

We now describe the state space, initial state, and transition function that obtain when running a concurrent ALisp program in an environment. These will be used to construct an SMDP, analogous to the ALisp case; actions in the SMDP will correspond to joint choices in the partial program.

The joint state space is $\Omega = \{\omega = (s, \theta)\}$ where s is an environment state and θ is a machine state. The machine state consists of a global memory state μ, a list Ψ of threads, and for each $\psi \in \Psi$, a unique identifier ι, the set $\mathcal{E}$ of effectors assigned to it, its program counter ρ, and its call stack σ. We say a thread is *holding* if it is at a **choose**, **action**, or **wait** statement. A thread is called a *choice thread* if it is at a **choose** statement. Note that statements can be nested, so for example in the last line of Figure 2, a thread could either be about to execute the **choose**, in which case it is a choice thread, or it could have finished making the choice and be about to execute the **action**, in which case it is not.

In the initial machine state, global memory is empty and there is a single thread starting at the top level function with all the effectors assigned to it.

There are three cases for the transition distribution. In the first case, known as an *internal state*, at least one thread is not holding. We need to pick a nonholding thread to execute next, and assume there is an external *scheduler* that makes this choice. For example, our current implementation is built on Allegro Lisp and uses its scheduler. However, our semantics will be seen below to be independent of the choice of scheduler, so long as the scheduler is fair (i.e., any nonholding thread will eventually be chosen). Let ψ be the thread chosen by the scheduler. If the next statement in ψ does not use any concurrent ALisp operations, its effect is the same as in standard Lisp. The **call** statement does a function call and updates the stack; the **spawn**, **reassign**, **my-effectors**, **wait**, and **notify** operations work as described in Section 3.1. After executing this statement, we increment ψ's program counter and, if we have reached the end of a function call, pop the call stack repeatedly until this is not the case. If the stack becomes empty, the initial function of the thread has terminated, ψ is removed from Ψ, and its effectors are reassigned to the thread that spawned it.

For example, suppose the partial program in Figure 3 is being run with three peasants, with corresponding threads ψ_1, ψ_2, and ψ_3, and let ψ_0 be the initial thread. Consider a situation where ψ_0 is at the dummy **choose** statement, ψ_1 is at the third line of `get-wood`, ψ_2 is at the **call** in the first line of `get-gold`, and ψ_3 is at the last line of `get-gold`. This is an internal state, and the set of nonholding threads is $\{\psi_1, \psi_2\}$. Suppose the scheduler picks ψ_1. The **call** statement will then be executed, the stack appropriately updated, and ψ_1 will now be at the top of the `nav` subroutine.

The second case, known as a *choice state*, is when all threads are holding, and there exists a thread which has effectors assigned to it and is not at an **action** statement. The agent must then pick a joint choice for all the choice threads[1] given the environment and machine state. The program counters of the choice threads are then updated based on this joint choice. The choices made by the agent in this case can be viewed as a *completion* of the partial program. Formally, a completion is a mapping from choice states to joint choices. In the example, suppose ψ_0 is at the dummy **choose** statement as before, ψ_1 and ψ_3 are at the **choose** in `nav`, and ψ_2 is at the `dropoff` action in `get-wood`. This is a choice state with choice threads $\{\psi_0, \psi_1, \psi_3\}$. Suppose the agent has learnt a completion of the partial program which makes it choose $\{(), \texttt{N}, \texttt{Rest}\}$ here. The program counter of ψ_0 will then move to the top of the **loop**, and the program counters of ψ_1 and ψ_3 will move to the **action** statement in `nav`.

The third case, known as an *action state*, is when all threads are holding, and every thread that has effectors assigned to it is at an **action** statement. Thus, a full joint action is determined. This action is performed in the environment and the action threads are stepped forward. If any effectors have been added in the new environment state, the **assign-new-effectors** function is called to decide which threads to assign them to. Continuing where we left off in the example, suppose ψ_0 executes the **my-effectors** statement and gets back to the **choose** statement. We are now at an action state and the joint action $\{\texttt{N}, \texttt{dropoff}, \texttt{Rest}\}$ will be done in the environment. ψ_1 and ψ_3 will return to the `until` in the loop and ψ_2 will die, releasing its effector to `top`.

[1] If there are no choice threads, the program has deadlocked. We leave it up to the programmer to write programs that avoid deadlock.

