# Hierarchical Learning in Stochastic Domains: Preliminary Results

**Leslie Pack Kaelbling**
Computer Science Department
Box 1910
Brown University
Providence, RI, USA 02912-1910
lpk@cs.brown.edu

## Abstract

This paper presents the HDG learning algorithm, which uses a hierarchical decomposition of the state space to make learning to achieve goals more efficient with a small penalty in path quality. Special care must be taken when performing hierarchical planning and learning in stochastic domains, because macro-operators cannot be executed ballistically. The HDG algorithm, which is a descendent of Watkins' Q-learning algorithm, is described here and preliminary empirical results are presented.

## 1 INTRODUCTION

Reinforcement learning is a general tool for deriving strategies that optimize a fixed reinforcement function in a stochastic environment. A crucial problem in reinforcement learning is *temporal credit assignment*: how to choose actions based on good results that happen after (perhaps long after) the action is taken. This problem is solved well in the general case by *temporal difference* methods, such as Watkins' Q learning [Barto *et al.*, 1989, Watkins, 1989] and Sutton's TD algorithm [Sutton, 1988].

In much of the work on reinforcement learning, however, researchers have studied a restriction of the problem to cases of "goals of achievement." Rather than having an arbitrary mapping of states of the world to reinforcements for the agent, there is a single "goal" situation at which the agent must arrive as soon as possible. Goals of achievement can be modeled in the general reinforcement-learning framework by having the goal situation generate a positive reinforcement and all other situations a zero reinforcement. Because there is a discounting factor built into the temporal-difference methods, actions that lead to the goal sooner rather than later will be preferred.

Domains with goals of achievement can be learned more efficiently by methods that are tailored to this special case. In previous work [Kaelbling, 1993b], we discussed the *DG learning* algorithm, which is analogous to Q learning, but directly suited to goals of achievement. In the basic single-goal case, DG learning is somewhat more effective than Q learning. The real importance of DG learning is shown when we consider the case of goals of achievement that change over time; the common scenario of a taskable delivery robot is an instance of this sort of problem. A simple extension to DG learning allows a large amount of between-task transfer, making it greatly preferable to Q learning for this sort of problem, without requiring any complex mechanisms.

Unfortunately, the DG learning algorithm is expensive, both in terms of the number of learning instances required to achieve good performance and in computational time and space per instance. In this paper, we consider learning to achieve goals in a two-level hierarchy. The new algorithm, HDG, is more efficient than DG in both time and space; preliminary empirical results show that it learns more quickly initially, but has a somewhat sub-optimal asymptotic performance. As state spaces become huge, we can use multiple levels of hierarchy to achieve great performance improvements that will, in general, offset a degree of sub-optimality in performance.

In the following sections we present background material on Q and DG learning, discuss the nature of our hierarchies, define the HDG learning algorithm, and give some preliminary experimental results. We conclude by comparing this to related work and by considering directions for future work.

## 2 Q AND DG LEARNING

We assume that a learning agent is embedded in an environment in such a way that it can discriminate the set $S$ of distinct world situations and can take the set $A$ of actions on the world. The world is modeled as a Markov process, making stochastic transitions based on its current situation and the actions taken by the agent. We let $T(s, a, s')$ be the probability that the world will make a transition to situation $s'$ given that it was in situation $s$ and the agent executed action $a$. In addition, for each situation $s$ and action $a$, $r(s, a)$ is the *reinforcement value* of taking action $a$ in situation $s$.

In general, this value is a scalar random variable; it must have a stationary distribution, but the same situation-action pair may have different results on different trials.

## 2.1 Q Learning

The general reinforcement-learning problem is typically stated as finding a policy that maximizes expected discounted future reinforcement. A policy $\pi$ is a mapping from $S$ to $A$. The expected discounted future reinforcement of a policy $\pi$ in a situation $s$ is defined as

$$\sum_{t=0}^{\infty} \gamma^t er(t) \ ,$$

where $er(t)$ is the expected value of the reinforcement obtained at step $t$, given that the agent started in situation $s$ and executed policy $\pi$. The variable $\gamma$ is the *discounting factor*; it controls the degree to which rewards in the distant future affect the total value of a policy and is usually just slightly less than 1.

