# Learning Hierarchical Behaviors

David Andre

**Computer Science Division,**
**University of California at Berkeley**
**387 Soda Hall, #1776**
**Berkeley, CA 94720-1776**
**dandre@cs.berkeley.edu**
**http://www.cs.berkeley.edu/~dandre**

**Abstract**: In the last few years, many researchers have begun to study how to introduce hierarchy into reinforcement learning methods. Generally, this work has pushed the envelope in one of several important directions, but it typically has avoided the question of how to learn systems of hierarchical behavior from experience. We present a preliminary system for learning simple hierarchical systems using a constrained language for behavior specification. The system learns "do-until" macros by choosing a subgoal from and using its past experience with respect to this subgoal to build a constrained macro behavior that achieves it. The system can either be run offline and used for new but similar problems, or can be used during learning. We present an algorithm that combines macro learning with asynchronous value iteration methods such as prioritized sweeping. Macros learned by the system for a simple environment are discussed. Additionally, we present the related literature and sketch out the future work for this project.

## Introduction

From nearly the beginning of the field of Artificial Intelligence, researchers have made reference to the idea that computers could learn, and then could use what they have learned to learn more. In the last six or seven years, researchers in the field of reinforcement learning have followed this maxim and have been attempting to incorporate hierarchical systems into their work. Typically, this work can be roughly grouped into four categories: (1) state aggregation, (2) explicit subgoal-based hierarchies, (3) explicit MDP decomposition, and (4) behavior/temporal abstraction. Note that of course some techniques fall into multiple categories.

In state aggregation, the idea is to utilize an approximation to the value function by clustering the values of certain states together and treating them as identical or related. The study of function approximation techniques falls in this camp to some degree. In that case, the values of the states are stored and updated using an approximation function (e.g., see Bertsekas & Tsitsiklis, 1996). These techniques can be hierarchical in a fashion, as they represent the modular structure of a state space. Although often these technique use explicit hard coding of the aggregation, many researchers have investigated adaptive aggregation algorithms (e.g. Moore's Parti-Game algorithm). McCallum's (1995) work in this area utilizes similar techniques in partially observable domains and learns decision tree like structures to encode the control policy. Work by Will Uther (1998) further develops McCallum's work and extends it to continuous state spaces.

Many researchers have noticed the similarity between deterministic planning domains and reinforcement learning and have brought in many of the ideas about abstraction developed in that field. Dayan and Hinton's (1993) *feudal* reinforcement learning system specifies an explicit tree like structure of state spaces and controllers and allows a hierarchical decomposition of the problem. Dietterich's (1997) MaxQ algorithm generalizes this idea.

Although somewhat falling into the other three areas, the subfield of MDP decomposition represents another area of quasi-hierarchical structure in reinforcement learning. The idea here that if a problem environment (MDP)

contains regions that are largely separable, the problem can perhaps be attacked with divide and conquer style algorithms, such as those presented by Dean and Lin (1995) and Bertsekas and Tsitsiklis (1996). Some work has been done on automatically determining the decomposition of the state space (Lin, 1993; Parr, 1998; Singh, 1992).

The idea of treating complex (potentially learned) actions as simple ones has long been utilized in AI (e.g. Fikes, Hart, and Nilsson, 1972; Laird, Rosembloom, and Newell, 1986, Benson and Nilsson, 1995). Instead of being limited to the single step actions of a domain, these techniques utilize more complex macro actions that extend in time and state space. Several researchers have attempted to increase the power of their RL systems by incorporating this idea (including Singh 1992; Kaelbling, 1993; Thrun and Schwartz, 1995; Weiring and Schmidhuber, 1997; Takahashi et. al., 1996; Precup and Sutton, 1998; Parr and Russell, 1998; Parr, 1998). Although many of these systems found benefit from hierarchy, in most cases, the various behaviors were either predetermined or pre-learned (i.e. defined by user-supplied subgoals and learned beforehand). In many applications, one would like to have the system determine the appropriate subgoals and learn the hierarchy of behaviors automatically. Although some recent work (Sutton, Precup, and Singh, 1998) addresses the issue of modifying parts of learned macros (called "options" in their work), their method of learning options is somewhat undesirable because it still requires the user to specify explicit subgoals. Additionally, their formulation does not allow for a full hierarchy, only compositions of primitive actions.

