BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-94-M11

"Hierarchical Learning in Stochastic Domains"

by

Rachita (Ronny) Ashar

Hierarchical Learning in Stochastic Domains

Rachita (Ronny) Ashar

Computer Science Department
Box 1910
Brown University
Providence, RI 02912
ra@cs.brown.edu

May 1994

This thesis by Rachita (Ronny) Ashar is accepted in its present form by the Department of Computer Science as satisfying the requirement for the Master of Science degree.

Date 17 May 94

Leslie P. Kaelbling

Approved by the Graduate Council

Date _____

Contents

1 Introduction 1

2 Foundation 4

2.1 Q Learning 5

2.2 Method of exploration 7

3 Dynamically Changing Goals of Achievement 8

3.1 Simulated Domains 8

3.2 Gaea 12

3.3 Evaluation Criteria 19

4 DG Learning 21

4.1 Multiple Goals Update 22

4.2 Method of Exploration 23

4.3 Empirical Results 23

4.4 Computational Complexity 25

4.5 Discussion 25

5 Hierarchical Learning 26

5.1 Landmark Networks 26

5.2 HDG Learning Algorithm 28
5.2.1 Executing Actions 28
5.2.2 Data Structures 28
5.2.3 Incrementally Learning DG Values 29
5.2.4 Updating Gamma Values 29
5.2.5 Paths followed by HDG 30
5.3 Empirical Results 31
5.4 Computational Complexity 33
5.5 Discussion 33

6 Incrementally Learning Γ Values 34
6.1 Empirical Results 34
6.2 Computational Complexity 39
6.3 Discussion 40

7 Method of Exploration 41
7.1 Empirical Results 42
7.2 Discussion 44

8 Learning the Hierarchy 46
8.1 HDGL 46
8.2 HDGM 47
8.3 Empirical Results 48
8.4 Computational Complexity 55
8.5 Discussion 56

9 Walls and Membranes 57

9.1 Modifications to the algorithm 57

9.1.1 Mappings 57

9.1.2 DG values 58

9.1.3 The algorithm 58

9.2 Empirical Results 58

10 Multiple Levels of Hierarchy 61

10.1 The Landmark Network 61

10.2 The Algorithm 61

10.2.1 Executing Actions 62

10.2.2 Data Structures 62

10.2.3 Learning 62

10.3 Empirical Results 63

10.4 Computational Complexity 65

10.5 Discussion 66

11 Landmark Layout 70

11.1 Optimal Landmark nodes in a planar graph 71

11.2 Incrementally adjusting landmark layout 73

11.2.1 Kohonen Maps 73

11.3 Discussion 76

12 Related Work 77

12.1 Feudal Reinforcement Learning 77

12.2 **Reinforcement Learning with a hierarchy of abstract models** 78
12.3 **Parti-game Algorithm** 79
12.4 **Ariadne's Clew Algorithm** 80

13 Conclusion **82**
13.1 **Summary and Future Work** 82
13.2 **Perspective** 85

Prologue

I like to think of flexible systems (be they carbon-based or silicon-based) that have the potential of evolving into independent thinking beings as *Unconventional Intelligences* or *UI's*. A quest for UI's inspires me to play with learning systems and robots. This thesis constitutes my first endeavor towards the quest for UI's.

My parents gave me a ZX Spectrum home computer when I was in the tenth grade. Dr. Ralph Hollingsworth, Muskingum College, provided encouragement when I needed it. Dr. Leslie Kaelbling taught me to do painstaking research, to make slides, to write papers and much more. Andrew Zolli added Geoff's pickles, Nietschzean conversations and chaotic games to life in Providence. I immensely thank all these people

This thesis is dedicated to my father who shewed me to value the truth and to dream big.

Chapter 1

Introduction

This thesis explores feasible methods for teaching autonomous and adaptive intelligent systems to accomplish dynamically changing goals in stochastic domains. The methods that we experimented with have been tested in various simulated domains. We provide an empirical and theoretical complexity analysis for these methods.

Imagine *programming* a robot to pick up interesting rock samples from the surface of Mars and collect them in some section of a spaceship.

You want this robot to be adaptive while functioning in a complex, unknown terrain. Even if you knew precisely what task the robot had to accomplish, and exactly how the robot worked (i.e., it had precise, perfect sensors), it would be at best extremely time-consuming, and at worst impossible to hand-code a static program that enabled such a robot to survive and accomplish its goal. Even if the robot were not autonomous, and you could constantly change your program very quickly, you would then have to assume the role of a baby-sitter for the robot.

Imagine *teaching* a child to collect her toys that are scattered across the

carpet and place them in a box.

It is highly unlikely that you will try to give precise instructions to the child indicating how she must pick up a toy, how she must walk, how many steps she must take, and exactly where in the box she must place a toy. You could demonstrate that task to the child and ask her to imitate you. That approach is akin to supervised learning. But, if she is self-styled or if you want her to be self-motivated at this task, the above approach might not fly. You could still accomplish your goal. Your strategy could be to not explicitly order the child, but merely to observe her, and, encourage her whenever she happens to place some toy in the box after she has finished playing. You could indeed evolve a strategy that is a conglomeration of the above approaches and works best for the task at hand. That is, you could explicitly mention that toys must be placed in boxes after one has finished playing, demonstrate by yourself placing a toy in the box, and then encourage the child with various kinds of reinforcements to follow your example.

One of the reasons that intelligence is such an intriguing subject is that despite our exquisite communication skills we are often unable to explicitly describe the intelligent tasks that we can so easily perform. Yet, we seem to readily learn these tasks via trial and error by evolving strategies to optimize trials that appear causally linked to beneficial effects.

The approach of training organisms to perform actions to acquire rewards (and to not exhibit behavior that entails negative rewards or punishments) is termed *reinforcement learning* by psychologists. A similar method can be applied to train autonomous agents such as robots.

The reinforcement learning problem is that of an agent placed in a world (which may be dynamically changing) balancing exploratory vs. exploitary actions and interacting with its environment to optimize long-term rewards received. For a succinct formal definition and details about reinforcement learning refer to [2].

In this thesis we first review some existing reinforcement learning algorithms and discuss the results that we obtained by replicating certain experiments comparing these algorithms in simulated domains. We then motivate reasons for wanting hierarchical reinforcement learning algorithms, focus on a previously developed hierarchical algorithm, and discuss the various modifications that we have made and the versions of that algorithm that we have spawned in an attempt to make that algorithm robust for large, complex domains. We do a complexity analysis and compare empirical results obtained by running the above-mentioned algorithms. We also review some other hierarchical algorithms from literature and contrast our approach with those methods. We conclude with some speculations on how this work can be extended.

Typically, while developing our algorithms, we make some simplifying assumptions. However, with a little ingenuity, the same algorithms can later be adapted for domains in which those assumptions no longer hold. In this work, we focused on reinforcement learning algorithms in Markov domains. The algorithms can potentially be integrated into a system that also has other components using other forms of learning.

Chapter 2

Foundation

Consider a learning agent named Zorbi. Zorbi can be a robot or a softbot or a small component of some other learning system. The domain Zorbi lives in can range from the surface of mars to a sub-section of a huge data-base. Let S denote the set of world states that are perceived by Zorbi, and, let A be the set of actions that Zorbi can perform. We model the world as a Markov process and consequently we deal with discrete time intervals. Hence, given that the world is in state s_1 at time t_1, and that Zorbi performs action a_1 at that time, then, the world will transition to some new state at time t_2, and, what state it transitions to is contingent solely on s_1 and a_1. Zorbi uses a policy while selecting actions. A policy is simply a mapping from states to actions, $\Pi : S \mapsto A$. Zorbi receives some reward from the world upon performing an action in the world. Note that we model the state transitions and reward functions as probability distributions. Let $T(s_1, a, s_2)$ represent the probability of transitioning from state s_1 to state s_2 upon performing some action $a \in A$. Let $R(s, a)$ be the reward received upon performing action a while in state s. Zorbi's goal is to derive and execute optimal policies, and thereby to optimize

the rewards received.

Note that ideally we want an agent to follow a policy that will optimize its total expected reinforcement. However, if learning proceeds infinitely, it is difficult to mathematically model such reinforcement values. Hence, we introduce a decay factor, γ, and decay future reinforcements by this value. If $ER(t)$ indicates the reinforcement expected at time step t, then we focus on optimizing

$$\sum_{t=0}^{\infty} \gamma^t ER(t) \; .$$

Although, in several applications the task might involve optimizing a measure of performance other than total expected reinforcement. Recent reinforcement learning literature indicates that algorithms to achieve such tasks are being actively researched. Schwartz formulated the average-adjusted value as a metric for such tasks and presented the R learning algorithm that is analogous to Q learning [14].