Let Γ be a partial program and S a scheduler, and consider the following random process. Given a choice state ω and joint choice u, repeatedly step the partial program forward as described above until reaching another choice state ω', and let N be the number of joint actions that happen while doing this. ω' and N are random variables because the environment is stochastic. Let $P_{\Gamma,S}(\omega', N|\omega, u)$ be their joint distribution given ω and u.

We say Γ is *good* if $P_{\Gamma,S}$ is the same for any fair S.[2] In this case, we can define a corresponding SMDP over Ω_c, the set of choice states. The set of "actions" possible in the SMDP at ω is the set of joint choices at ω. The transition distribution is $P_{\Gamma,S}$ for any fair S. The reward function $R(\omega, u)$ is the expected discounted reward received until the next choice state. The next theorem shows that acting in this SMDP is equivalent to following the partial program in the original MDP.

Theorem 1 *Given a good partial program Γ, there is a bijective correspondence between completions of Γ and policies for the SMDP which preserves the value function. In particular, the optimal completion of Γ corresponds to the optimal policy for the SMDP.*

Here are some of the design decisions implicit in our semantics. First, threads correspond to tasks and may have multiple effectors assigned to them. This helps in incorporating prior knowledge about tasks that require multiple effectors to work together. It also allows for "coordination threads" that don't directly control any effectors. Second, threads wait for each other at action statements, and all effectors act simultaneously. This prevents the joint behaviour from depending on the speed of execution of the different threads. Third, choices are made jointly, rather than sequentially with each thread's choice depending on the threads that chose before it. This is based on the intuition that a Q-function for such a joint choice $Q(u_1, \ldots, u_n)$ will be easier to represent and learn than a set of Q-functions $Q_1(u_1), Q_2(u_2|u_1), \ldots, Q_n(u_n|u_1, \ldots u_{n-1})$. However, making joint choices presents computational difficulties, and we will address these in the following section.

4 Implementation

In this section, we describe our function approximation architecture and algorithms for making joint choices and learning the Q-function. The use of linear function approximation turns out to be crucial to the efficiency of the algorithms.

4.1 Linear function approximation

We represent the SMDP policy for a partial program implicitly, using a Q-function, where $Q(\omega, u)$ represents the total expected discounted reward of taking joint choice u in ω and acting optimally thereafter. We use the linear approximation $\hat{Q}(\omega, u; \vec{w}) = \sum_{k=1}^{K} w_k f_k(\omega, u)$, where each f_k is a *feature* that maps (ω, u) pairs to real numbers. In the resource domain, we might have a feature $f_{\text{gold}}(\omega, u)$ that returns the amount of gold reserves in state ω. We might also have a set of features $f_{\text{coll},i,j}(\omega, u)$ which returns 1 if the navigation choices in u will result in a collision between peasants i and j and 0 otherwise.

Now, with N peasants, there will be $O(N^2)$ collision features, each with a corresponding weight to learn. Intuitively, a collision between peasants i and j should have the same effect for any i and j. Guestrin et al. [2003] proposed using a *relational value-function approximation*, in which the weights for all these features are all tied together to have a single value w_{coll}. This is mathematically equivalent to having a single "feature template" which is the sum of the individual collision features, but keeping the features separate exposes more structure in the Q-function, which will be critical to efficient execution as shown in the next section.

4.2 Choice selection

Suppose we have a partial program and set of features, and have somehow found the optimal set of weights $\vec{w}$. We can now run the partial program, and whenever we reach a choice state ω, we need to pick the u maximizing $Q(\omega, u)$. In the multieffector case, this maximization is not straightforward. For example, in the Resource domain, if all the peasants are at navigation choices, there will be 5^N joint choices.

An advantage of using a linear approximation to the Q-function is that this maximization can often be done efficiently. When we reach a choice state ω, we form the *coordination graph* [Guestrin *et al.*, 2002]. This is a graph containing a node for each choosing thread in ω. For every feature f, there is a clique in the graph among the choosing threads that f depends on. The maximizing joint choice can then be found in time exponential in the treewidth of this graph using *cost network dynamic programming* [Dechter, 1999].

In a naive implementation, the treewidth of the coordination graph might be too large. For example, in the Resource domain, there is a collision feature for each pair of peasants, so the treewidth can equal N. However, in a typical situation, most pairs of peasants will be too far apart to have any chance of colliding. To make use of this kind of "context-specificity", we implement a feature template as a function that takes in a joint state ω and returns only the component features that are active in ω—the inactive features are equal to 0 regardless of the value of u. For example, the collision feature template $F_{\text{coll}}(\omega)$ would return one collision feature for each pair of peasants who are sufficiently close to each other to have some chance of colliding on the next step. This significantly reduces the treewidth.

