Writing Stratagus-playing Agents in Concurrent ALisp

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

Abstract

We describe Concurrent ALisp, a language that allows the augmentation of reinforcement learning algorithms with prior knowledge about the structure of policies, and show by example how it can be used to write agents that learn to play a subdomain of the computer game Stratagus.

1 Introduction

Learning algorithms have great potential applicability to the problem of writing artificial agents for complex computer games [Spronck *et al.*, 2003]. In these algorithms, the agent learns how to act optimally in an environment through experience. Standard "flat" reinforcement-learning techniques learn very slowly in environments the size of modern computer games. The field of hierarchical reinforcement learning [Parr and Russell, 1997; Dietterich, 2000; Precup and Sutton, 1998; Andre and Russell, 2002] attempts to scale RL up to larger environments by incorporating prior knowledge about the structure of good policies into the algorithms.

In this paper we focus on writing agents that play the game Stratagus (stratagus.sourceforge.net). In this game, a player must control a medieval army of units and defeat opposing forces. It has high-dimensional state and action spaces, and successfully playing it requires coordinating multiple complex activities, such as gathering resources, constructing buildings, and defending one's base. We will use the following subgame of Stratagus as a running example to illustrate our approach.

Example 1 *In this example domain, shown in Figure 1, the agent must defeat a single ogre (not visible in the figure). It starts out with a single peasant (more may be trained), and must gather resources in order to train other units. Eventually it must build a barracks, and use it to train footman units. Each footman unit is much weaker than the ogre so multiple footmen will be needed to win. The game dynamics are such that footmen do more damage when attacking as a group, rather than individually. The only evaluation measure is how long it takes to defeat the ogre.*

Despite its small size, writing a program that performs well in this domain is not completely straightforward. It is not immediately obvious, for example, how many

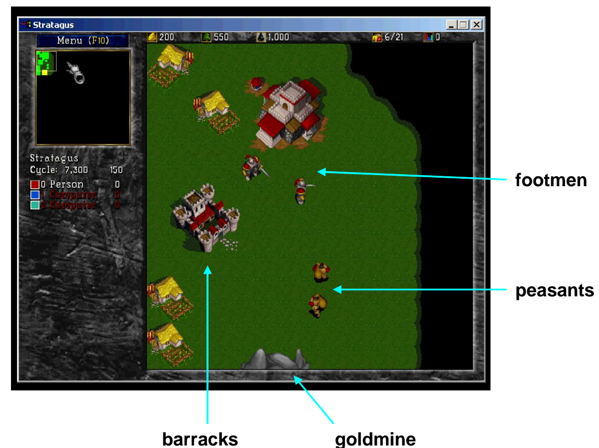

Figure 1: An example subgame of Stratagus.

peasants should be trained, or how many footmen should be trained before attacking the ogre. One way to go about writing an artificial agent that played this program would be to have the program contain free parameters such as **`num-peasants-to-build-given-single-enemy`**, and then figure out the optimal setting of the parameters, either "by hand" or in some automated way. A naive implementation of this approach would quickly become infeasible[1] for larger domains, however, since there would be a large number of parameters, which are coupled, and so exponentially many different joint settings would have to be tried. Also, if the game is stochastic, each parameter setting would require many samples to evaluate reliably.

The field of reinforcement learning [Kaelbling, 1996] addresses the problem of learning to act optimally in sequential decision-making problems and would therefore seem to be applicable to our situation. However, standard "flat" RL algorithms scale poorly to domains the size of Stratagus. One reason for this is that these algorithms work at the level of primitive actions such as "move peasant 3 north 1 step". These algorithms also provide no way to incorporate any prior knowledge one may have about the domain.

[1] A more sophisticated instantiation of this approach, combined with conventional HRL techniques, has been proposed recently [Ghavamzadeh and Mahadevan, 2003].