Stochastic dynamic programming methods such as value iteration and policy iteration [2] can be used to find optimal policies if we have the transition probabily function, T, at our disposal. But, in several important applications such information about the domain is not available or highly inaccurate.

2.1 Q Learning

Watkins' Q learning is an elegant reinforcement learning algorithm [9]. Q learning is an on-line version of value iteration that runs in domains where the transition probability function T is not available, by maintaining the Q values for each state-action pair and backpropogating reinforcement values

across states.

Let $Q : SXA \mapsto Value$. The Q values are arbitrarily initialized. As learning progresses, at each step we update

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

where α is the learning rate, s was the old state of the world in which Zorbi performed action a, and s' is the new state that the world transitions to.

The expected discounted reinforcement for taking action a while in state s and continuing to act optimally is given as

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

Given certain assumptions (such as that all state-action pairs be tried infinitely often), the Q values are guaranteed to converge to the optimal values[17].

As learning progresses, the Q values converge towards optimal, and then it is in the agent's interest to exploit this knowledge and perform an action a that optimizes $Q(s,a)$. However, initially the Q values are largely inaccurate. At that point the agent must perform a great deal of exploration in order to incrementally update its Q values. Gradually, as learning proceeds, the exploration vs. exploitation ratio must change. In practice, this exploration vs. exploitation trade-off is resolved by using a suitable method (for example, Boltzmann distribution) to stochastically select actions.

2.2 Method of exploration

In our Q learning experiments we used a Boltzmann distribution to stochastically select actions.

When Zorbi was in some state s, she selected an action a with probability

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

where T is the temperature parameter of the Boltzmann distribution. T should be in the (0,1] range. A higher temperature introduces a high degree of randomness in action selction.

Chapter 3

Dynamically Changing Goals of Achievement

Many real-world applications require an agent to optimally achieve a specified goal. Often, this goal is dynamically changing. And, typically the agent does not receive much reinforcement from the environment until this goal is achieved. For example, a courier robot in a hotel or an office might start out at the reception desk, be asked to make a delivery to a certain specified location, be rewarded when it makes the delivery, and then be requested to make another delivery to a different location.

We tested the Q learning algorithm and the DG learning algorithm (discussed in the next section) and the hierarchical algorithms that we have been developing (which will be discussed extensively in the following sections) on such goal-of-achievement tasks in simulated domains.

3.1 Simulated Domains

As mentioned earlier, the algorithms discussed here can be applied to various domains. In our experiments, we have applied them to robot navigation tasks

in simulated domains.

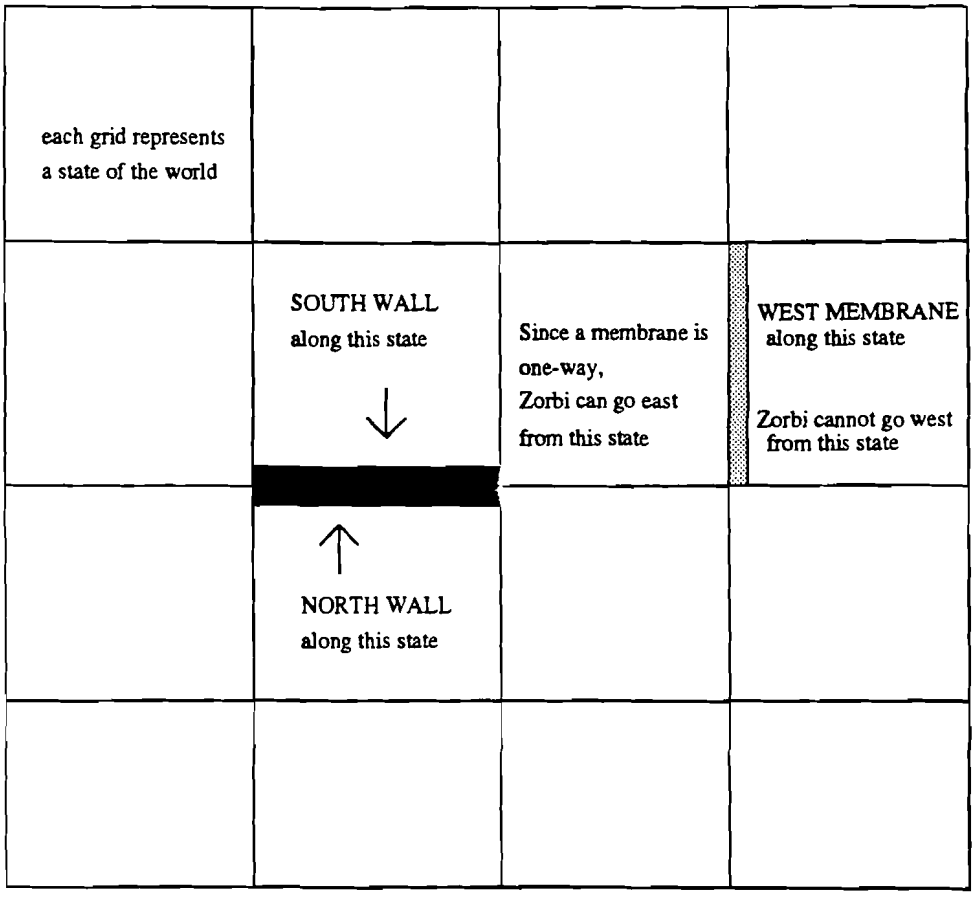

Figure 3.1: a $4X4$ grid-world with walls and membranes.

We worked extensively with simulated grid-world domains of varying sizes and complexities. Figure 3.1 displays a sample domain. As indicated in the legend, each cell represents a state of the world. The very thick black lines constitute walls. The thick shaded lines represent membranes or one-way walls. $A = \{N, S, E, W\}$ constitutes the action set. That is, Zorbi can attempt to

move North, South, East or West in the simulated worlds. Upon performing an action, the state transitioned to is determined probabilistically based on the transition probability parameter of the world. Any attempt to move out of the boundary of the world is futile and results in no change of state.

For our experiments, the transition function was set up such that, with probability *trans_prob*, the world transitioned to the expected state, and with probability $(1 - trans\_prob)/4.0$ the world transitioned to one of the four neighboring states of the expected state. Figure 3.2 illustrates this.

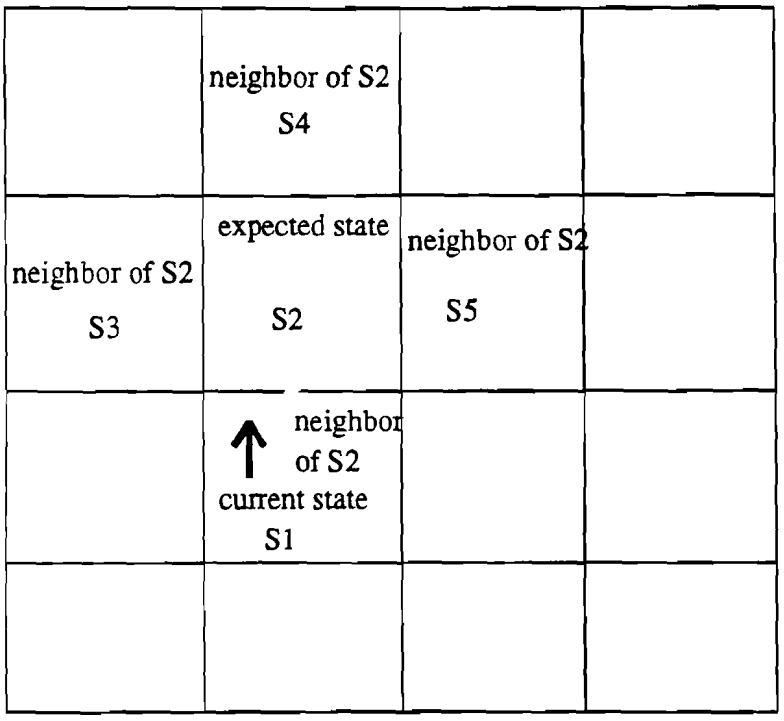

Figure 3.2: Actions and state transitions in grid worlds: if the world is in state $S1$ and Zorbi performs action $NORTH$ then with probability *trans_prob* the world transitions to the expected state, that is, state $S2$. With probability $(1 - trans\_prob)/4.0$ the world transitions to one of the four neighboring states of the expected state. The neighboring states os state $S2$ in this figure are states $S1$, $S3$ $S4$ and $S5$.