Given definitions of the transition probabilities and the expected reinforcements, it is possible to solve for the optimal policy, using methods from dynamic programming. A more interesting case occurs when we wish simultaneously to learn the dynamics of the world and to construct the policy. Watkins' Q learning algorithm gives us an elegant and efficient method for doing this.

Let $Q^*(s, a)$ be the expected discounted reinforcement for taking action $a$ in situation $s$ and continuing thereafter with the optimal policy. It can be recursively defined by

$$Q^*(s, a) = er(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a' \in A} Q^*(s', a') \ .$$

Because we do not know $T$ and $er$ initially, we construct incremental estimates of the $Q$ values on line. Starting with $Q(s, a)$ at any value (but typically 0), every time an action is taken, we update the $Q$ values as follows:

$$Q(s, a) := (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a' \in A} Q(s', a')) \ ,$$

where $r$ is the actual reinforcement value received for taking action $a$ in situation $s$, $s'$ is the next situation, and $\alpha$ is a learning rate (between 0 and 1).

Given the $Q$ values, there is a policy defined by taking, in any situation $s$, the action $a$ that maximizes $Q(s, a)$. Watkins showed [Watkins, 1989] that, given some assumptions, including that every situation-action pair is tried infinitely often on an infinite run, the $Q$ values will converge to the true $Q^*$ values, and hence the induced policy will converge to the optimal one.

Of course, in any practical situation, as the learned $Q$ values converge to the true $Q^*$ values, we will have to use the policy to control action. But in doing so, we are in danger of violating the requirement that every situation-action pair be tried infinitely often. This is an instance of the exploration versus exploitation trade-off; it has been treated extensively elsewhere [Kaelbling, 1993a, Thrun, 1992]. In the interest of simplicity in this paper, we generate actions probabilistically based on the $Q$ values using a Boltzmann distribution. Given a situation $s$, we choose action $a$ with probability

$$\frac{e^{Q(a,s)/T}}{\sum_{a \in A} e^{Q(a,s)/T}} \ .$$

This serves to choose actions whose values are much better than the others with much greater likelihood. The *temperature* parameter $T$ controls the amount of exploration (the degree to which actions other than the one with the best $Q$ value are taken).

## 2.2 DG Learning

The DG learning method applies only to domains in which the aim of the agent is to arrive at some goal situation in the least number of steps. We still assume that the world makes stochastic transitions, but the goal is explicitly named and there is no reinforcement function. The aim of DG learning, at its simplest, is to arrive at the policy that minimizes the expected number of steps to the goal. The method is applicable without change to the case where the goal varies on line: now a policy is a mapping from $S \times S$ to $A$, specifying an action to take based on the current situation and the goal situation.

We define $DG^*(s, a, g)$ to be the expected number of steps required to get to situation $g$ from situation $s$ by starting with action $a$ and continuing with the optimal policy. It is conveniently described recursively as

$$DG^*(s, a, g) = 0$$

if $s = g$ and

$$DG^*(s, a, g) = 1 + \sum_{s' \in \mathcal{S}} T(s, a, s') \min_{a' \in A} DG^*(s', a', g)$$

otherwise.

Rather than learning $T$ then calculating *DG*, we estimate *DG* directly on line. Starting with $DG(s, a, g)$ at any value (but typically 0), every time an action is taken, we update the *DG* values as follows:

$$\begin{aligned} DG(s, a, g) \ &:= \ (1 - \alpha)DG(s, a, g) \\ &\quad + \alpha(1 + \min_{a' \in A} DG(s', a', g)) \ , \end{aligned}$$

where $\alpha$ is the learning rate and $s'$ is the next situation.