This work presents a method for automatically learning hierarchical behaviors given only a list of possible subgoals (or an algorithm for determining these subgoals). By using a constrained view of a macro behavior as a simple do-until loop, our system is able to construct macro behaviors based on experience and an estimate of the utility of a macro based on potential computational savings. In other words, we build concise macros based on experience that are based on our current policy and how many states that are 'abstracted' with the given macro. The macro learning system can be applied offline using learned models of the environment and a learned value function, or it can be applied during learning. The idea is that for complex problems, the agent should be able to learn macros based on experience in the early stages of solving the problem that allow it to solve the overall problem more quickly.

The paper is organized as follows. First, we introduce the MDP notation that will be used in this paper. Then, we attempt to define what makes a good macro. Next, we present our macro language, and discuss how they "work" and relate to previous methods of defining macros. After this, since the paper represents work for a class project, and it is useful in any case to discuss approaches that do not succeed, we briefly describe an alternative approach that failed to work. The next section describes our system for learning macros. We then present several results of applying the macro learner to several simple worlds in the offline case. Then, we discuss the incorporation of the macro learner into model-based reinforcement learning algorithms, focusing on prioritized sweeping. Finally, we conclude with discussion, description of future work, and a brief conclusion.

## Background on Markov Decision Processes and Prioritized Sweeping

We assume the standard *Markov Decision Process* (MDP) framework for reinforcement learning (Kaelbling et. al, 1996). We denote by $S$ the set of possible environment states and by $A$ the set of possible actions. We assume that the system is *Markovian* so that the probability $p(s,a,s')$ of reaching state $s'$ from state $s$ by executing $a$ does not depend on how the system arrived at state $s$. Additionally, we assume that at each state, the agent receives some reward $r(s,a,s')$ that depends only on the current transition $(s,a,s')$. Finally, we assume that the system is fully observable and the agent can determine the current state of the environment and that the agent attempts to maximize its *expected, discounted accumulated reward*.

The problem of reinforcement learning is to build procedures that achieve this objective when the agent does not have initial knowledge of $p$ and $r$. It is well known that in the MDP setting, the agent can choose an optimal policy if it has access to the optimal value function $V^*$. This function measures the optimal expected accumulated reward that can be achieved from state $s$ and satisfies the Bellman equation:

$$V^*(s) = \max_{a \in A} \left[ r(s,a,t) + \sum_{s' \in Succ(s,a)} p(s,a,s') V^*(s') \right]$$

If the agent has access to $V^*(s)$, then the agent can choose optimal actions by using the models of the reward, $r$, and the transition probabilities, $p$. However, we typically assume that the agent must learn the value function from experience – the task of reinforcement learning. (For a review of the field, see Bertsekas and Tsitsiklis, 1996; Kaelbling et. al., 1993). There are two main types of methods, model-based and model-free methods. Model-free

methods, such as Q-learning (Watkins, 1989), will not be considered in detail here. Model based methods utilize the model of the environment to perform value propagation steps (planning) during learning. These techniques essentially work by repeating the following simple loop: take some step in the world, update the reward and transition model, then do some number of value propagations. The algorithms differ in which states undergo a value update after each step in the world. In the classic asynchronous value iteration model (Bertsekas and Tsitsiklis, 1989), all states undergo several value updates (until significant changes cease to take place). In DYNA (Sutton, 1990), a small number of random states undergo updates. In prioritized sweeping (Moore and Atkeson, 1993), states that are likely to undergo large changes in value are chosen to be updated. Peng and William's (1993) present a similar approach for use with the DYNA architecture. Generalized prioritized sweeping (Andre, Friedman, and Parr, 1998) extends the prioritized sweeping model by noticing and utilizing the fact that priorities of the states are ordered based on the gradient of the Bellman equation.

## What makes a macro good?

In order to understand the motivations for the macro scoring metrics we use in this work, it is important to grasp the characteristics of a macro that we are shooting for. Clearly, we want to use hierarchy because it has, so far, proved to be the dominant technique at allowing difficult problems to be tackled. However, exactly what does that mean in the context of reinforcement learning? What types of individual macros will contribute to such an overall hierarchy?

There are many things we would like macros to do for us. First, we would like them, with high probability, to accomplish a subtask useful to the overall task at hand. Ideally, we would also like macros to be useful in more than one setting or context. We want them to abstract away some of the detail of a subproblem so that the overall problem becomes easier through their use. Finally, we argue that it is reasonably important for macros to be more concise than the full value function for the states being covered by the macro. Although some interesting issues of decomposition arise without this constraint, the generalization ability of the macros is somewhat limited. Thus, from these desiderata, we see that there are two dimensions with which macros can be judged. First, macros should not specify behaviors that achieve poor payoff according to the reward structure of the space. Despite the potential elegance or simplicity of a macro to walk along a hallway, bashing one's head against the wall, it will be unlikely to have application if one seeks a high quality solution to a problem. Second, macros should be concise, and should abstract away some of the complexity of the problem. Through the use of macros, the learner should have to make fewer decisions as it follows the policy. As noted by Thrun and Schwartz (1995), these two criteria represent the common tradeoff between expressiveness and conciseness which can approach with a variety of methods, including minimum description length metrics, priors over hypotheses, and simple constraints on the expressiveness of the learned behaviors.