Hierarchical reinforcement learning (HRL) can be viewed as combining the strengths of the two above approaches, using *partial programs*. A partial program is like a conventional program except that it may contain *choice points*, at which there are multiple possible statements to execute next. The idea is that the human designer will provide a partial program that reflects high-level knowledge about what a good policy should look like, but leaves some decisions unspecified, such as how many peasants to build in the example. The system then learns a *completion* of the partial program that makes these choices in an optimal way in each situation. HRL techniques like MAXQ and ALisp also provide an additive decomposition of the value function of the domain based on the structure of the partial program. Often, each component in this decomposition depends on a small subset of the state variables. This can dramatically reduce the number of parameters to learn.

We found that existing HRL techniques such as ALisp were not directly applicable to Stratagus. This is because an agent playing Stratagus must control several units and buildings, which are engaged in different activities. For example, a peasant may be carrying some gold to the base, a group of footmen may be defending the base while another group attacks enemy units. The choices made in these activities are correlated, so they cannot be solved simply by having a separate ALisp program for each unit. On the other hand, a single ALisp program that controlled all the units would essentially have to implement multiple control stacks to deal with the asynchronously executing activities that the units are engaged in. Also, we would lose the additive decomposition of the value function that was present in the single-threaded case.

We addressed these problems by developing the *Concurrent ALisp* language. The rest of the paper demonstrates by example how this language can be used to write agents for Stratagus domains. A more precise description of the syntax and semantics can be found in [Marthi *et al.*, 2005].

2 Concurrent ALisp

Suppose we have the following prior knowledge about what a good policy for Example 1 should look like. First train some peasants. Then build a barracks using one of the peasants. Once the barracks is complete, start training footmen. Attack the enemy with groups of footmen. At all times, peasants not engaged in any other activity should gather gold.

We will now explain the syntax of concurrent ALisp with reference to a partial program that implements this prior knowledge. Readers not familiar with Lisp should still be able to follow the example. The main thing to keep in mind is that in Lisp syntax, a parenthesized expression of the form `(f arg1 arg2 arg3)` means the application of the function `f` to the given arguments. Parenthesized expressions may also be nested. In our examples, all operations that are not part of standard Lisp are in boldface.

We will refer to the set of buildings and units in a state as the *effectors* in that state. In our implementation, each effector must be given a command at each step (time is discretized into one step per 50 cycles of game time). The command may be a no-op. A concurrent ALisp program can be multi-threaded, and at any point, each effector is assigned to some thread.

```
(defun top ()
  (spawn ''allocate-peasants''
    #'peas-top nil *peas-eff*)
  (spawn ''train-peasants'' #'townhall-top
    *townhall* *townhall-eff*)
  (spawn ''allocate-gold''
  #'alloc-gold nil)
  (spawn ''train-footmen'' #'barracks-top nil)
  (spawn ''tactical-decision''
    #'tactical nil))
```

Figure 2: Top-level function

```
(defun peas-top ()
  (loop
    unless (null (my-effectors))
    do (let ((peas (first (my-effectors))))
         (choose ''peas-choice''
           (spawn (list ''gold'' peas)
             #'gather-gold nil peas)
           (spawn (list ''build'' peas)
             #'build-barracks nil peas)))))
```

Figure 3: Peasant top-level function

Execution begins with a single thread, at the function `top`, shown in Figure 2. In our case, this thread simply creates some other threads using the **`spawn`** operation. For example, the second line of the function creates a new thread with ID "allocate peasants", which begins by calling the function `peas-top` and is assigned effector `*peas-eff*`.

Next, examine the `peas-top` function shown in Figure 3. This function loops until it has at least one peasant assigned to it. This is checked using the **`my-effectors`** operation. It then must make a choice about whether to use this peasant to gather gold or to build the barracks, which is done using the **`choose`** statement. The agent must learn how to make such a choice as a function of the environment and program state. For example, it might be better to gather gold if we have no gold, but better to build the barracks if we have plentiful gold reserves.

Figure 4 shows the `gather-gold` function and the `navigate` function, which it calls. The navigate function navigates to a location by repeatedly choosing a direction to move in, and then performing the move action in the environment using the **`action`** operation. At each step, it checks to see if it has reached its destination using the **`get-env-state`** operation.