A policy can be directly constructed from the *DG* values by choosing, for every current situation $s$ and goal situation $g$, the action $a$ that minimizes $DG(s, a, g)$. Although no theoretical results have yet been developed with respect to this learning procedure, we conjecture that it can be shown to converge under restrictions similar to those for the Q-learning result.

The trade-off between exploration and exploitation is important in DG learning as well; we use the Boltzmann distribution to construct action probabilities, though interval estimation techniques [Kaelbling, 1993a] could be employed for more careful exploration.

### 2.3 Updating all Goals in DG Learning

A very simple kind of extra learning step can be directly added to DG learning. Whenever an action $a$ is taken as a step from situation $s$ to a goal $g$, we update the value of $DG(s, a, g)$ by looking ahead one step at the value of $DG(s', a', g)$ for the maximizing $a'$. This look-ahead process hinges on the relationship between $s$, $a$, and $s'$; the goal is relevant only for bookkeeping purposes.

We define *all-goals* updating mode for DG learning to update $DG(s, a, g')$ for all $g' \in S$, independent of what goal was being sought when the action was taken. This updating mode requires an amount of work linear in the size of the set of situations; if this cost is prohibitive, a random or systematically-chosen subset of goals could be updated on each iteration.

The work that is done in updating all goals will not speed learning or performance for the particular goal that is being achieved; but it *will* result in an extremely efficient transfer of knowledge to the achievement of other goals. Singh [Singh, 1992c] has also addressed this issue, but in a more complex network architecture and for a different class of goals. We have found that DG learning with all-goals updating performs very well, learning to achieve changing goals with an order of magnitude less data than is required by conventional Q learning [Kaelbling, 1993b].

### 2.4 Computational Complexity of DG Learning

We are interested in three complexity measures of the learning algorithms: execution time without learning, execution time with learning, and space. Each step, without learning, requires finding the action with the maximum *DG* value for the given start and goal situations; assuming that array indexing is constant time, this requires $O(|A|)$ time, where $A$ is the set of possible basic actions. If we perform all-goals updating, it requires an additional $O(|S||A|)$ time per instance. The *DG* values require $O(|S|^2|A|)$ space.

## 3 LANDMARK NETWORKS

When we plan a route for driving a long distance, we don't even attempt to find the optimal path; rather, we take advantage of an existing path hierarchy. We plan a route from our current location to the nearest freeway on-ramp, plan a series of legs on freeway segments, then plan a route from the freeway off-ramp to our destination. The hierarchical DG learning algorithm presented in this paper will use a related hierarchical structure known as a *landmark network*.

Given a set, $S$, of situations, a landmark network is specified by the tuple $\langle L, NL, N \rangle$, where $L \subset S$ is a distinguished set of *landmark* situations, $NL : S \rightarrow L$ is a mapping from each situation to its nearest landmark, and $N : L \rightarrow 2^L$ is a mapping from each landmark to a set of neighbor landmarks. For such a structure to make sense, we need a distance metric on $S$; the most useful one would be expected

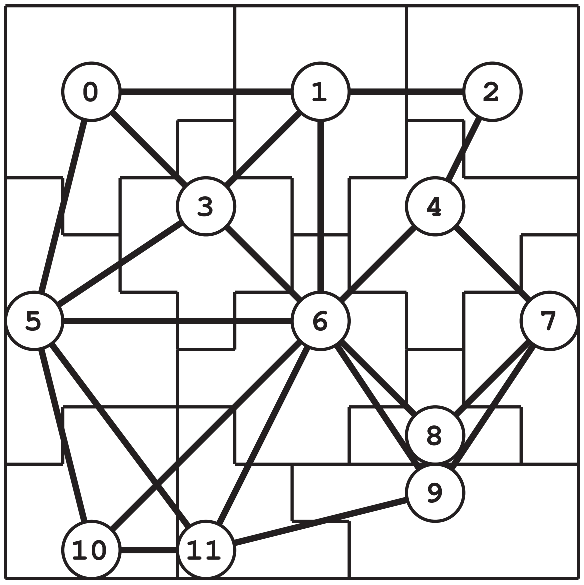


Figure 1: Landmark network constructed on a discrete space

number of steps needed to get from situation to situation using the optimal strategy. We will not have that information available to us initially, however, so any approximation that allows us to partition $S$ into subsets that contain situations relatively near each other will do. At the end of the paper, we speculate on how to learn a good landmark structure.