## The macro representation language

Simple iterative actions occur in many different environments. Whether it is as simple as performing some action until a goal is achieved (drive to the store, walk down the hallway) or more complex (write a paper), complex behaviors are often constructed out of repeated executions of a relatively simple behavior. In our macro language, we constrain the system to use simple do-until style macros. Although this is not as powerful as the representation in Parr's (1998) work, they are simple enough to hope that we might be able to learn them. Thus, our macros consist of two parts, the behavior specification itself, and the function that decides when to cease the execution of the

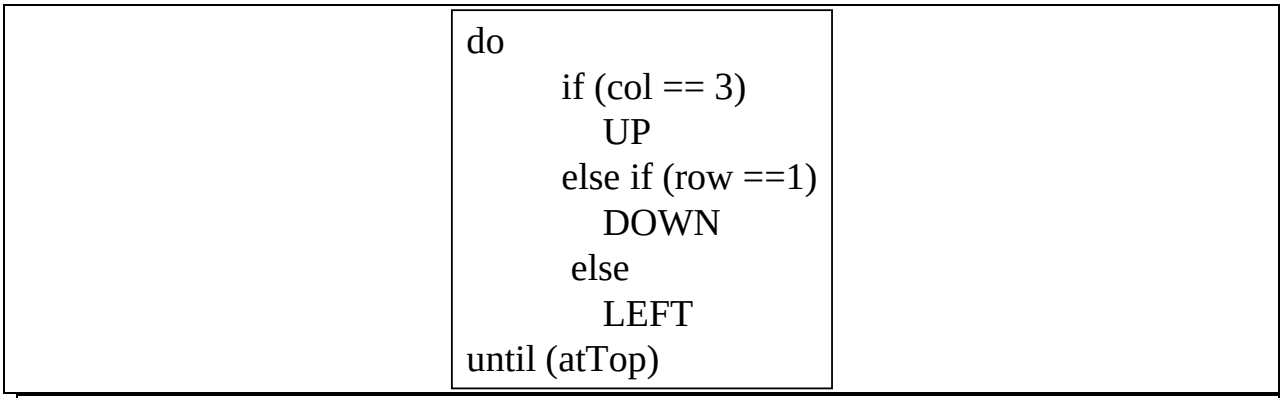


**Figure 1**. a) An example macro, with beahvior (in the do part) and a termination condition (the until part).

macro. An example macro is shown in figure 1.

The behavior specification of a macro consists of a decision list (or, a decision tree), constrained both in the complexity of the decisions and the depth of the list (or tree). The action specified as the leaves of the list (or tree) are actions, which can be either primitive actions or other macros. We assume that the macros are not allowed to be recursive. The termination conditions consist of a simple conjunction of a constrained number of literals. The literals that are used in both the termination conditions and the behavior specification are based on a set of features, *F*, which can be determined from the state *s*. For the current project, we are assuming that the state is fully observable, and thus the features in *F* are guaranteed to be at least sufficient to exactly pick out the state, *s*. Importantly, however, they are not a minimal set of features, and so extra "perceptual" information can be easily added in. This concept is motivated by the fact that many simple behaviors consist of utilizing local perceptual information (such as the end of a hallway or the fact that one can see the store to which one is walking.).

To be precise, some of the features $f \in F$ are percepts, $p \in P, P \subseteq F$. Only those features that are also percepts can be in the termination conditions. Any features can be in the behavior specification. To appear in a macro, the feature is used together with an assignment (i.e., $f = v$, where $v$ is one of the values that feature $f$ can take on. In the future, in addition to the equality operator, we will consider inequality expressions. It is assumed that feature assignments are only allowed to appear once within the macro.

When a macro is executed, it takes control of the agent until it either achieves its goal or a timeout function causes it to stop early. This is necessary because learned macros are not necessarily guaranteed to return, and, following Sutton et al.'s (1998) work on similar macros (what they call options), it might be useful to stop early depending on the value function. Contrary to both the Hierarchies of Abstract Machines (HAMs) in Parr and Russell's (1998) work or the options in Sutton et al.'s (1998) work, the macros defined here do not use explicit internal state inside the macro, and thus policies expressed by these macros might seem to be Markovian. When timeout-based stopping is required, however, the macros become semi-Markov.