3.2 Gaea

We developed a graphical interface simulator called Gaea for running experiments. Gaea is a useful debugging tool since running the algorithms in Gaea and getting dynamic pictorial results helps us better understand how things are really working. Needless to say Gaea adds a colorful twist to our work. Gaea was developed using BAUM (Brown Augmented Utilities for Motif) which is a C++ shell over Motif developed at Brown.

Gaea has various features that are especially relevant for the hierarchical algorithms discussed later. Of course, Gaea also supports the non-hierarchical algorithms that we have been experimenting with. Since all the algorithms have at some point been tested in Gaea, I have placed the Gaea description here so that while reading about the empirical results in various later sections it is easy to visualize the kinds of test domains that were created. However, knowing what the algorithms do and how they work will help better appreciate this section. It is important to know that for the hierarchical algorithms some states of the world are selected to be landmarks. As explained in the later sections, a landmark network is constructed on the world by partitioning the world into regions around these landmark states.

A few of Gaea's salient features are:

- **Main Window:** Gaea has one huge window split-up into left and right sections.
  - **Grid World:** The right half has some menu options at the top boundary. A major portion of the right half contains the graphical

grid-world as illustarated in figure 3.3.

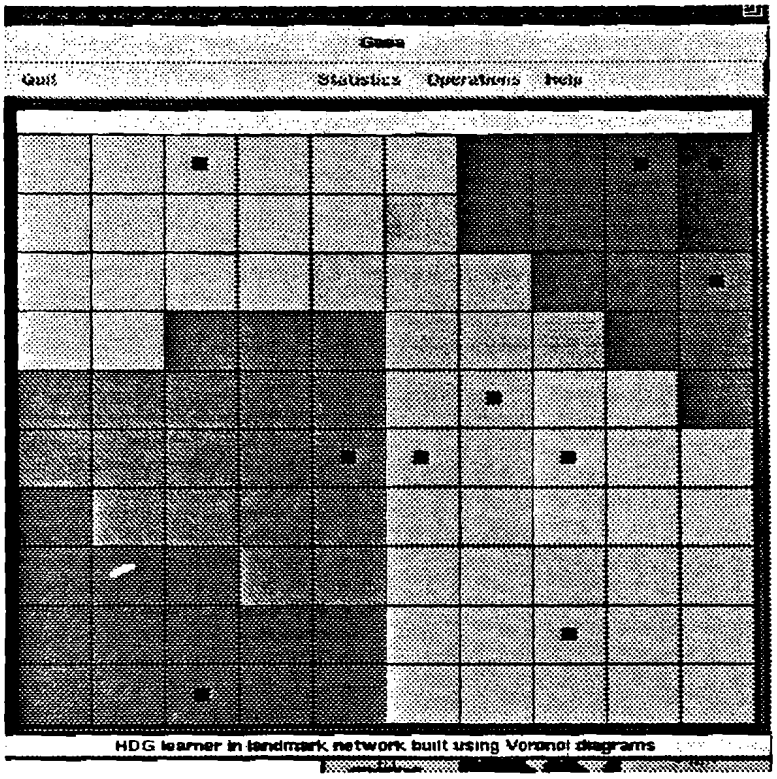

Figure 3.3: Gaea's Grid World Window

In the $10X10$ hierarchical world of figure 3.3 the partitions have been constructed using Voronoi diagrams. The grid-world can have walls and membranes as indicated in figure 3.4. The thinner lines are the membranes and the thicker lines are the walls. Figure 3.4 shows a $20X20$ hierarchical world in which Zorbi the learner has just started learning partitions. Figure reff18 in chapter 9 illustrates the partitions that are learned in a similar world when the run completes. The black squares in some states of the grid-world indicate that those states are landmarks. Each landmark has its own color,

and states belonging to a landmark's region have the same color as the landmark state. When the hierarchical algorithms are learning to partition states into regions, the regions are constantly changing, and those changes are reflected graphically in the grid-world by changing the colors of the states. That makes an enchanting display.

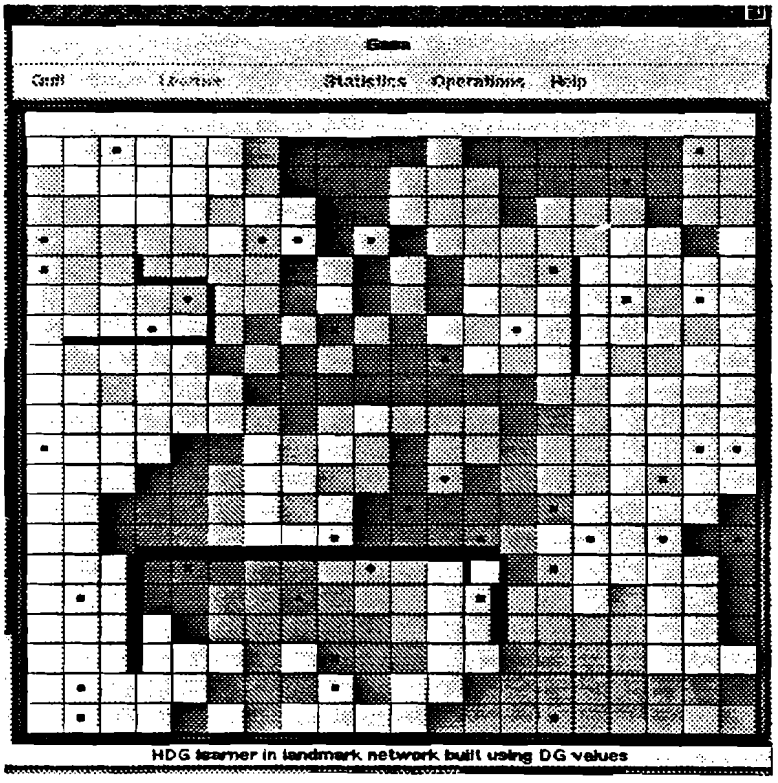

Figure 3.4: a $20X20$ grid-world with walls and memebranes and 40 landmark states.