We will not give the entire partial program here, but Figure 5 summarizes the threads and their interactions. The `allocate-gold` thread makes decisions about whether the next unit to be trained is a footman or peasant, and then communicates its decision to the `train-footmen` and `train-peasants` threads using shared variables. The

```
(defun gather-gold ()
  (call navigate *gold-loc*)
  (action *get-gold*)
  (call navigate *base-loc*)
  (call *dropoff*))

(defun navigate (loc)
  (loop
    with peas = (first (my-effectors))
    for s = (get-env-state)
    for current = (peas-loc peas s)
    until (equal current loc)
    do (action ''nav''
         (choose *N* *S* *W* *E*))))
```

Figure 4: Gather-gold and navigate functions

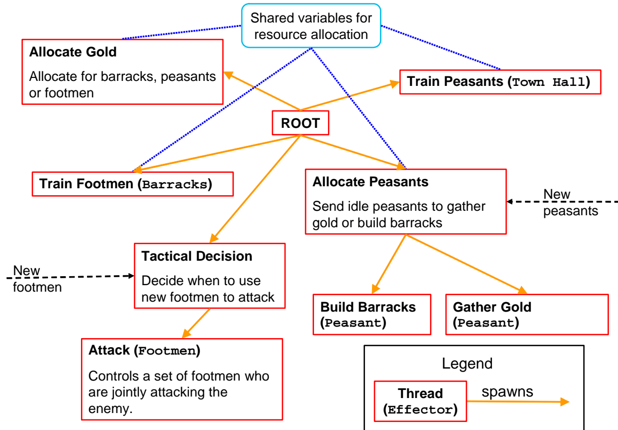

Figure 5: Structure of the partial program for the example domain

`tactical-decision` thread is where new footman units are assigned. At each step it chooses whether to launch an attack or wait. The attack is launched by spawning off a new thread with the footmen in the tactical thread. Currently, it just consists of moving to the enemy and attacking, but more sophisticated tactical manouvering could be incorporated as prior knowledge, or learnt by having choices within this thread.

3 Semantics

We now give an informal semantics of what it means to execute a partial program. We view each state of the environment as having a set of effectors, such that the set of actions allowed at that state is the set of assignments of individual actions to each effector. Thus, we have to make sure that the **`action`** statements in all the threads execute simultaneously. Also, we would like **`choose`** statements to execute simultaneously as much as possible. This is based on the intuition that it is easier to represent and learn the value function for a single joint choice than for a set of sequential choices, each depending on the previous ones. Finally, no time is assumed to elapse in the environment except when the **`action`** statements are being executed, and each joint action takes exactly one time step. Section 6 describes how to fit Stratagus into this framework.

We build on the standard semantics for interleaved execution of multithreaded programs. At each point, there is a set of threads, each having a call stack and a program counter. There is also a set of global shared variables. All this information together is known as the *machine state* θ. We also refer to the *joint state* $\omega = (s, \theta)$ where s is the environment state. A thread is said to be an *action thread* in a given joint state if it is at an **`action`** statement, a *choice thread* if it at a **`choice`** statement, and a *running thread* otherwise.

Given a particular joint state ω, there are three cases for what happens next. If every thread with effectors assigned to it is an action thread, then we are at an *joint action state*. The joint action is done in the environment, and the program counters for the action threads are incremented. If every thread is either an action thread or a choice thread, but we are not at an action state, then we are at a *joint choice state*. The agent must simultaneously make a choice for all the choice threads, and their program counters are updated accordingly. If neither of these two cases holds, then some external scheduling mechanism is used to pick a thread from the running threads whose next statement is then executed.

It can be shown that a partial program together with a Markovian environment yields a *semi-Markov Decision Process* (SMDP), whose state space consists of the joint choice states. The set of "actions" possible at a joint state corresponds to the set of available joint choices, and the reward function of making choice u in ω is the expected reward gained in the environment until the next choice state ω'.

The learning task is then to learn the Q-function of this SMDP, where $Q(\omega, u)$ is the expected total future reward if we make choice u in ω and act optimally thereafter. Once a Q-function is learnt, at runtime the agent simply executes the partial program, and when it reaches a choice state ω, picks the choice u maximizing $Q(\omega, u)$. We will discuss how to do this efficiently in Section 4.