The $N$ function defines a connectivity graph among the landmarks; it provides an abstract level at which to plan paths. From a landmark, $l$, we can choose to move toward any landmark in $N(l)$. If the graph is highly connected, then there are lots of alternatives and the routes will be very efficient. But high connectivity increases the branching factor of our search, potentially making planning and learning less efficient.

A landmark network can be constructed for any space of situations, but navigation in cartesian space is a good illustrative example because we have a great deal of intuitive experience with such spaces. One way to construct plausible landmark networks in cartesian space is to use the Delaunay triangulation [Preparata and Shamos, 1985]. The landmark situations can be distributed in any way over the space. Then, a Voronoi diagram is constructed on those landmarks, partitioning the space into regions of situations that share the same closest landmark. Finally, any two landmarks whose Voronoi regions touch are connected. Even in discrete spaces that do not satisfy the triangle inequality, it is possible to perform an analogous partition. Figure 1 shows a landmark network constructed in this way on a discrete space. Each square in the grid is a situation; the ones with circles are landmarks. The light rectilinear lines indicate the neighborhood boundaries, and the heavy lines indicate the neighbor relation among landmarks.

## 4 HDG LEARNING ALGORITHM

Given a landmark network, we can modify the DG learning algorithm to work in the hierarchical space; we will call this algorithm HDG for *hierarchical distance to goal* learning. We first describe the internal structures of the algorithm and how they can be used to select actions, then provide a method for learning them.

### 4.1 Acting hierarchically

In a deterministic domain, it is easy to construct hierarchical navigation algorithms. First you move to the nearest landmark, then you move along the edges in the landmark network until you're at the landmark nearest the goal, then you move to the goal. In stochastic domains, it is more difficult, because there is no nominal path between landmarks that can be followed ballistically (using the previous analogy, it is possible to mistakenly drive off the freeway at any moment). Therefore, rather than using landmarks as way-points, we use them as locations to aim for, reducing the number of possible goals for DG learning. This kind of navigation more closely resembles the navigation done in sailing or flying rather than navigation on freeways.

The basic operation of the HDG algorithm is as follows: given a current situation $s$ and a goal situation $g$,

1. Find nearest landmark to $s$, $NL(s)$, and nearest landmark to $g$, $NL(g)$.
2. If $NL(s) = NL(g)$, then execute the best local action for getting from $s$ to $g$.
3. Otherwise, let $l_i$ be the second landmark on the shortest path from $NL(s)$ to $NL(g)$.
4. Choose the best local action for getting from $s$ to $l_i$ and execute it.

In order to support this algorithm we need two data structures. At the low level, we need to store *DG* values from every situation $s$ to every other situation $s'$ such that $NL(s) = NL(s')$. These values will allow us to choose the best actions to take in final phase of getting to the goal once in the correct region (step 2 above). In addition, we need *DG* values from every situation $s$ to every landmark in $N(NL(s))$; that is, to every landmark that is a neighbor of the nearest landmark (step 4 above). At the next level of abstraction, we need to be able to find shortest paths between landmarks. We define $\Gamma(l_1, l_2, l_3)$ to be the shortest distance from landmark $l_1$ to $l_3$ on a path between landmarks that starts by going to landmark $l_2$, where $l_2$ is a neighbor of $l_1$. Let