4.3 Learning

Thanks to Theorem 1, learning the optimal completion of a partial program is equivalent to learning the Q-function in an SMDP. We use a Q-learning algorithm, in which we run the partial program in the environment, and when we reach a choice state, we pick a joint choice according to a GLIE exploration policy. We keep track of the accumulated reward and number of environment steps that take place between each pair of choice states. This results in a stream of samples of the form $(\omega, u, r, \omega', N)$. We maintain a running estimate of $\vec{w}$, and upon receiving a sample, we perform the update

$$\vec{w} \leftarrow \vec{w} + \alpha \left(r + \gamma^N \max_{u'} Q(\omega', u'; \vec{w}) - Q(\omega, u; \vec{w}) \right) \vec{f}(\omega, u)$$

[2] This is analogous to the requirement that a standard multithreaded program contain no race conditions.

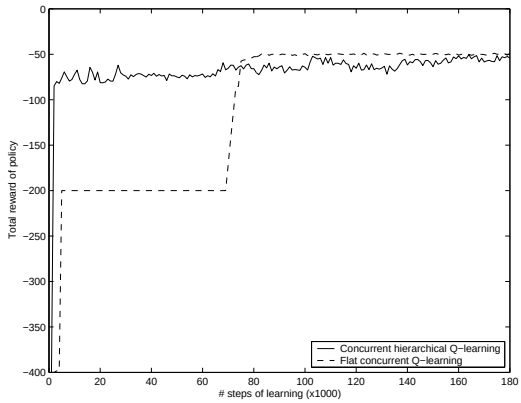

Figure 4: Learning curves for the 3x3 Resource domain with 4 peasants, in which the goal was to collect 20 resources. Curves averaged over 5 learning runs. Policies were evaluated by running until 200 steps or termination. No shaping was used.

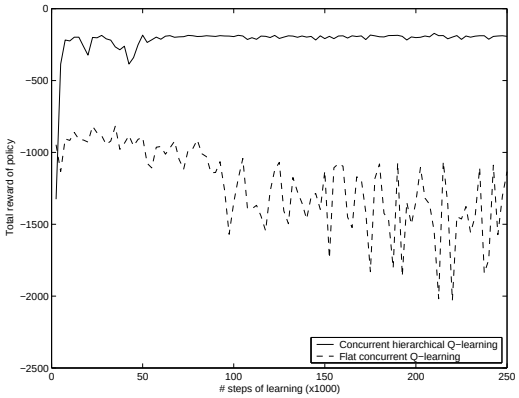

Figure 5: Learning curves for the 15x15 Resource domain with 20 peasants, in which the goal was to collect 100 resources. The shaping function from Section 4.4 was used.

4.4 Shaping

Potential-based shaping [Ng *et al.*, 1999] is a way of modifying the reward function R of an MDP to speed up learning without affecting the final learned policy. Given a *potential function* Φ on the state space, we use the new reward function $\tilde{R}(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)$. As pointed out in [Andre and Russell, 2002], shaping extends naturally to hierarchical RL, and the potential function can now depend on the machine state of the partial program as well. In the Resource domain, for example, we could let $\Phi(\omega)$ be the sum of the distances from each peasant to his current navigation goal together with a term depending on how much gold and wood has already been gathered.

5 Experiments

We now describe learning experiments, first with the Resource domain, then with a more complex strategic domain. Full details of these domains and the partial programs and function approximation architectures used will be presented in a forthcoming technical report.

5.1 Running example

We begin with a 4-peasant, 3x3 Resource domain. Though this world seems quite small, it has over 10^6 states and 8^4 joint actions in each state, so tabular learning is infeasible. We compared flat coordinated Q-learning [Guestrin *et al.*, 2002] and the hierarchical Q-learning algorithm of Section 4.3. Figure 4 shows the learning curves. Within the first 100 steps, hierarchical learning usually reaches a "reasonable" policy (one that moves peasants towards their current goal while avoiding collisions). The remaining gains are due to improved allocation of peasants to mines and forests to minimize congestion. A "human expert" in this domain had an average total reward around -50, so we believe the learnt hierarchical policy is near-optimal, despite the constraints in the partial program. After 100 steps, flat learning usually learns to avoid collisions, but still has not learnt that picking up resources is a good idea. After about 75000 steps, it makes this discovery and reaches a near-optimal policy.