4 Approximating the Q-Function

It is infeasible to represent the function $Q(\omega, u)$ exactly in large domains for two reasons. First, the joint state space is huge, resulting in too many parameters to represent or learn. Second, in situations with many effectors, the set of joint choices, exponential in the number of effectors, will be too large to directly maximize over during execution.

A solution to both these problems is provided by approximating the Q-function as a linear combination of features :

$$Q(\omega, u) = \sum_{k=1}^{K} w_k f_k(\omega, u)$$

We can control the number of parameters to learn by setting K appropriately. Also, if features are chosen to be "local", i.e. to each depend on a small subset of the choice threads, then the maximization can, in many cases, be performed efficiently [Guestrin *et al.*, 2002] using nonsequential dynamic programming. Some example features for the Stratagus subgame are :

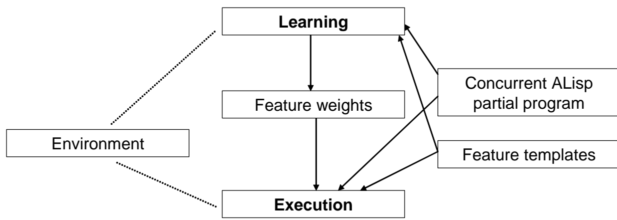

Figure 6: Architecture of the system

1. A feature f_{gold} that counts how much gold we have in state ω
2. A feature f_{attack} that is 1 if there are at least 3 footmen in the tactical thread and u involves choosing to start an attack, and 0 otherwise.
3. A feature $f_{\text{dist, 3}}$ that returns the distance of peasant 3 to his destination after making the navigation choice in u.

Thus, features may depend on the environment state, thread states, or memory state of the partial program. They may also depend on the choices made by any subset of the currently choosing threads.

To further reduce the number of parameters, we make use of relational feature templates [Guestrin *et al.*, 2003]. The feature $f_{\text{dist, 3}}$ above refers to a specific peasant, but it seems reasonable to expect that the weight of this feature should be the same regardless of the identity of the peasant. To achieve this, we allow the specification of a single "feature template" f_{dist} that results in a distance feature for each peasant in a given state, all sharing the same weight w_{dist}.

In our implementation, the set of feature templates must be specified by the user (as Lisp functions). The performance of the learning algorithms can be strongly dependent on the specific features used, and in practice feature engineering takes much more time than writing the partial program.

5 Learning

Figure 6 summarizes the overall system architecture. The user provides a partial program and features to the learning algorithm, which uses experience in the environment to learn a set of weights. The learnt weights can then be used, in conjunction with the partial program and features, to act in the environment.

Since the learning task is to learn the Q-function of an SMDP, we can adapt the standard SMDP Q-learning algorithm to our case. The algorithm assumes a stream of samples of the form (ω, u, r, ω'). These can be generated by executing the partial program in the environment and making joint choices randomly, or according to some exploration policy. After observing a sample, the algorithm performs the online update

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

where α is a learning-rate parameter that decays to 0 over time.

The above algorithm does not make explicit use of the procedural or thread structure of the partial program - learning is centralized and acts on the entire joint state and joint choice. In recent work, we have developed an improved algorithm in which the learning is done separately for each thread. The algorithm also makes use of the procedural structure of the partial program within each thread. To achieve this, the system designer needs to specify in advance a *reward decomposition* function that takes the reward at each timestep and divides it among the threads.

6 Experiments

We implemented the Concurrent ALisp language and the above algorithms on top of standard Lisp. We interfaced with the Stratagus game using a socket. Time was discretized so that every 50 steps of game time (typically about a quarter of a second) corresponds to one "timestep" in the environment. For simplicity, we made the game static, i.e., it pauses at each step and waits for input from the ALisp program, but the algorithms ran fast enough that it should be possible to make the environment dynamic. To fit Stratagus into our framework, in which a joint action at a state must assign an individual action to all the effectors, we added a `noop` action that can be assigned to any unit for which there is no specific command on a given timestep.

We ran the original learning algorithm on the domain from Example 1. Videos of the policies over the course of learning can be found on the web[2]. The initial policy trains no new peasants, and thus collects gold very slowly. It also attacks immediately after each footman is trained. In contrast, the final policy, learnt after about 15000 steps of learning, trains multiple peasants to ensure a constant supply of gold, and attacks with groups of footmen. Thanks to these improvements, the time to defeat the enemy is reduced by about half.