$$D(l_1, l_2) = \min_a DG(l_1, a, l_2)$$

for any $l_2 \in N(l_1)$. Then,

$$\Gamma(l_1, l_2, l_3) = D(l_1, l_2) + \min_{l_i} \Gamma(l_2, l_i, l_3)$$

and step 3 above can be accomplished by finding the $l_i \in N(NL(s))$ that minimizes

$$\Gamma(NL(s), l_i, NL(g)) \quad .$$

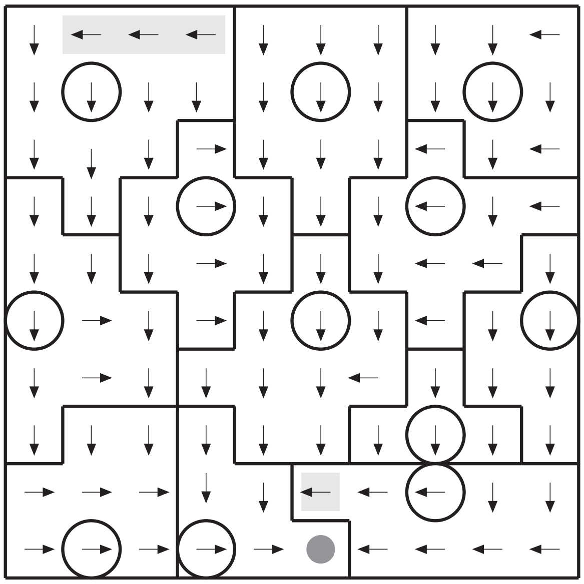

Figure 2: Policy for achieving goal in bottom center of the space.

We can rewrite the algorithm more precisely as:

- If $NL(s) = NL(g)$, then execute the basic action $a$ that minimizes $DG(s, a, g)$.
- Otherwise, let $l_i$ be the landmark that minimizes $\Gamma(NL(s), l_i, NL(g))$ and execute the basic action $a$ that minimizes $DG(s, a, l_i)$.

This algorithm results in a behavior of aiming toward the next appropriate landmark; it is rarely the case that the agent actually goes through the landmark situations, though, because as soon as a new region is entered, the landmark toward which it is aiming is changed. Figure 2 shows the policy derived from the hierarchical action algorithm for the goal indicated by a dark circle in the middle of the bottom row. The squares highlighted in grey indicate situations in which this policy disagrees with the optimal low-level policy. In the upper left corner, the error arises because the shortest path via landmarks includes the landmark in the middle at the far left; rather than aiming straight for the goal, the policy is aiming for that next landmark, resulting in a perturbation of the path to the left. Similarly, in the situation just above the goal situation, the policy is aiming at the landmark in the next region, rather than straight at the goal. The policy for this goal has more of these deviations than those for other goals. This may be an artifact of the simplicity of the domain. Note that these policies are not explicitly stored, but are computed as needed by the method given above.

### 4.2 Learning

The *DG* values are learned just as they are in the basic DG algorithm. We perform all-goals updating, but notice that the space of target goals is much smaller; it encompasses only the other situations in the same partition as well as the neighboring landmarks.

The $\Gamma$ values are defined as a function of the *DG* values. It is difficult to learning them using a standard incremental technique, however, because the landmarks are so rarely actually encountered. Instead, we simply periodically recompute the $\Gamma$ values. We start by setting

$$\Gamma(l_1, l_2, l_2) = D(l_1, l_2)$$

for all $l_2 \in N(l_1)$ and setting

$$\Gamma(l_1, l_2, l_1) = 0$$

for all $l_1, l_2$, then use a modified version of the Floyd-Warshall all-sources shortest-paths algorithm [Cormen *et al.*, 1990] to compute the rest of the $\Gamma$ values. Early in a run, the *DG* values change very rapidly, so the $\Gamma$ values should be recomputed frequently; as the run progresses, the frequency of recomputation can be decreased.