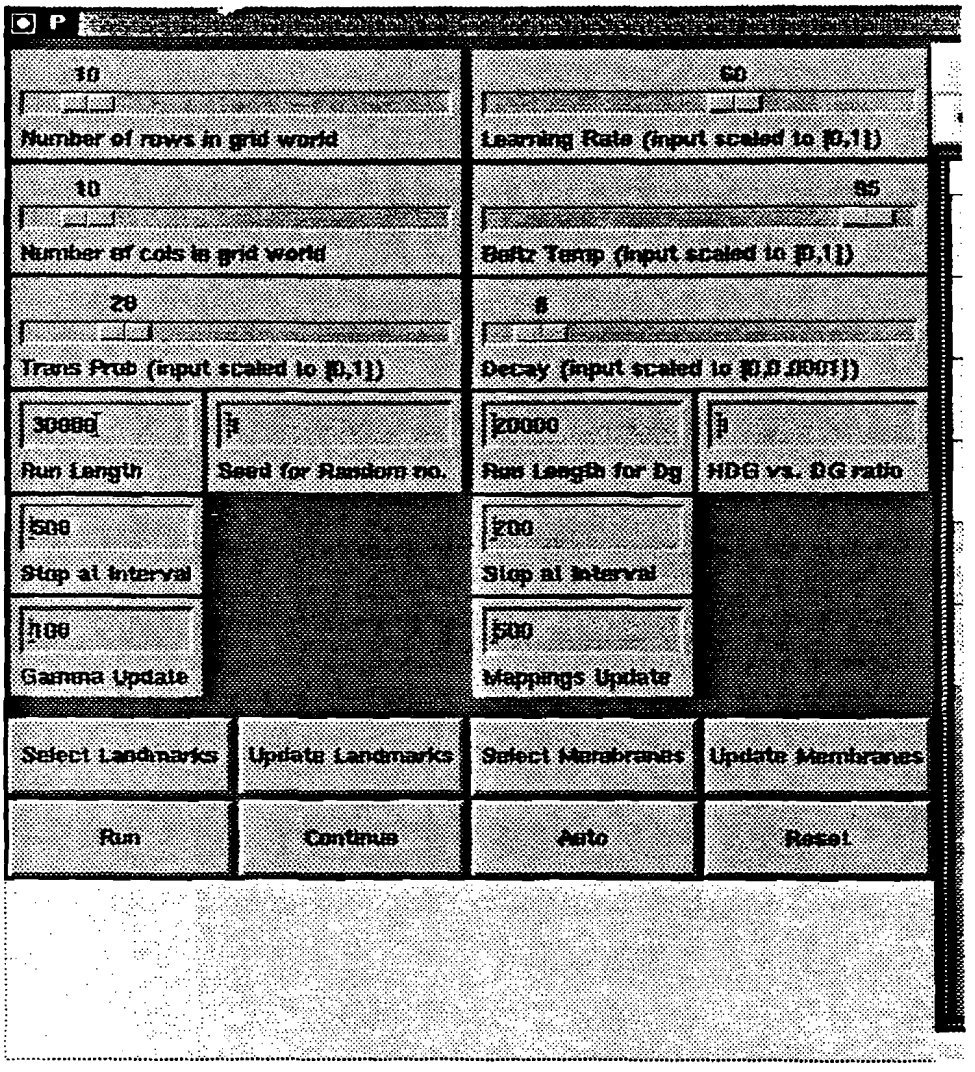

Figure 3.5: Gaea's User Interface

– **User Interface for Parameters:** The left half has various switches that allow users to enter the learning parameters. This is illustrated in figure 3.5.

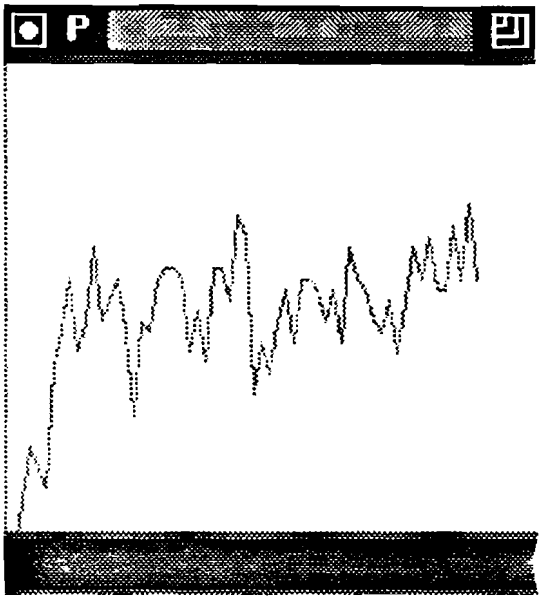

Figure 3.6: Learning curve displayed in Gaea as a run progresses. The Y-axis represents the average number of goals reached per tick, and the X-axis represents the number of steps taken.

- **Running Curve:** Gaea has a tiny window that dynamically displays a learning curve plotting the average number of goals reached per tick as the run progresses. Figure 3.6 illustrates a sample curve that was obtained by running the DG algorithm in a $10X10$ grid-world for 30,000 steps.

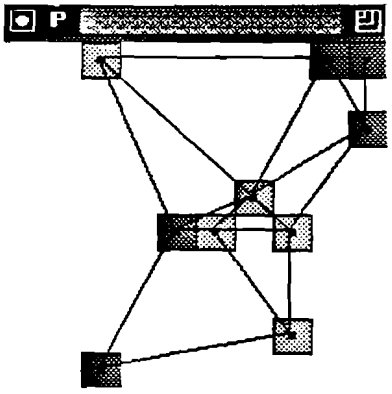

Figure 3.7: A graph of the landmark network for the hierarchical algorithms dynamically plotted in Gaea. Each node of this graph represents a landmark state, and, as a neighborhood relationships between landmarks are formed/dissolved, arcs connecting their corresponding nodes are added/deleted in the graph.

- **Landmark Network Graph Window:** The other tiny window, shown in figure 3.7, plots a graph of the landmark network for the hierarchical algorithms. The landmark network is explained in the HDG algorithm section. In the graph plotted, the landmark states are the nodes, and, neighboring landmarks are connected by arcs. Gaea offers various other options that facilitate development of the hierarchical algorithms. When the partitions are being learned by DGHDG, each time a partition update is performed the regions displayed in Gaea's world window, and, the landmark network in Gaea's network window dynamically change to reflect the new world structure.

- **Running Options:** Gaea can run the algorithms for the specified number of steps while in automode. Alternatively, a user can toggle off the automode and pause at desired break-points.

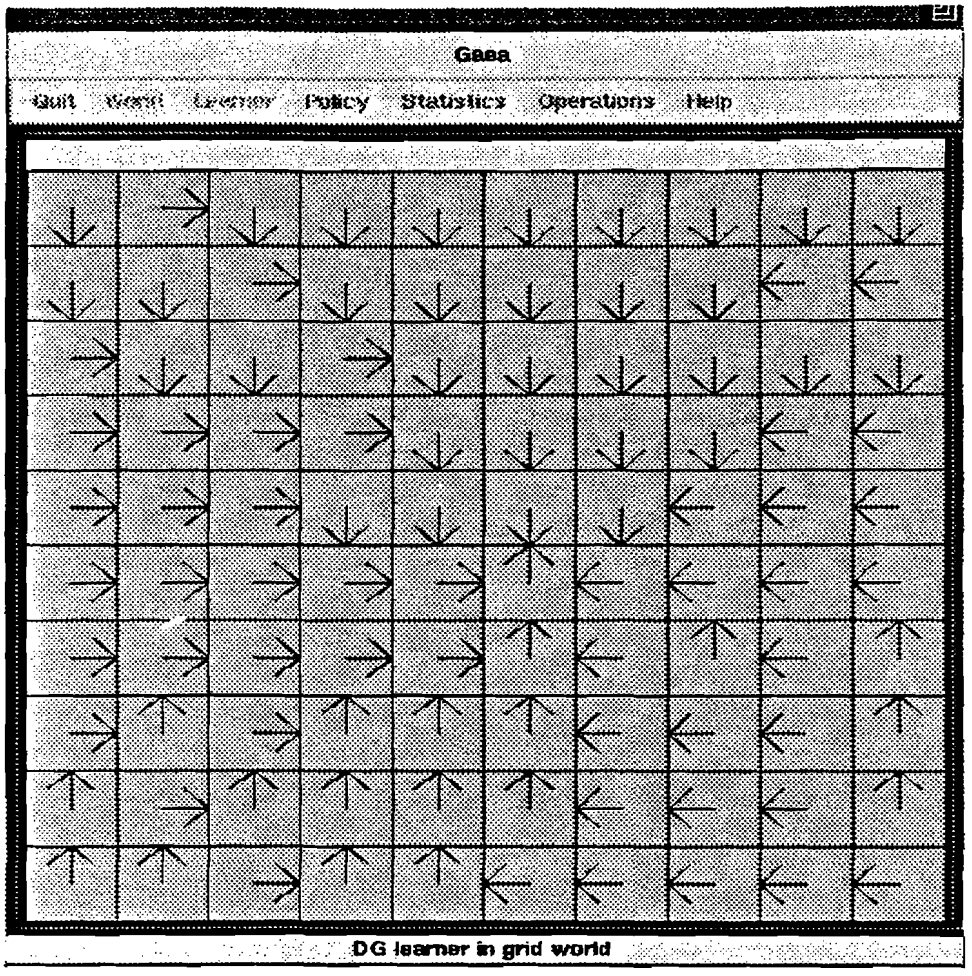

Figure 3.8: this figure illustrates the policy learned by DG for goal state 55. The up, down, right and left directions of the arrows correspond to actions North, South, East and West respectively

- **Displaying Policies:** The policy option allows the user to specify a goal state and display the learned optimal policy from every other state of the world to that state. The states of the world are implicitly numbered from 0 (top-left) to $S-1$ (bottom right). Figure 3.8 shows the policy learned for goal-state 55 by running DG learning for 30,000 steps.

- **Statistics:** Relevant statistical information about the learning curve (such as average number of goals reached during the last interval of the run) is available if desired.

- **Saving Files:** The running curves and the constructed worlds can be saved in files.

- **Online help:** Some basic information about reinforcement learning, and instructions on running the algorithms are available in Gaea's Help menu option. Hence a novice user can build-up a good intuition for these algorithms by playing with Gaea.