## The first attempt: an iterative improvement algorithm

When this project was first begun, the conception of a macro and of the learning algorithm was different from the primary work presented in this paper. In addition to termination conditions, the macros had another conjunction serving as a set of preconditions that specified where the macro could be applied. Precup et al. (1998) also uses this idea of a set of states in which a macro can potentially be applied. The conditions of a macro (precondition and post condition) were applied to determine both if a macro could start and if it should complete. The conditions defined a region of state space within which the macro would act.

To learn these macros, we designed a system that would iteratively improve the conditions, then use those conditions to improve the behavior, and so forth. The method for choosing the behavior was similar to that described in the next section – essentially the idea is to choose a each condition in the decision list to minimize the policy loss over the set of states determined by the macro's current conditions. Determining the conditions, however, was somewhat different. The algorithm would loop over all possible pairs of conjunctions (termination conjunction and precondition conjunction), and score each according to a metric that attempted to describe the characteristics of a good macro. We tried several different metrics for the conditions, including the connectedness of the region covered by the macro, the goodness of the existing behavior over the states defined by the conditions, and combinations of these metrics.

None of the metrics we tried seemed to work very well in offline testing. The macros produced by the system were not terribly useful, and, what's worse, the iterative algorithm would not always converge to a single macro. To attempt to ascertain the problems with the system, we tested it by giving it either the "correct" conditions or the "correct" behavior to begin with. When the system was provided with the correct conditions, it learned a reasonable decision list for the behavior. However, when the system was provided with the appropriate behavior, it still failed to learn the appropriate region. There were several factors that seemed to play a role in the failure of this system, including:

- The tradeoff between generality and the adherence of the policy to the value function was difficult to express and control.
- The learning methods for the two different parts of the macro did not take each other into account well enough. In other words, the condition finder did not take into account how well the behavior learner could learn a behavior for the region specified by the conditions.

- There was no good way to express the utility of the macro in terms of how many decision points were removed from the problem environment.
- By attempting to find regions where a simple behavior was "accurate" with respect to the value function, the system often found very tiny regions or ones with ending states that hardly resembled one another. In other words, by attempting to define the exact region where the macro would be applied, the system was trying to decide the value of the given macro and limit the preconditions such that it would only be applied when it matched the optimal policy. This task (deciding what action to perform at each state) is really the role of the reinforcement learning algorithm, not the macro learner.

## Macro learning

In our second system, we have attempted to address the deficiencies of the first approach. The motivations were to remove the preconditions from the definition of a macro, to utilize explicit subgoals, to use the experience of the agent to guide the selection of macros, and to have a good metric on the possible utility of a macro. The algorithm consists of the following three components.

1. Build a "path" for each percept assignment (subgoal)
2. Choose the best percept assignment according to a simple scoring metric on the path
3. Build a constrained behavior to achieve the chosen subgoal

To start with, we loop over each percept and possible assignment to that percept. For each, the percept assignment specifies a set of states in the world. This step consists of calculating a "path" to these states based on the previous experience of the agent. In other words, from what other states can we achieve this subgoal? More precisely, for a given percept assignment, $p=v$, we use simple dynamic programming to calculate the discounted likelihood, $L_{p=v}(s)$ of reaching any of the subgoal states, using the current exploration policy. The choice of using the exploration policy is not the only one consistent with our design – one could also use Monte Carlo simulation or the previous history of the agent in the environment to estimate the values of $L_{p=v}(s)$. During this process, we also keep track of the minimum number of actions required to get to the subgoal from each state, $d_{p=v}(s)$. Note that by using the exploration policy, we learn paths to the subgoal states that are consistent with the current value function. It is here that we can control the tradeoff between generality and adherence to the value function. By varying the policy from fully adherent (no exploration) to random (total exploration), we can represent a wide range of compromises between the two desirable characteristics of macros. When we do macro learning online, there will be the effect that earlier learned macros will be more "general" and less adherent, because the exploration policy will presumably be less adherent.

After calculating $L_{p=v}(s)$ and $d_{p=v}(s)$ for each percept assignment, the macro learning algorithm scores each of the percept assignments. The score, $y_{p=v}$ for a percept assignment, $p=v$, is shown below.

$$y_{p=v} = \sum_{s:d_{p=v}(s)>1} L_{p=v}(s)$$

This score is the sum, over those states that are at least a minimum of 2 actions away from the goal, of the likelihoods of reaching the subgoal by following the current exploration policy. This approximates the likelihood-weighted number of states at which a decision about what action to take must no longer be taken. The states that are only one action away from the subgoal roughly represents the number of states at which a decision still must be taken to enter the given macro, and the score represents roughly the number of Q-values that are removed from the problem domain by the macro. The actual number is slightly more complicated to calculate, as it has to do with all of the entrance states for the macro, which has to do with the other macros and actions of the agent. One could use such a score if one has the computational resources to calculate it.