### 4.3 Computational Complexity

Let $|L|$ be the number of landmarks, $|D|$ be an upper bound on the number of neighbors a landmark can have (the degree of the landmark graph), and $|P|$ be an upper bound on the size of a neighborhood. To execute a single step, without learning, we need time $O(1)+O(|D|)+O(|A|)$; it requires constant time to look up the two nearest landmarks in a table, then time on the order of the degree of the landmark graph to find the best next landmark, then time on the order of the number of primitive actions to select the best action. When we add all-goals updating for the *DG* values, it requires time $O(|A|(|P| + |D|))$, because we have to find the best action for every other situation in the same partition as well as for every neighboring landmark. To compute $\Gamma$ values at constant-time intervals requires time $O(|D||L|^3)$, so every execution step takes time $O(|D||L|^3 + |A|(|P| + |D|))$. The cost of computing $\Gamma$ will tend to be the leading factor, but its effect can be limited in two ways. In the following work, the values are recomputed quite rarely, so the cost is amortized over a large number of steps. If $|L|$ gets quite large, then it would be appropriate to construct a further level of hierarchy to reduce this complexity; the final section discusses possibilities for doing this.

It requires space $O(|S|)$ to store a table for $NL$, space $O(|D||L|)$ to store a table for $N$, space $O(|S||A|(|P| + |D|))$ to store the *DG* values, and space $O(|D||L|^2)$ to store the $\Gamma$ values. So the total space complexity is $O(|S||A|(|P| + |D|) + |D||L|^2)$.

Compared to the *DG* algorithm, we trade a factor of $|P| + |D|$ for a factor of $|S|$ in both time and space and incur some extra overhead for the abstraction level. In general, $|P|$ will be approximately equal to $|S|/|L|$, and we can see the tension that arises about the number of landmarks. The more landmarks, the more expense we incur at the abstract level, but the smaller the partitions, and hence, the smaller the *DG* array and the less work required to update it. In addition, the more landmarks there are, the better the paths will be. We can easily limit $|D|$ to a small constant, then consider how to choose $|L|$ optimally. In order to minimize space, we can choose $|L| = |S|^{2/3}|A|^{1/3}$, giving space complexity of $O(|S|^{4/3}|A|)$. To minimize time, we can choose $|L| = |S|^{1/4}|A|^{1/4}$, giving time complexity of $O(|S|^{3/4}|A|)$.

## 5 PRELIMINARY EXPERIMENTAL RESULTS

In this section we present some preliminary experimental results comparing the DG and HDG learning algorithms. We use the domain shown in figure 1 above. Each location on the grid is a single situation, and the actions available to the agent are to move north, south, east, and west. If the agent tries to move through a boundary, it stays where it was. The domain is stochastic, with the agent landing in the nominal square (that is, the square to the north if the north action is taken) with probability .2; it lands in each of the squares neighboring the nominal square with probability .2, as well. This error distribution was chosen arbitrarily; informal testing with other error distributions seemed to yield similar results.

The domain has dynamic goals; as soon as the agent reaches the current goal, the goal is moved to a new location chosen uniformly at random. Both algorithms were tested on runs of length 20,000. For both of them, parameter values $\alpha$ (learning rate) of 0.4 and $T$ (temperature) of 0.1 were determined through experimentation to be best. The algorithms were compared with these parameter settings on 10 runs of length 20,000. The averaged learning curves for each algorithm are shown together in figure 3. The HDG algorithm performs better earlier in the run, but has lower asymptotic performance. This lower performance is due, in part, to the fact that paths are constrained by the landmark network. It may also be due to the fact that there is experimentation taking place at two levels in the HG algorithm; the Boltzmann distribution is used both to choose the best next landmark and to choose the best low-level action. Once the domain is well learned, this extra experimentation contributes to slightly degraded performance. We expect that in larger domains with multiple levels of hierarchy, the difference between the early performance of DG and HDG will be considerably greater.

Both algorithms were implemented quite naively and straightforwardly. In the HDG algorithm, the $\Gamma$ values are recomputed every 1000 ticks. The HDG algorithm runs over 3 times as fast as the DG algorithm.