3.3 Evaluation Criteria

Note that while comparing the performance of our algorithms on goal of achievement tasks, we consider the *performance factor* which gives the average number of times that Zorbi reached the goal state per step. To obtain the performance factor we divide the number of times that Zorbi reached her goal in the course of the entire run by the total run length.

We will often compare the performance of our algorithms with an *optimal agent* in the same domain. An optimal agent executes an optimal (usually hand-coded) policy.

Each algorithm that we discuss was first run several times with varying learning parameters. To obtain suitable parameters for the algorithms, in some cases, we ran the algorithms several times while systematically varying a single parameter at fixed intervals over an exhaustive range, and then selected

the value that produced the best average performance factor over those runs. But, usually, we simply selected values that seemed suitable for a parameter, ran the algorithm several times and narrowed down to a suitable range. After that we varied the parameter value over this small range and values that were determined to be optimal in the course of such ad-hoc experimentation were then used during subsequent runs. For all the results discussed in this thesis each algorithm was run with optimal parameters thus obtained.

While comparing algorithms we ran the student's t-test and tested the performance factors at 5% levels of significance. Unless specified otherwise, whenever we mention that a result was significant we mean significant at 5% level as per the t-test.

Chapter 4

DG Learning

Previously existing reinforcement learning algorithms such as Q learning can be easily applied to goal of achievement tasks. These algorithms can also be modified to handle situations where the goal varies dynamically; however that causes a substantial degradation in performance.

Kaelbling [9] presented the DG learning algorithm which is a descendant of Q learning and learns to efficiently accomplish goal of achievement tasks. This algorithm readily incorporates tasks with dynamically changing goals.

The DG algorithm uses the fact that the goal is explicitly named, and, hence there is no need for a reinforcement function. The algorithm must simply find a policy that minimizes the expected number of steps (i.e., the expected distance) to the goal.

Let G indicate the set of goal states. At any given point, Zorbi's task is to aim for some goal-state $g \in G$ from her present state $s \in S$ by performing some action $a \in A$. Hence, Zorbi must have a policy mapping states to actions for a given goal state.

Following Kaelbling [7], we define the estimated cost of executing action a

while in state s, and thereafter following the optimal policy to get to state g as $DG^*(s,a,g)$, which can be written as

$$DG^*(s,a,g) = \begin{cases} 0 & \text{when } s = g \\ 1 + \sum_{s' \in S} T(s,a,s') min_{a' \in A} DG^*(s',a',g) & \text{otherwise} \end{cases}$$

We start with $DG(s,a,g)$ set to any arbitrary value (usually 0). Then, at each step, upon taking an action a and reaching some new state s', update,

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

where α is the learning rate.

We effectively maintain a running average of the DG values. Initially, we would prefer to have a high α to expedite learning. As learning proceeds, the DG values stabilize and converge towards optimal. At that stage, we prefer a low value for α since a single bad trial could unduly disrupt the learned policy if α is too high. Hence we decay α at each step.

Watkins and Dayan have proved that a Q learning system will converge if certain conditions are satisfied and if all state/action pairs are tried infinitely often [17]. We conjecture that DG learning can be shown to converge under similar conditions.

4.1 Multiple Goals Update

In the case of multiple goals, instead of updating the DG value for the single selected goal $g \in G$, we do the update for all $g \in G$. While this incurs an additional time-cost at each step, it enables Zorbi to perform global learning even as she focuses on achieving a single goal.

Note that the DG values converge towards the actual expected distances between the states of the world, and hence if we run DG in the all-goals mode by setting $G = S$ then a world-model is implicitly learned.

4.2 Method of Exploration

In the set of experiments comparing DG with Q learning we used a Boltzmann distribution to select actions. (We later evolved a new method for stochastically selecting actions that better balances exploration vs. exploitation. That method is discussed later in the hierarchical learning section and was used in later experiments comparing DG learning with the hierarchical algorithms.)

Since the minimum DG value is optimal, while in some state s, Zorbi selects an action a with probability

$$\frac{e^{DG^{max}(s,g)-DG(s,a,g)/T}}{\sum_{a\in A} e^{DG^{max}(s,g)-DG(s,a,g)/T}}$$

where $DG^{max}(s,a) = \max_{a\in A} DG(s,a,g)$.

4.3 Empirical Results

We replicated Kaelbling's results [9] comparing DG and Q learning on goal-of-achievement tasks in $10X10$ grid-world domains.

For those set of experiments, the *trans_prob* parameter of the world was set to 0.2. We used such a low value (that particular value was arbitrarily selected) so that the model may reflect the unpredictable nature of the real-world. During those experiments we had a learning rate $\alpha = 0.4$, and, we did not decay α. For Q learning, the decay-factor γ was set to a high value that

was experimentally determined to be optimal for those experiments. We had $\gamma = 0.995$. The temperature parameter of the Boltzmann distribution was set to 0.1.

We first tested the algorithms on tasks involving a single goal state. We arbitrarily selected state 5 of the $10X10$ grid-world to be the goal. Zorbi started at some random location, learned to converge towards this goal and when she reached the goal she was teleported to a different location. For DG learning, the performance factor averaged over 10 runs was 0.07184 with a standard deviation of 0.01184, and for Q learning the performance factor averaged over 10 runs was 0.07475 with a standard deviation of 0.00927. The performance of the optimal learner averaged over 10 runs was 0.10523 with a standard deviation of 0.00108. The difference between the DG learning and Q learning performance factors is not significant whereas both performance factors are significantly lower than that of the optimal learner.

Next, we tested the algorithms with multiple goal states. Since Q learning cannot really handle multiple goals in its usual form, we modified problem description by considering the goal state to be an additional aspect of the state description. Hence the algorithm essentially had to maintain Q values for a total of $S \times G$ states. For our experiments, we set $G = S$. During that set of experiments, Zorbi started at some random state and aimed for a randomly selected goal $g \in G$. When she reached the goal, a new goal was randomly selected, and, Zorbi was teleported to a different location. We compared the performance of the algorithms averaged over 10 runs each. The learning parameters for both algorithms were set to be the same as in the above single-goal

case. For DG learning in all-goals mode, the performance factor averaged over 10 runs was 0.07334 with a standard deviation of 0.001532, and for Q learning the performance factor averaged over 10 runs was 0.005335 with a standard deviation of 0.0005169. The performance of the optimal learner averaged over 10 runs was 0.10419 with a standard deviation of 0.00123. The Q learning performance factor is significantly lower than the DG learning performance factor and both performance factors are significantly lower than that of the optimal learner.

4.4 Computational Complexity

As explained in [7], with DG learning, at each step we require $O(A)$ time for action selection and $O(GA)$ time for updating the DG values. Hence in the all-goals mode, we require $O(SA)$ time per step.

Storing the DG values entails a space requirement of $O(SAG)$. Thus we require $O(S^2A)$ space in the all-goals mode.

4.5 Discussion

We described the DG learning algorithm. Our experimental results on DG learning in the all-goals mode illustrate that the algorithm successfully achieves transfer of learning across tasks. However, we note that the algorithm does require a considerable amount of time and space while running in the all-goals mode.

Chapter 5

Hierarchical Learning

DG learning is difficult to implement in larger domains owing to time and space costs. A plausible solution is to split up the domain and use some form of divide-and-conquer. Kaelbling [7] presents the HDG learning algorithm that uses a 2-level hierarchy consisting of a partition of the domain and then uses a modified version of the DG algorithm to learn to achieve goals in the partitioned domain.

5.1 Landmark Networks

If you were at home and intended to attend a conference in a different country, on a different continent, how would you accomplish your goal? You might plan out an itinerary that includes going to the airport, catching a flight, disembarking from the flight at the proper city, and, then reaching a certain hotel in that city. Given that high-level plan, you would commence by focusing on getting to the airport from home. This task might include driving down from your garage to a main street, from there to some highway exit and so on until you reach the airport.

People tend to organize spatial information hierarchically, and at each level of hierarchy the information tends to revolve around certain cognitive reference points or landmarks [16]. Effectively, people form cognitive maps by reorganizing the information provided in a conventional map or obtained from the real-world via observations. As explained and illustrated via several interesting examples in [16], while such cognitive maps do not reflect precisely the real-world information, the distortions introduced by such maps are systematic and help people to retain and utilize relevant information.