7 Scaling up

It is reasonable to ask how relevant the above results are to the overall problem of writing agents for the full Stratagus game, since the example domain is much smaller. In particular, the full game has many more state variables and effectors, and longer episodes which will require more complex policies. These will increase the complexity of each step of learning and execution, the amount of sampled experience needed to learn a good policy, and the amount of input needed from the human programmer.

We first address the complexity of each step of the algorithms. Since we are using function approximation, the complexity of our algorithms doesn't depend directly on the number of joint states, but only on on the number of features. We believe that it should be possible to find good feature sets that are not so large as to be bottlenecks, and are working to verify this empirically by handling increasingly large subdomains of Stratagus.

The larger number of effectors will typically result in more threads, which will increase the complexity of each joint choice. As discussed in Section 4, a brute-force algorithm for making joint choices would scale exponentially with the number of threads, but our algorithm grows exponentially in

[2] http://www.cs.berkeley.edu/~bhaskara/ijcai05-videos

the tree-width of the coordination graph and only linearly with the number of threads. By making each feature depend only on a small "local" subset of the choosing threads, we can usually make the treewidth small as well. Occasionally, the treewidth will still end up being too large. In this case, we can use an approximate algorithm to find a reasonably good joint choice rather than the best one. Our current implementation selectively removes edges from the coordination graph. Methods based on local search in the space of joint choices are another possibility. Once again, this is unlikely to be the major bottleneck when scaling up.

The cost of executing the partial program itself is currently negligible, but as partial programs become more complex and start performing involved computations (e.g. path-planning), this may change. It should be noted that this issue comes up for any approach to writing controllers for games, whether or not they are based on reinforcement learning. One intriguing possibility in the HRL approach is to treat this as a *meta-level control problem* [Russell and Wefald, 1991] in which the amount of computation is itself decided using a choice statement. For example, an agent may learn that in situations with few units and no immediate combat, it is worth spending time to plan efficient paths, but when there is a large battle going on, it's better to quickly find a path using a crude heuristic and instead spend computation on the joint choices for units in the battle.

The amount of experience needed to learn a good policy is likely to be more of a concern as the domains get increasingly complex and the episodes become longer. The number of samples needed will usually increase at least polynomially with the episode length. This can be mitigated by the use of reward shaping [Ng *et al.*, 1999]. Note also that concurrent ALisp is not wedded to any one particular learning algorithm. For example, we have extended *least-squares policy iteration* [Lagoudakis and Parr, 2001], which aims to make better use of the samples, to our situation. Algorithms that learn a model of the environment along with the Q-function [Moore and Atkeson, 1993] are another promising area for future work.

Finally, the amount of human input needed in writing the partial program, features, reward decomposition, and shaping function will increase in more complex domains. A useful direction to pursue in the medium-term is to learn some of these instead. For example, it should be possible to add a feature selection procedure on top of the current learning algorithm.

8 Conclusion

We have outlined an approach to writing programs that play games like Stratagus using partial programming with concurrent ALisp, and demonstrated its effectiveness on a subdomain that would be difficult for conventional reinforcement learning methods. In the near future, we plan to implement our improved learning algorithm, and scale up to increasingly larger subgames within Stratagus.

References

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

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

[Ghavamzadeh and Mahadevan, 2003] M. Ghavamzadeh and S. Mahadevan. Hierarchical policy-gradient algorithms. In *Proceedings of ICML 2003*, pages 226–233, 2003.

[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.

[Kaelbling, 1996] L. Kaelbling. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 1996.

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

[Marthi *et al.*, 2005] B. Marthi, S. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *Proceedings of IJCAI 2005*, 2005. to appear.

[Moore and Atkeson, 1993] Andrew Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, October 1993.

[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.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the right thing: studies in limited rationality*. MIT Press, Cambridge, MA, USA, 1991.

[Spronck *et al.*, 2003] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Online adaption of game opponent ai in simulation and in practice. In *Proceedings of the Fourth International Conference on Intelligent Games and Simulation*, 2003.