For example, consider the two possible subgoals shown in figure 2. The first, atTop=1, has a relatively deep, high probability path, with path lengths as long as 4. The second, deadEnd=1, has a much shallower path. Additionally, the likelihoods for the second subgoal will also be lower according to most exploratory policies, assuming a single goal state, unless the transition probabilities for entering the deadEnd states despite moving UP or DOWN are very high.

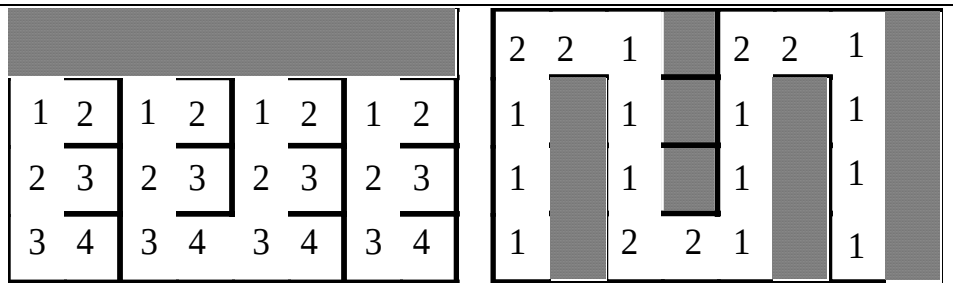


**Figure 2**. The left figure highlights the subgoal states for atTop=1, whereas the figure on the right highlights the subgoal states for deadEnd=1. The numbers represent the values of *d(s)*.

After choosing a percept assignment (subgoal), the algorithm then builds a constrained behavior to achieve the subgoal. This takes place by building a decision list that best matches the value function of the path (computed using the *L(s)* from the path). The first step of this process is to build a set, *R*, of states on the path that have a sufficiently high likelihood of getting to the goal (10% used in our experiments). Then, the algorithm builds each of a set number of decision rules, choosing at each step the rule and action that minimize the policy loss over the states remaining on the path. As each rule is added to the behavior, the states whose action is specified by the rule are removed from *R*. To calculate the policy loss, *Ploss()*, for a given feature assignment, *f=v*, one uses the following equation:

$$Ploss(f = v) = \min_{a \in A} \sum_{s \in R: s_f = v} \left\{ bestval(s) - \sum_{s' \in S: p(s,a,s')>0} p(s,a,s')L(s') \right\},$$

where *bestval*(s) is expressed as

$$bestval(s) = \max_{a \in A} \sum_{s' \in S: p(s,a,s')>0} p(s,a,s')L(s')$$

Figure 3 shows an example 4-state partial environment where the policy loss is calculated for each of the percept assignments and the default. After the set number of decision rules is added, an action to perform on any remaining states in *R* is chosen. This method of building a macro greedily builds a decision list that attempts to get the agent to the subgoal states specified by the percept assignment. When using a decision tree, more information theoretic methods can be utilized to choose on which states to split.

## Results of macro learning

To perform initial tests of the system, we used the offline version of the algorithm. In this version, the system learns macros based on the learned value function and model for a domain. The macros can then be incorporated into the