Kaelbling's HDG learning algorithm exquisitely renders this aspect of human spatial organization while accomplishing dynamically changing goals in stochastic domains. The algorithm uses hierarchical constructs called *Landmark Networks*. At each level of hierarchy, certain states are selected to be landmarks. The world is partitioned into regions by clustering states around the landmarks. Each landmark is associated with other nearby landmarks via a neighborhood mapping. Essentially, intra-regional goals can be accomplished by simple DG learning. For inter-regional goals, the landmarks are used as way-points to aim for, and, the algorithm follows a shortest path along the landmark graph to get to the region corresponding to the goal state [7].

Given the set S of states, a landmark network is specified by the tuple $\langle L, NL, N \rangle$, where $L \subset S$ is a distinguished set of *landmark states*, $NL : S \mapsto L$ is a mapping from each state to its nearest landmark, and $N : L \mapsto 2^L$ is a mapping from each landmark to a set of neighbor landmarks [7].

5.2 HDG Learning Algorithm

The HDG algorithm summarized below has been adapted from [7].

5.2.1 Executing Actions

Given a landmark network (in later sections we will discuss how to create/learn landmark networks), the current state $s \in S$ and the goal state g,

1. Find $NL(s)$ and $NL(g)$, i.e., the nearest landmarks corresponding to s and g.
2. If $NL(s) = NL(g)$, then execute the best local action to reach from s to g.
3. Else, let l_i be the second landmark on the shortest high-level path from $NL(s)$ to $NL(g)$.
4. Perform the best local action to reach from s to l_i.

5.2.2 Data Structures

We must maintain the N and NL mappings for the landmark network and the inter-state distances. There are two structures for maintaining the distance values between states. At the lower level, we maintain the DG values from every state s to every other state s' such that $NL(s) = NL(s')$, and from every state s to every neighboring landmark of its corresponding landmark, that is, to every landmark l_i such that $l_i \in N(NL(s))$. At the next level of abstraction,we maintain Γ values between landmarks. Following [7], we define $\Gamma(l_1, l_2, l_3)$ to be the shortest distance from landmark l_1 to l_3 on a path that starts by visiting landmark l_2 where l_2 is a neighbor of l_1.

Let

$$D(l_1, l_2) = \min_a DG(l_1, a, l_2), \text{ where } l_2 \in N(l_1),$$

then,

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

and the

$$l_i \in N(NL(s)) \text{ that minimizes } \Gamma(NL(s), l_i, NL(g))$$

corresponds to the second landmark on the shortest path from s to g.

5.2.3 Incrementally Learning DG Values

The DG values are learned exactly as they were with the basic DG algorithm. While running DG in all-goals mode, we had $G = S$, and hence explicitly learned DG values from every state to every other state. With HDG, we only maintain explicit DG values from a state s to a systematically selected set of goal states G_s. Specifically, $G_s = NL(s) \cup N(NL(s))$.

5.2.4 Updating Gamma Values

Note that the Γ values are contingent on the DG values, making it difficult to learn them incrementally because DG values are being simultaneously learned. Kaelbling [7] resolves this issue by periodically recomputing the Γ values as follows:

They are initialized such that,

for all landmarks l_1 and for all landmarks l_2 such that $l_2 \in N(l_1)$,

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

and,

for all landmarks l_1 and l_2,

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

Then a modified version of the Floyd-Warshall all-sources shortest-paths algorithm [3] is used to compute the rest of the Γ values.

5.2.5 Paths followed by HDG

An interesting aspect of this algorithm is that during inter-regional travel it results in a behavior of *aiming* towards the next landmark on the path, but does not require that Zorbi actually reach the next landmark. Figure 5.1 displays a path found by HDG for naviagtion from state 70 to goal-state 76. Zorbi notes that goal state 76 is in a different region than state 76 and begins aiming for the next landmark on the shortest path from current landmark state 50 to goal landmark state 77. But Zorbi need not actually reach the intermediate landmark state 55. Upon reaching state 63 she starts aiming for the next nearest landmark that is state 77. This behavior of aiming for but not necessarily reaching intermediate goals results in paths that are on average far better than paths that would result if the landmark graph were rigidly followed. Hence even though the paths traversed by executing HDG can be slightly longer than the paths that could be followed in a non-hierarchical world the penalty tends to be very small.

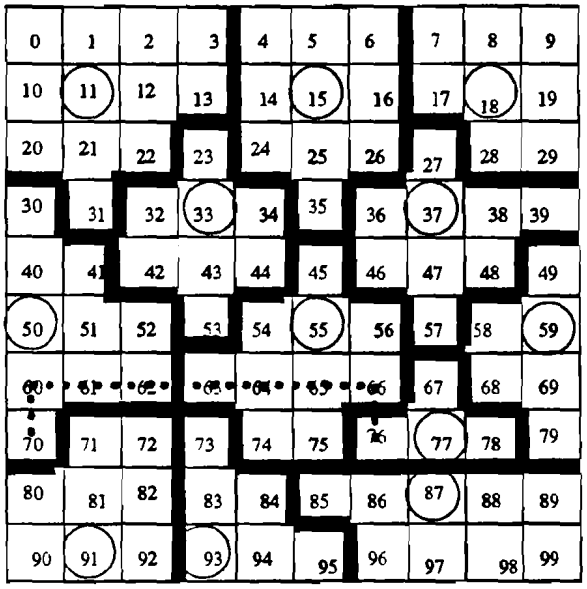

Figure 5.1: The dotted line illustrates a path found by the HDG algorithm to navigate from state 70 to state 76 in a $10X10$ grid world. The states with circles in them are landmarks and the thick black lines represent the partitions on the domain.

Also, all global planning is implicit. At any given moment, Zorbi perceives the state of the world at that moment, and decides what action to perform locally. Hence, if an action results in an unexpected outcome, Zorbi can easily adapt to the new situation. For example, in figure 5.1, if while traversing from 70 to 60, Zorbi ends up reaching 80 instead, she can simply decide that the next landmark on the path from 80 to 76 is 93 and satrt aiming for that landmark.

5.3 Empirical Results

In our initial experiments, following Kaelbling [7], we used Delaunay triangulation [13] to construct landmark networks. We select a certain number of

world states as landmarks, construct a Voronoi diagram on those landmarks and thereby partition the world into regions corresponding to the landmarks. Also, landmarks whose regions are adjacent are considered to be neighbors.

We replicated Kaelbling's [7] results comparing the DG and HDG algorithms in simulated domains. Learning curves comparing DG and HDG in gridworld domains of size $10X10$ and $20X20$ are shown in figures 10 and 11 respectively. Since figures 10 and 11 also plot the learning curve for an algorithm dicussed in chapter 6 they are included in that chapter. For a discussion of those results and details about the learning parameters please refer to section 6.1.

Note that our results from the figure 6.1 experiment indicate that DG learning performs significantly better than HDG learning. The nature of our learning curves is somewhat different than those of Kaelbling's similar experiment [7]. In Kaelbling's experiment in the $10X10$ domain HDG initially performs better than DG but DG performs better in the asymptote. Whereas our learning curves indicate that DG performs better than HDG throughout the run. We had 10 landmark states as opposed to Kaelbling's 12 landmarks. Also, for this set of experiments we independently selected 10 random landmarks for each run. Our run length was $30,000$ and we plotted buckets of 500 ticks as opposed to Kaelbling's runs of length $20,000$ with 200 ticks per bucket.

The learning curves in the $20X20$ grid-world plotted in figure 6.2 appear rather similar to Kaelbling's learning curves in the $10X10$ world. That is, the HDG algorithm initially performs better than DG and DG eventually catches

up. The algorithms do not seem to have reached an asymptote at the end of the runs.

5.4 Computational Complexity

When we introduce a single level of hierarchy wherein $|L|$ is the number of landmarks, $|D|$ is an upper bound on the number of neighbors a landmark can have and $|r|$ is an upper-bound on the region size, we entail time costs of $O(|A|(|r| + |D|))$ at each step, and a cost of $O(|D||L|^3)$ for performing Floyd Warshall update on the Γ values at each set interval [7]. Kaelbling [7] also derives the space complexity $O(|S||A|(|r| + |D|) + |D||L|^2)$.

5.5 Discussion

We motivated reasons for wanting hierarchical algorithms and presented Kaelbling's HDG learning algorithm. We presented empirical results comparing HDG with DG learning (in all-goals mode). While DG does perform better than HDG, the hierarchical algorithm has the potential of being adapted for really large domains since it is faster and requires less space. Also the results from our $20X20$ domain suggest that in larger worlds HDG might initially out-perform DG eventhought DG may perform better asymptotically.