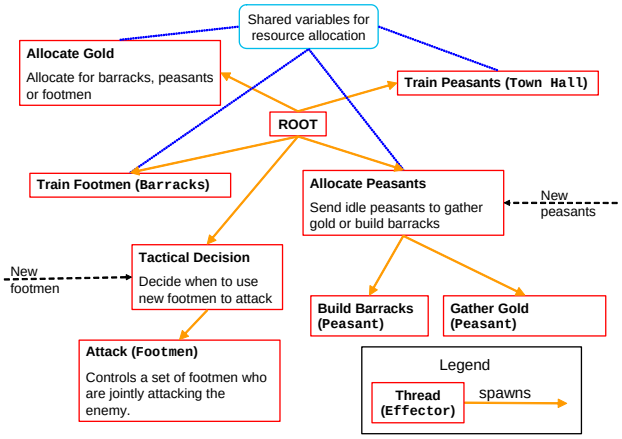

Figure 6: Structure of the partial program for the strategic domain

Our next test was on a 15x15 world with 20 peasants, with learning curves shown in Figure 5. Because of the large map size, we used the shaping function from Section 4.4 in the hierarchical learning. In flat learning, this shaping function cannot be used because the destination of the peasant is not part of the environment state. To write such a shaping function, the programmer would have to figure out in advance which destination each peasant should be allocated to as a function of the environment state. For similar reasons, it is difficult to find a good set of features for function approximation in the flat case, and we were not able to get flat learning to learn any kind of reasonable policy. Hierarchical learning learnt a reasonable policy in the sense defined earlier. The final learnt allocation of peasants to resource areas is not yet optimal as its average reward was about -200 whereas the human expert was able to get a total reward of -140. We are working to implement the techniques of Section 6, which we believe will further speed convergence in situations with a large number of effectors.

5.2 Larger Stratagus domain

We next considered a larger Stratagus domain, in which the agent starts off with a single town hall, and must defeat a single powerful enemy as quickly as possible. An experienced human would first use the townhall to train a few peasants, and as the peasants are built, use them to gather resources. After enough resources are collected, she would then assign a peasant to construct barracks (and reassign the peasant to resource-gathering after the barracks are completed). The barracks can then be used to train footmen. Footmen are not very strong, so multiple footmen will be needed to defeat the enemy. Furthermore, the dynamics of the game are such that it is usually advantageous to attack with groups of footmen rather than send each footman to attack the enemy as soon as he is trained. The above activities happen in parallel. For example, a group of footmen might be attacking the enemy while a peasant is in the process of gathering resources to train more footmen.

Our partial program is based on the informal description above. Figure 6 shows the thread structure of the partial program. The choices to be made are about how to allocate resources, e.g., whether to train a peasant next or build a barracks, and when to launch the next attack. The initial policy performed quite poorly—it built only one peasant, which means it took a long time to accumulate resources, and sent footmen to attack as soon as they were trained. The final learned policy, on the other hand, built several peasants to ensure a constant stream of resources, sent footmen in groups so they did more damage, and pipelined all the construction, training, and fighting activities in intricate and balanced ways. We have put videos of the policies at various stages of learning on the web.[3]

6 Concurrent learning

Our results so far have focused on the use of concurrent partial programs to provide a concise description of multieffector behaviour, and on associated mechanisms for efficient action selection. The learning mechanism, however, is not yet concurrent—learning does not take place at the level of individual threads. Furthermore, the Q-function representation does not take advantage of the *temporal Q-decomposition* introduced by Dietterich [2000] for MAXQ and extended by Andre and Russell [2002] for ALisp. Temporal Q-decomposition divides the complete sequence of rewards that defines a Q-value into subsequences, each of which is associated with actions taken within a given subroutine. This is very important for efficient learning because, typically, the sum of rewards associated with a particular subroutine usually depends on only a small subset of state variables—those that are specifically relevant to decisions within the subroutine. For example, the sum of rewards (step costs) obtained while a lone peasant navigates to a particular mine depends only on the peasant's location, and not on the amount of gold and wood at the base.

We can see immediately why this idea cannot be applied directly to concurrent HRL methods: with multiple peasants, there is no natural way to cut up the reward sequence, because rewards are not associated formally with particular peasants and because at any point in time different peasants may be at different stages of different activities. It is as if, every time some gold is delivered to base, all the peasants jump for joy and feel smug—even those who are wandering aimlessly towards a depleted forest square.