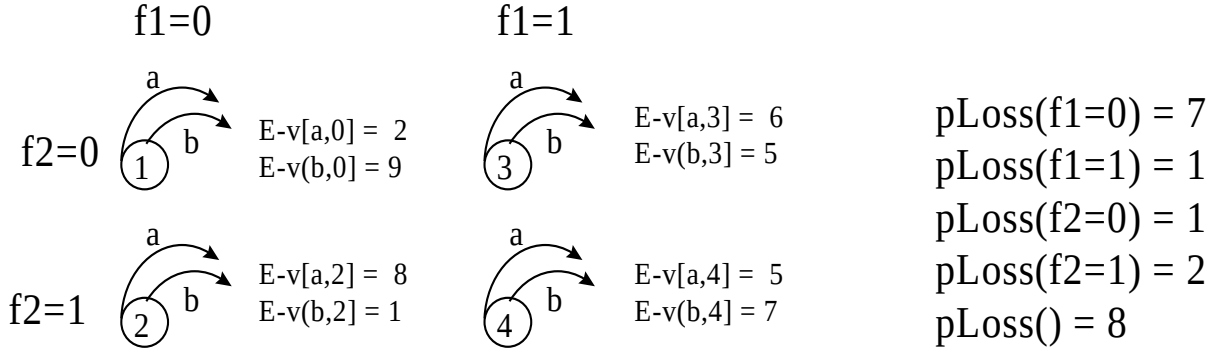


**Figure 3**. A simple 4-state, 2-action example of computing policy loss.

action set for learning on a similar world (with different reward structure). At this stage of the project, we have only run the macro learning algorithm and have examined the macros it produces. The next section describes how to use the macro learning procedure in an online setting.

We first tested the macro learner on the environment shown in figure 2 – a simple 4x8 world with a single start state (the dead-end on the lower right), and a single goal state (the dead-end on the lower left). The feature set was {row, col, atTop, atBottom, deadEnd}, and the percepts were atTop, atBottom, and deadEnd. This meant that the algorithm had to choose a subgoal from the following set: {atTop=0, atTop=1, atBottom=0, atBottom=1, deadEnd=0, and deadEnd=1). The algorithm assigned high scores to only two of the percepts (atTop=1 and atBottom=1); it correctly judged the others to have little value for this world. Due to a small asymmetry in the value function, the score for atTop=1 was slightly better, and the algorithm chose to build a macro to achieve that subgoal. The learned macro is shown below:

```
do{
    if (col=2)
          UP
    else if (row=0)
          UP
    else if (deadEnd)
          LEFT
    else
          UP
} until (atTop=1)
```

There are several things to note about this learned macro. First, because the system was forced to learn decision lists with exactly three rules, this macro is unnecessarily complex. The last else clause and the default rule are sufficient for this particular macro. The second rule is especially problematic, as it removes no states from *R*. This is easy to fix – simply require that any chosen feature eliminate some number of states from *R*. Additionally, one could use an information theoretic stopping criteria instead of the fixed limit used here. Secondly, it is worthwhile to note that the macro has exactly the behavior one would want for a go-to-top macro in this world.

Once the system has learned a single macro, it can learn others by choosing the best percept assignment out of those other than that chosen by the learned macros. In this case the system next learned the go-to-bottom macro. We also tested the system on worlds with similar structure to that shown here (such as a no-key version of the world shown in figure 4), and found that the system tended to create macros for go-to-top, go-to-bottom, go-to-left, and go-to-right quite reliably. Clearly, further testing is required – both of different environments/percepts and of the utility of the macros for later learning.

## Integrating with reinforcement learning

This section describes how to incorporate the macro learning procedure into model-based reinforcement learning algorithms. Both Precup et al. (1998) and Parr and Russell (1998) have utilized similar macro-like entities with reinforcement learning algorithms, and this presentation is based on theirs.

Let us assume that we are attempting to solve a reasonably complex problem, such as the 16K state problem shown in figure 4. The key idea is that the macro learning procedure will be executed only occasionally, and after a reasonably sized portion of the search space has been explored. Note that macros such as go-to-top can be discovered after only an exploration of only a small amount of the state space, and that such macros would probably be quite useful in exploring the rest of the state space.

The basic algorithm is shown below.

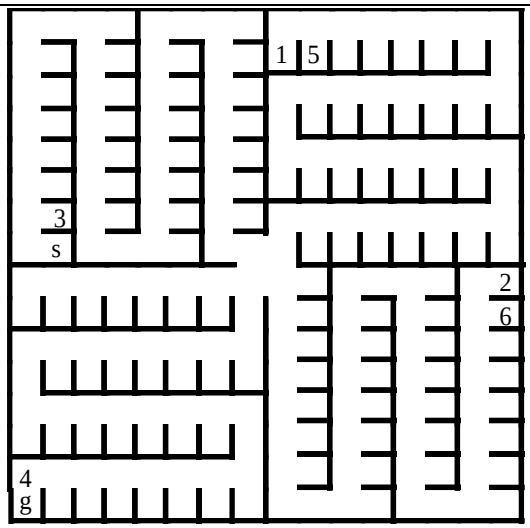


**Figure 4**. A key-world. The agent starts at s, and must collect all of the keys, in order, before being able to receive the reward at g.

```
Until done with learning:
      Take a step in the world (s,a,s').
      Update model of transitions, reward for (s,a,s').
      Do a value update for (s).
      Do some number of other value updates:
            For asynchronous value iteration, update all states
            For DYNA, randomly choose states to update
            For prioritized sweeping, update states based on priority.
      Every N steps, do macro learning.
            Build "path" for each unused percept assignment
            Score each path, and choose best
            Build decision list for chosen percept assignment
```

Several issues arise when attempting to work with macros in model based RL algorithms. First, the algorithm must keep track of transition models for macro actions that include the expected discount for reaching particular successor states. Secondly, when executing a macro, the algorithm must keep track of the current discount and the accumulated discounted reward in order to update the models of discount, transition, and reward when the macro completes. Finally, the system must choose when to apply the macro based on its expected discounted reward – in other words, the macro must be included in the list of actions.