Chapter 6

Incrementally Learning Γ Values

In the earlier version of the algorithm, the Γ values were computed periodically using a modified version of the Floyd Warshall all-sources shortest-paths algorithm [3]. There was a trade-off involved in selecting the update intervals since smaller intervals increased the time requirements of the algorithm whereas larger intervals caused the algorithm's performance to deteriorate.

We introduced an incremental method for updating Γ values. Whenever there is a regional transition, say from the region corresponding to landmark l_1 to the region corresponding to landmark l_2, we update for all $l_i \in L$

$$\Gamma(l_1, l_2, l_i) = (1 - \alpha_2)\Gamma(l_1, l_2, l_i) + \alpha_2(min_{a \in A} DG(l_1, a, l_2) + min_{l' \in L}\Gamma(l_2, l', l_i))$$

where α_2 is the learning rate for Γ.

6.1 Empirical Results

Our experiments in simulated grid-world domains indicated that HDG-Incremental (this newer version of the algorithm) performs as well as HDG-Floyd-Warshall

(the older version) and is far more efficient.

Figures 6.1, 6.2 and 6.3 display the learning curves comparing these newer versions. In all cases, the transition probability parameter of the domain was set to 0.2. In the $10X10$ world, the initial learning rate for DG was 0.75, and for HDG was 0.6. This was eventually decayed to 0.4 with both algorithms. HDG had 10 randomly picked landmark states. In the larger worlds, the learning rate for all algorithms was initially 0.4 and eventually decayed to 0.3. In the $20X20$ domain HDG had 20 randomly selected landmark states, and, in the $30X30$ domain there were 30 randomly distributed landmark states.

In the $10X10$ world, the performance factor for DG learning averaged over 10 runs was 0.08054 with a standard deviation of 0.001542023. For HDG-Floyd-Warshall, we obtained an average performance factor of 0.06116 with a standard deviation of 0.001824654. HDG-Incremental gave an average performance factor of 0.0605 with a standard deviation of 0.001124782. DG learning performed significantly better than the two hierarchical algorithms whereas the difference in performance of the Floyd-Warshall vs. Incremental versions is not significant.

In the $20X20$ world, we obtained different results. In this case DG learning gives an average performance factor of 0.0189 with a standard deviation of 0.001542023. With HDG-Floyd-Warshall we have a performance factor of 0.02528 and a standard deviation of 0.001824654. The Incremental version of HDG produces a performance factor of 0.0245 with a standard deviation of 0.0009362954. Our tests indicate that both the hierarchical algorithms perform significantly better than simple DG learning, and, also the performance of the

Floyd-Warshall version is better than that of the incremental version at the 5% level of significance.

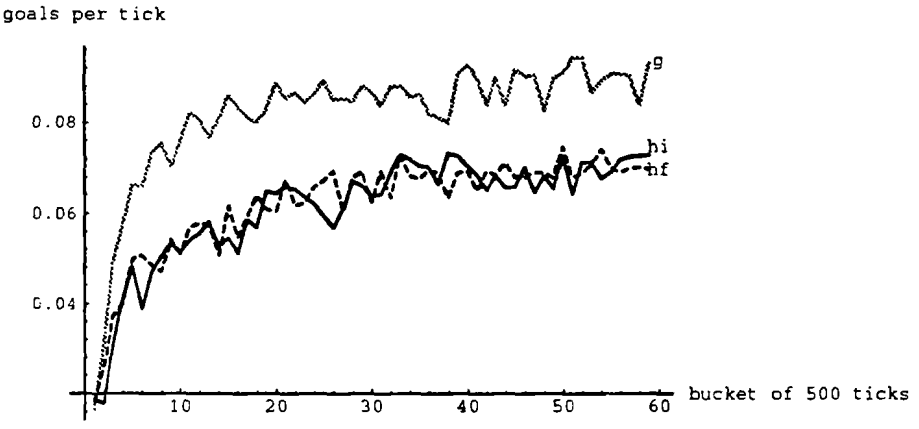

Figure 6.1: g, hi and hf represent the learning curves for DG, HDG-Incremental and HDG-FloydWarshall algorithms respectively in $10X10$ gridworld domains. Each of these curves is averaged over 10 runs of length 30,000 each.

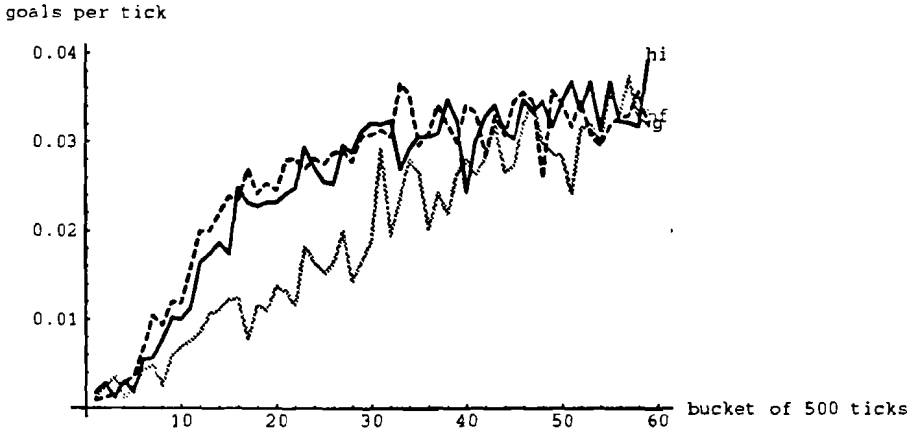

Figure 6.2: g, hi and hf represent the learning curves for DG, HDG-Incremental and HDG-FloydWarshall algorithms respectively in $20X20$ gridworld domains. Each of these curves is averaged over 10 runs of length 30,000 each.

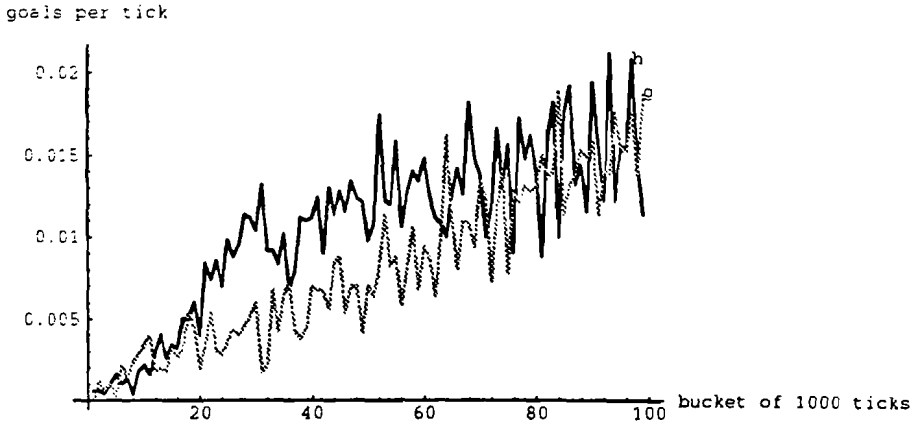

Figure 6.3: g, and h represent the learning curves for DG, HDG-Incremental algorithms respectively in $30X30$ gridworld domains. Each of these curves is over a single run of length 100,000 each.

We compared DG and HDDG-Incremental in a $30X30$ world and our results were similar to those of the $20X20$ world. As indicated by the learning curves plotted in figure 6.3, HDG-Incremental initially performs better than DG and DG eventually catches up. Over a run of length 100,000 DG learning had a performance factor of 0.0076 and HDG had a performance factor of 0.01919.

World Size	Landmarks	Algorithm	Run Length	CPU Time
10 x 10	0	DG	30000	18
10 x 10	10	HDG-Floyd Warshall	30000	23.7
10 x 10	10	HDG-Incremental	30000	9.5
20 x 20	0	DG	30000	77
20 x 20	20	HDG-Floyd Warshall	30000	168
20 x 20	20	HDG-Incremental	30000	17

Table 1: Comparing Running time of the algorithms

Table 1 gives the running time of DG, HDG-Incremental and HDG-Floyd Warshall in $10X10$ and $20X20$ grid-world domains. With the Floyd Warshall versions the Γ values were updated at intervals of 100 steps. The Incremental version of HDG is tremendously faster than the Floyd Warshall version and much faster than DG.