We sketch the solution to this problem in the resource domain example. Implementing it in the general case is ongoing work. The idea is to make use of the kind of *functional reward decomposition* proposed by Russell and Zimdars [2003], in which the global reward function is written as $R(s,a) = R_1(s,a) + \ldots + R_N(s,a)$. Each sub-reward function R_j is associated with a different functional sub-agent—for example, in the Resource domain, R_j might reflect deliveries of gold and wood by peasant j, steps taken by j, and collisions involving peasant j. Since there is one thread per peasant, we can write $R(s,a) = \sum_{\psi \in \Psi} R_\psi(s,a)$ where Ψ is a fixed set of threads. This gives a thread-based decomposition of the Q-function $Q(\omega, u) = \sum_{\psi \in \Psi} Q_\psi(\omega, u)$. That is, Q_ψ refers to the total discounted reward gained in thread ψ.

Russell and Zimdars [2003] derived a decomposed SARSA algorithm for additively decomposed Q-functions of this form that converges to globally optimal behavior. The same algorithm can be applied to concurrent ALisp: each thread receives its own reward stream and learns its component of the Q-function, and global decisions are taken so as to maximize the sum of the component Q-values. Having localized rewards to threads, it becomes possible to restore the temporal Q-decomposition used in MAXQ and ALisp. Thus, we obtain fully concurrent reinforcement learning within the concurrent HRL framework, with both functional and temporal decomposition of the Q-function representation.

7 Related work

There are several HRL formalisms as mentioned above, but most are single-threaded. The closest related work is by Mahadevan's group [Makar *et al.*, 2001], who have extended the MAXQ approach to concurrent activities (for many of the same reasons we have given). Their approach uses a fixed set of identical single-agent MAXQ programs, rather than a flexible, state-dependent concurrent decomposition of the overall task. However, they do not provide a semantics for the interleaved execution of multiple coordinating threads. Furthermore, the designer must stipulate a fixed level in the MAXQ hierarchy, above which coordination occurs and below which agents act independently and greedily. The intent is to avoid frequent and expensive coordinated decisions for low-level actions that seldom conflict, but the result must be that agents have to choose completely nonoverlapping high-level tasks since they cannot coexist safely at the low level. In our approach, the state-dependent coordination graph (Section 4.2) reduces the cost of coordination and applies it only when needed, regardless of the task level.

The issue of concurrently executing several high-level actions of varying duration is discussed in [Rohanimanesh and Mahadevan, 2003]. They discuss several schemes for when to make decisions. Our semantics is similar to their "continue"

[3] http://www.cs.berkeley.edu/∼bhaskara/ijcai05-videos

scheme, in which a thread that has completed a high-level action picks a new one immediately and the other threads continue executing.

8 Conclusions and Further Work

We have described a concurrent partial programming language for hierarchical reinforcement learning and suggested, by example, that it provides a natural way to specify behavior for multieffector systems. Because of concurrent ALisp's built-in mechanisms for reinforcement learning and optimal, yet distributed, action selection, such specifications need not descend to the level of managing interactions among effectors—these are learned automatically. Our experimental results illustrate the potential for scaling to large numbers of concurrent threads. We showed effective learning in a Stratagus domain that seems to be beyond the scope of single-threaded and nonhierarchical methods; the learned behaviors exhibit an impressive level of coordination and complexity.

To develop a better understanding of how temporal and functional hierarchies interact to yield concise distributed value function representations, we will need to investigate a much wider variety of domains than that considered here. To achieve faster learning, we will need to explore model-based methods that allow for lookahead to improve decision quality and extract much more juice from each experience.

References

[Andre and Russell, 2002] D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI*, 2002.

[Andre, 2003] D. Andre. *Programmable Reinforcement Learning Agents*. PhD thesis, UC Berkeley, 2003.

[Dechter, 1999] R. Dechter. Bucket elimination : a unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[Dietterich, 2000] T. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR*, 13:227–303, 2000.

[Guestrin *et al.*, 2002] C. Guestrin, M. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *ICML*, 2002.

[Guestrin *et al.*, 2003] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI*, 2003.

[Makar *et al.*, 2001] R. Makar, S. Mahadevan, and M. Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *ICRA*, 2001.

[Ng *et al.*, 1999] A. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations : theory and application to reward shaping. In *ICML*, 1999.

[Parr and Russell, 1997] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 9*, 1997.

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

[Rohanimanesh and Mahadevan, 2003] K. Rohanimanesh and S. Mahadevan. Learning to take concurrent actions. In *NIPS*. 2003.

[Russell and Zimdars, 2003] S. Russell and A. Zimdars. Q-decomposition for reinforcement learning agents. In *ICML*, 2003.