## Future work and discussion

This paper has presented an algorithm for learning hierarchical behaviors. The results presented herein are only preliminary, and current work continues on several fronts. First, we are applying the algorithm to a variety of different worlds using more and different types of percepts. Secondly, we are testing the macros generated by the offline version of the macro learner on similar worlds and attempting to demonstrate that the macros speed learning compared with only using the primitive actions. Also, we are currently implementing the online version of the algorithm, and plan to test that on a variety of environments. It will be interesting to investigate whether the heuristics used here to choose subgoals will work on non-navigational style problems.

This work, although preliminary, has also raised some interesting issues and topics for discussion and for future research.

- There is the question of how to initialize the transition tables for new macros? One suggestion is to initialize the model to take the agent to the nearest state picked out by the subgoal with high probability, and distribute some low probability over nearby states within the subgoal set.
- Macros could potentially be modified as well as created by the macro learner. The system could keep track of how useful the macro is in practice and how often the subgoal states are involved in the high quality paths. If a

set of subgoal states is often involved but the macro is not being utilized, this indicates that a better macro should be learned. The system could then build a new path for the updated policy and then greedily choose a new behavior specification.

- Following Precup et al. (1998), as suggested earlier, we could use a more intelligent quitting criteria for the macros. For example, the probability of quitting could be a function of the relative goodness of the Q values for the macro versus other actions and how many steps have been taken in the macro.
- One potential problem is that unless macros are "perfect" in the sense that they accurately reflect the optimal policy, the final optimal policy will involve no macros. We could introduce explicit computational penalties for having to make a decision, and thus encourage the use of macros. Similarly, we could penalize macros slightly for bailing out, and this information could aid in determining which actions need modification.
- Given a set of learned macros, it would be useful to examine how the macros can be used to abstract the underlying state space. By using some of the techniques presented in Parr's (1998) thesis, one could potentially use global information about the set of choice points induced by the set of learned macros. This might suggest techniques for modifying macros to improve the abstraction.
- Also, we intend to combine this approach with the state of the art methods for exploration (Friedman et al.'s Bayesian exploration, 1998), model representation (Andre et al.'s generalized prioritized sweeping, 1998), model learning (Friedman's SEM algorithm, 1997), and function approximation (various methods presented in Bertsekas and Tsitsiklis, 1996).
- Finally, we intend to attempt to extend this technique to partially observable domains.

## Conclusions

This paper has presented a simple method for learning constrained hierarchical behaviors. By using a do-until representation for macros, the presented system can utilize an agent's experience with the world to choose subgoals that are likely to produce useful actions. The method can be run either offline or online, and allows for a localized tradeoff between policy adherence and generality. The algorithm succeeded at learning reasonable macros for a simple world. Although presenting only preliminary results, it is hoped that this paper is a first step toward harnessing the power of hierarchy for the automatic learning of behaviors.

## Acknowledgements

I would like to thank Sebastian Thrun for his discussing and helping me with the work while I was visiting Carnegie Mellon in the fall. I was also greatly helped by conversations with Illah Nourbakhsh, Jerry Feldman, Stuart Russell, Ron Parr, Nir Friedman, Andrew McCallum, Astro Teller, and Andrew Moore. Also, David is supported by a National Defense Science and Engineering Graduate Fellowship.

## Bibliography

Andre, D., Friedman, N., Parr, R. 1998. Generalized Prioritized Sweeping. . In *Advances in Neural Information Processing Systems.* Vol 10. MIT Press: Cambridge, MA.

Benson, S., and Nilsson, N., 1995. Reacting, planning, and learning in an autonomous agent. In Furukawa, K, Michie, D., and Muggleton, S., editors, *Machine Intelligence 14.* Oxford University Press: Oxford.

Bertsekas, D.C., and Tsitsiklis, J.N. 1996. *Neuro-dynamic programming*. Athena Scientific: Belmont, Mass.

Bertsekas, D.P., and Tsitsiklis, J.N. 1989. *Parallel and distributed computation.* Prentice Hall:Englewood Cliffs.