The HDG algorithm was also tested on randomly-generated landmark networks; the results did not vary widely, which suggests that the method is relatively insensitive to the de-

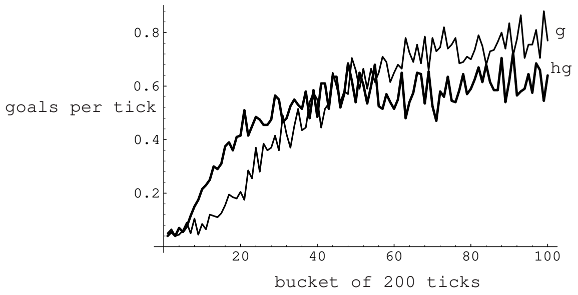


Figure 3: Learning curves of DG and HDG algorithms on stochastic grid world; average of 10 runs of length 20,000

tails of the landmark network.

## 6 RELATED WORK

The most closely related work, using hierarchies in the context of reinforcement learning, is that of Singh [Singh, 1992a, Singh, 1992b]. Singh addresses the slightly different learning problem of *sequential tasks*, in which each of a sequence of subgoals must be achieved in turn. This task structure induces a natural temporal abstraction hierarchy consisting of *macro operations*. This domain has the especially nice property that planning at higher levels of abstraction does not result in sub-optimal solutions. Singh's algorithm learns forward models (transition probabilities and expected reinforcements) at different levels of abstraction, then uses Sutton's Dyna [Sutton, 1990] technique (also used by Whitehead and Ballard [Whitehead and Ballard, 1989]) of simulating experience using the models. He finds that learning is much more efficient when updates are performed using models at a higher level of abstraction. The HDG method applies to a different class of tasks, in which the goal varies over time and is a particular low level situation. In addition, the DG and HDG methods learn without constructing forward models, with the all-goals updating on the *DG* values and the Floyd-Warshall computation of the $\Gamma$ values performing the similar function of propagation information throughout the entire policy.

In his thesis, Lin [Lin, 1993] explores a model of hierarchical learning in which the system first learns elementary "skills" from pre-specified reinforcement functions, then learns how to compose them in order to maximize reinforcement at some higher level of abstraction. This work assumes that the environment is deterministic and that it is possible to "reset" the agent by instantaneously moving it to another part of the state space. This work is difficult to compare to standard Q learning, because some initial training on the basic skills must take place. Lin argues, plausibly, that such training is done in a variety of biological agents, and might be necessary for artificial agents, as well.

Finally, Dayan and Hinton [Dayan and Hinton, 1993] have developed *feudal reinforcement learning*. It has a strict hierarchy of "managers," each of whom can dictate subgoals to managers below them. At every level, the actions are executed until they terminate, and control is given away to the lower level as in a subroutine call. This method works very well in the deterministic domain in which it was tested. However, in stochastic domains, giving control away may mean that an entirely inappropriate low-level behavior is taking place. In addition, as Lin points out, it can be difficult to find termination conditions for low-level actions. The feudal learning method is more generally applicable than HDG, in that it does not rely on having goals of achievement; however, the general reinforcement function is not allowed to change dynamically. The feudal learning algorithm, as it stands, has potential difficulty with the higher levels of abstraction being non-Markovian; however, the authors conjecture that there are techniques available to address this problem.

## 7 FUTURE WORK

Much remains to be done in the exploration of this hierarchical approach to learning to achieve goals, both on the experimental and theoretical levels.

Experimentally, the most convicing validation of the method will be to apply it to a real-world domain. Two domains from different levels of robotics suggest themselves. One would be an extension of the pole-balancing domain, in which the posture of the pole is specified explicitly as a goal; we could ask the system to keep the pole leaning to the left, or to stay on the right third of its track. A more useful problem would be realistic navigation in an outdoor or hallway environment in which exact locations cannot be sensed reliably.

