Concurrent Hierarchical Reinforcement Learning

Bhaskara Marthi
Department of Computer Science
University of California, Berkeley

One major long-term goal in AI is that of building agents that successfully cope with large, uncertain environments. Hierarchical reinforcement learning (HRL) has recently emerged as a way to scale reinforcement learning techniques up towards this goal by incorporating prior knowledge about the structure of policies. However, existing HRL techniques are not suitable for many domains in which the agent has several "effectors" that may be engaged in different tasks. This paper contains an overview of *concurrent hierarchical reinforcement learning* (CHRL), a method of extending HRL to such multieffector domains. For a more complete description, see (Marthi *et al.* 2005).

To motivate HRL, consider the problem of writing the action of overtaking for an automated vehicle. As system designers, we might know that this action should consist of several steps - turn on the indicator, check that the left lane is free, etc. and we might have further prior knowledge about each of these steps. However, we might not know enough to fully specify a controller; for example, we may not know the optimal turning rate when switching lanes to balance speed with passenger comfort, or, at a higher level, we may not know under which circumstances overtaking is worth the slightly higher fuel cost. HRL allows the designer to specify the prior knowledge that she does have and let the rest be learnt. Existing HRL techniques include Options, MAXQ, HAMs, and ALisp. The latter three can be viewed as *partial programming languages* in which the agent's behavior is constrained using a program that includes nondeterministic choice points. Learning consists of finding the right choice at each choice point as a function of the state. Such a partial program results in a huge reduction in the space of policies to be considered. Also, the hierarchical structure in the partial program may expose additive structure in the value function, which can be split into "local" components, each depending on a small subset of the variables. This dramatically reduces the number of parameters to learn.

HRL brings reinforcement learning many steps closer to solving real-world problems, but there are certain types of domains that the above HRL techniques are not well-suited for. As an example, consider the computer game Stratagus. In this game, a player must control a medieval army consisting of footmen, archers, wizards, and the like, and defeat opposing armies. Along the way, he must use his units to perform tasks such as resource gathering, construction of new units and buildings, and combat operations. Now, humans have plenty of prior knowledge about how to play this game well. However, this prior knowledge is very difficult to express as a partial program in MAXQ or ALisp. The problem is that these languages are inherently *single-threaded*, but playing Stratagus requires carrying out many activities in parallel with the different effectors (units). For example, at a given point in time, a peasant unit might be gathering wood, two footman units might be defending the base, and a group of archer units might be launching an attack on the enemy base. An ALisp program for this domain would essentially have to implement multiple control stacks for the different activities, and much of the hierarchical structure would be lost.

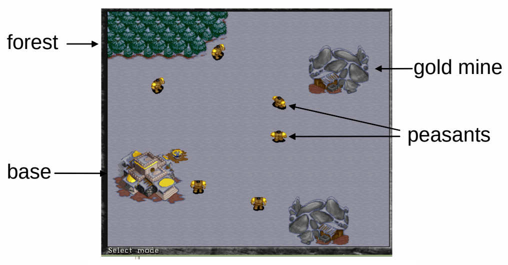

Figure 1: A resource-gathering subgame within Stratagus. Peasants can move one step in the N, S, E, W directions, or remain stationary (all transitions are deterministic). They can pick up wood or gold if they are at a forest or goldmine and drop these resources at the base. A reward of 1 is received 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 domains like Stratagus, we have developed concurrent ALisp, a language for *multithreaded partial programs*, in which each thread corresponds to a "task". Threads are created and destroyed over time as new tasks are

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

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

(defun get-gold ()
  (call nav (choose *goldmine-list*))
  (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

initiated and terminated. At any point, each effector is assigned to some thread, but this assignment may change over time. Prior knowledge about coordination between threads may be included in the partial program, but even if it is not, threads will coordinate their choices at runtime to maximize a joint Q-function. The semantics of a concurrent ALisp program running in an environment can be formalized using a semi-Markov decision process.

Some of the learning algorithms for single-threaded HRL can be extended to CHRL. New algorithmic issues arise in the multi-threaded case, because if multiple threads are making a joint decision, the number of possible choices is exponential in the number of threads. We have adapted the technique of *coordination graphs* to the hierarchical setting. This allows efficient learning and execution in most cases.

As an example, consider the Resource domain in Figure 1. Our prior knowledge for this domain might be that each peasant should pick a resource type, then pick a location from which to gather the resource resource, navigate to that location, gather the resource, and return to the base and drop it off. The partial program in Figure 2 expresses this knowledge. The program has left certain things unspecified. For example, it does not say how to allocate peasants to the different forests and mines, or how to navigate through the map. Our Q-learning algorithm will “fill in” this knowledge based on experience.

We have implemented our algorithms, and tested them out on various subdomains of Stratagus. The algorithms learn successfully on problems that would be too large for conventional RL techniques and the learnt policies demonstrate interesting coordinated behaviors.

One important direction for further work is learning. Our current learning algorithms are not decomposed across threads or subroutines. However, experience in the single-threaded case shows that such decomposition can lead to improved state abstraction and faster learning. We have developed one such decomposed learning algorithm for the case where there is a fixed set of threads and the reward signal decomposes across these threads. We are also work towards scaling up to larger domains. The ultimate goal is to be able to learn to play the entire Stratagus game. We also plan to explore other promising applications of CHRL, such as air traffic management.

References

Marthi, B.; Latham, D.; Russell, S.; and Guestrin, C. 2005. Concurrent hierarchical reinforcement learning. In *IJCAI*. to appear.