Dayan, P., and Hinton, G.E. 1993. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 5, MIT Press: Cambridge, MA. 271-278.

Dean, T., and Lin, S.-H. 1995. Decomposition techniques for planning in stochastic domains. *Proceedings of the fourteenth international joint conference on artificial intelligence. Morgan Kaufmann. 1121-1127.*

Dietterich, T.G. 1997. Hierarchical reinforcement learning with maxq value function decomposition. Technical report, Computer Science Dept, Oregon State University.

Fikes, R.E., Hart, P.E., and Nilsson, N.J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251-288.

Friedman, N. 1998. The Bayesian structural EM algorithm. *In Proceedings of the 14th conference on uncertainty in artificial intelligence UAI'98.*

Friedman, N., Dearden, R., and Russell, S.J. 1998. Bayesian Q-Learning. *In Proceedings of the 15th national conference on artificial intelligence AAAI'98.*

Gordon, G.J. 1995. Sequential update of Bayesian network structure. In *Proc. 13th Conf. On Uncertainty in AI*.

Kaelbling, L.P. 1993. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the tenth international conference on machine learning ICML'93*. Morgan Kaufmann:San Mateo, CA. 167-173.

Kaelbling, L.P., Littman, M.L., and Moore, A.W. 1996. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237-285.

Laird, J.E., Rosenbloom, P.S., Newell, A. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1:11-46.

Lin, L.J. 1993. *Reinforcement learning for Robots using neural networks.* PhD Thesis. Computer Science Department, Carnegie Mellon University.

McCallum, A.K. 1995. *Reinforcement Learning with Selective Perception and Hidden State.* PhD Thesis. University of Rochester.

McGovern, A., Sutton, R.S., and Fagg, A.H. 1997. Roles of macro-actions in accelerating reinforcement learning. In *Grace Hopper Celebration of Women in Computing*,13-18.

Moore, A.W. 1994. The part-game algorithm for variable resolution reinforcement learning in muli-dimensional spaces, *Advances in Neural Processing Systems 7:*711-718, MIT Press:Cambridge, MA.

Moore, A.W., and Atkeson, C.G. 1993. Prioritized sweeping – reinforcement learning with less data and less time. *Machine Learning*, 12:103-130.

Nilsson, N.J. 1973. Hierarchical robot planning and execution system. SRI AI Center Technical Note 76, SRI International, Inc., Menlo Park, CA.

Nilsson, N.J. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139-158.

Parr, R. 1998. Hierarchical control and learning for Markov decision processes, PhD Thesis, Computer Science Division, University of California at Berkeley.

Parr, R., and Russell, S.J. 1998. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems.* Vol 10. MIT Press: Cambridge, MA.

Precup, D., and Sutton, R.S., 1998. Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing System.* Vol 10. MIT Press: Cambridge, MA.

Precup, D., Sutton, R. S., Singh, S. 1998. Theoretical results on reinforcement learning with temporally abstract options. *Proceedings of the Tenth European Conference on Machine Learning*, Springer Verlag.

Precup, D., Sutton, R.S., Singh, S.P. 1997. Planning with closed-loop macro actions. *Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems*, pp. 70-76.

Singh, S. 1992. Scaling reinforcement learning by learning variable temporal resolution models. In *Proceedings of the Ninth International Conference on Machine Learning ICML'92*. Morgan Kaufmann:San Mateo, CA. 521-539.

Singh, S.P. 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine learning*, 8(3). May.

Sutton, R. S., Precup, D., Singh, S. 1998 (submitted). Between MDPs and semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. *JAIR*.

Sutton, R. S., Precup, D., Singh, S. 1998. Intra-option learning about temporally abstract actions. *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann.

Sutton, R.S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning*: *Proc. 7th Int. Conf.*

Thrun, T., Schwartz, A. 1995. Finding structure in reinforcement learning. *Advances in Neural Information Processing Systems 7.* San Mateo: Morgan Kaufmann.

Uther, W., and Veloso, M.M. 1997. Generalizing Adversarial Reinforcement Learning. *AAAI Fall symposium on Model directed autonomous systems.*

Watkins, C.J. 1989. *Models of delayed reinforcement learning.* PhD Thesis. Psych. dept, Cambridge University.

Weiring, M., Schmidhuber, J. 1997. HQ-Learning. *Adaptive Behavior.* 6(2).

Y. Takahashi, M. Asada and K. Hosoda. 1996. Reasonable Performance in Less Learning Time by Real Robot Based on Incremental State Space Segmentation. *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems.* 1518-1524.