There has been some recent work on approximation algorithms for shortest path problems in cartesian space [Klein and Sairam, 1992]. It may be possible to use these or related results to show that, in certain kinds of domains, the optimal paths in the hierarchical models are within a constant factor in length of the optimal paths in the lowest-level model.

We have only explored a single level of hierarchy in this paper. It is clear how to continue this process: a set of level 2 landmarks can be chosen, inducing a clustering on the level 1 landmarks. Rather than having to learn $\Gamma$ values for every pair of level 1 landmarks, the algorithm can learn them only within neighborhoods, as is done currently with *DG* values. Multiple levels of hierarchy will have to be introduced as state spaces get extremely large.

Finally, we have a bootstrapping problem surrounding the partitioning of the space; getting a useful partition presupposes much of the knowledge required to generate a good strategy. As the *DG* values are learned, the algorithm can revise its partition of the space and the neighborhood relation. In addition, it might be possible to move landmarks dynamically or to add and delete them. One suggestion, due to Chris Watkins, would be to notice landmarks that are rarely chosen as next steps and delete them, adding new landmarks in more useful places.

In order to make learning to achieve goals more practical, we must understand how to do it hierarchically. This paper has presented an initial practical step in that direction.


### Acknowledgements

This work was supported in part by a National Science Foundation National Young Investigator Award IRI-9257592 and in part by ONR Contract N00014-91-4052, ARPA Order 8225.


## References


[Barto *et al.*, 1989] Barto, A. G.; Sutton, R. S.; and Watkins, C. J. C. H. 1989. Learning and sequential decision making. Technical Report 89-95, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts. Also published in *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, Michael Gabriel and John Moore, editors. The MIT Press, Cambridge, Massachusetts, 1991.

[Cormen *et al.*, 1990] Cormen, Thomas H.; Leiserson, Charles E.; and Rivest, Ronals L. 1990. *Introduction to Algorithms*. The MIT Press / McGraw Hill, Cambridge, Massachusetts.

[Dayan and Hinton, 1993] Dayan, Peter and Hinton, Geoffrey E. 1993. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, San Mateo, California. Morgan Kaufmann.

[Kaelbling, 1993a] Kaelbling, Leslie Pack 1993a. *Learning in Embedded Systems*. The MIT Press, Cambridge, Massachusetts. Also available as a PhD Thesis from Stanford University, 1990.

[Kaelbling, 1993b] Kaelbling, Leslie Pack 1993b. Learning to achieve goals. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambéry, France. Morgan Kaufmann.

[Klein and Sairam, 1992] Klein, Philip N. and Sairam, S. 1992. Parallel and dynamic approximation schemes for planar shortest paths. Technical Report 92-61, Computer Science Department, Brown University, Providence, Rhode Island.

[Lin, 1993] Lin, Long-Ji 1993. *Reinforcement Learning for Robots Using Neural Networks*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[Preparata and Shamos, 1985] Preparata, Franco P. and Shamos, Michael Ian 1985. *Computational Geometry: An Introduction*. Springer-Verlag, New York.

[Singh, 1992a] Singh, Satinder Pal 1992a. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, California. AAAI Press. 202–207.

[Singh, 1992b] Singh, Satinder Pal 1992b. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland. Morgan Kaufmann. 406–415.

[Singh, 1992c] Singh, Satinder Pal 1992c. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8(3):323–340.

[Sutton, 1988] Sutton, Richard S. 1988. Learning to predict by the method of temporal differences. *Machine Learning* 3(1):9–44.

[Sutton, 1990] Sutton, Richard S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas. Morgan Kaufmann.

[Thrun, 1992] Thrun, Sebastian B. 1992. The role of exploration in learning control. In White, David A. and Sofge, Donald A., editors 1992, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold, New York.

[Watkins, 1989] Watkins, C. J. C. H. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge.

[Whitehead and Ballard, 1989] Whitehead, Steven D. and Ballard, Dana H. 1989. A role for anticipation in reactive systems that learn. In *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, New York. Morgan Kaufmann. 354–357.