6.2 Computational Complexity

When we introduce a single level of hierarchy wherein $|L|$ is the number of landmarks, $|D|$ is an upper bound on the number of neighbors a landmark can have and $|r|$ is an upper-bound on the region size, we entail time costs of $O(|A|(|r| + |D|))$ at each step. This is exactly the same as the cost incurred by HDG-Floyd Warshall. Learning can cost $O(|A|(r_0 + |D|) + |L|^2|D|)$. since the cost of updating the DG values at the lowest level is $O(|A|(r_0 + |D|))$. The cost of updating the Γ values during a regional transition is $O(|L||D|)$ which is far more efficient than the cost of performing a Floyd Warshall update.

6.3 Discussion

We presented an efficient method to incrementally learn the Γ values. In the $10X10$ domain our experiments indicated that HDG-Incremental performs as well as HDG-Floyd Warshall. In the $20X20$ domain the Floyd Warshall version performs slightly better but the difference between the two algorithms' performance factors is not that large. We derived the computational complexity for the incremental version and it indicates that this version is less complex than the Floyd Warshall version. And, as indicated by table 1 the Incremental version runs much faster. Based on these results we conclude that it is preferable to use the Incremental version of the algorithm.

Our experiments from the $20X20$ and $30X30$ domains indicate that initially HDG performs better and DG eventually catches up. Also, the learning curves do not seem to have reached asymptotic levels during these runs.

Chapter 7

Method of Exploration

Initially, we were using a Boltzmann distribution to stochastically generate actions in the simulator. That method of exploration placed a high level of confidence in the learned DG values and we discovered scenarios wherein a few erratic trials could result in certain actions being rarely explored.

We noted that the Interval Estimation algorithm or IE algorithm [8] resolves the above-mentioned problem by maintaining *confidence intervals*. Since the IE algorithm requires a stationary distribution, it is not directly applicable in our domains.

Instead, we experimented with an alternative method of exploration, called UE (for Uncertainty Estimation), wherein we account for our confidence in a particular DG value by introducing an uncertainty factor. With exploitation probability, p, we let the agent select action a that minimizes $DG(s,a,g) - c/frequency(s,a)$, where c is the confidence factor, and, $frequency(s,a)$ is the number of times action a was attempted from state s. With probability $1-p$, we uniformly select a random action. We also increment p at certain intervals, so that eventually $p = 1 - \epsilon$ for some small constant ϵ.

Intuitively, the confidence factor can be interpreted as the number of times that Zorbi must perform some action from some state before she starts being confident about her knowledge regarding that state action pair. Since our domains are stochastic, even when Zorbi is very confident about her learned values she must continue performing some amount of random exploration. To ensure such occassionally adventurous behavior we let Zorbi perform actions that she considers optimal with probability p and we let p converge towards but never reach 1.0.

UE can also be used to select intermediate landmarks. For each landmark l, we maintain the transition frequency, $TF(l, l_n)$ for all $l_n \in N(l)$, that indicates the number of times the agent made a transition from some state in the region of l to some state in the region of l_n. Let p_1 and c_1 be the exploitation probability and the confidence factor respectively at leevel 1 of the hierarchy. Then, to go from the region of l to some region corresponding to l_g, with probability p_1 the agent aims towards an intermediate landmark l_n that minimizes $\Gamma(l, l_n, l_g) - c_1/TF(l, l_n)$. With probability $1 - p_1$, an intermediate landmark l_n such that $l_n \in N(l)$ is randomly selected.

7.1 Empirical Results

We performed experiments comparing DG-Boltzmann and DG-UE over 5 runs of length 30,000 in 10x10 grid worlds. The transition probability parameter of the domain was set to 0.2 for these experiments. For both algorithms we had a learning rate of 0.75 which was gradually decayed to 0.4. The Boltzmann temperature parameter was set at 0.1. For UE, the confidence factor was set to

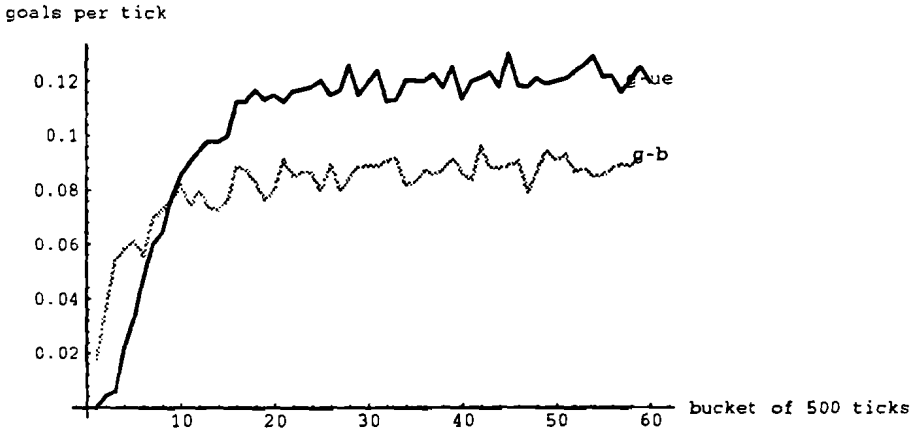

Figure 7.1: g-ue and g-b are the learning curves for DG-UD and DG-Boltzmann respectively in a $10X10$ grid world domain.

20.0. The exploitation probability p was set to 0.95 and eventually increased to 0.995. For DG-Boltzmann, the performance factor averaged over 10 runs was 0.08012 with a standard deviation of 0.0000912, and, for DG-UE we had an average performance factor of 0.104167 with a standard deviation of 0.0010915. The UE version performed significantly better than the Boltzmann version. The learning curves for these algorithms are given in figure 7.1. We observe that the Boltzmann version initially leads. Then, as Zorbi gains confidence, the UE version overtakes the other algorithm. Also, the UE version performs asymptotically better than the Boltzmann version.

We also compared HDG-Boltzmann and HDG-UE in 10x10 grid worlds with 10 randomly selected landmark states. For the HDG algorithms the learning parameters were the same as those used in the experiments dicussed in figure 10. For HDG-UE, at the lower level we had $c = 0.1$, $p = 0.95$

and $\epsilon = 0.005$. At hierarchical level 1, we had $c_1 = 1.0$, $p_1 = 0.6$, and, $\epsilon_1 = 0.005$. We performed 10 runs of length 30,000 each. For HDG-Boltzmann we obtained a performance factor of 0.06262 with a standard deviation of 0.00249, and, for HDG-Incremental-UE a performance factor of 0.0573 with a standard deviation of 0.00201. The difference in performance factors of these two algorithms is not significant.

We have also performed some ad-hoc experiments in $20X20$ domains for runs of length 70,000 and discovered no significant difference between DG-UE and DG-Boltzmann. However, it is possible that we simply haven't fined-tuned UE with optimal parameters for that domain.

7.2 Discussion

We determined that the Boltzmann distribution method of exploration can cause problems in certain situations. We proposed the UE method to resolve those difficulties and expected the UE version of the algorithms to perform better than the Bolzmann version. Our experiments produced results indicating that the DG algorithm performs significantly better with this new method of exploration and that there is no significant differnce in the performance of the HDG algorithm.

Since the UE version of DG is so encouraging we are inclined to perform additional experiments to determine whether the performance of the HDG algorithm can also be enhanced. We have already experimented with several different combinations of values for the UE parameters for HDG. Yet, it might be worth further investigating this issue (specifically, by tweaking the

confidence intervals).

Chapter 8

Learning the Hierarchy

In the HDG version described in [7], the researchers construct the landmark networks. We induce hand-coded partitions to form regions of states corresponding to nearest landmarks by constructing Voronoi diagrams on the landmarks.

However, we would prefer that our algorithms bootstrap from scratch as far as possible. That is, we would like the algorithms to learn or derive the NL and N mappings given incomplete information or no information about the domain. We have been exploring various means of accomplishing this.

8.1 HDGL

We can obviously automate the process of finding the nearest landmark to each state if we have information regarding the distances from states to landmarks. We can run the DG algorithm to *learn* the distances and then use those distances to construct the NL mapping. Given the NL mapping, we can derive the N mapping and induce partitions on the domain. We have implemented this algorithm and call it HDGL.

With the HDGL algorithm, we do the following:

1. start by letting DG learn for some fixed interval
2. use those learned values to bootstrap HDG
3. start running HDG and DG in parallel for some fixed interval
4. use the learned DG values to create new partitions for HDG
5. goto step 3

The advantage of this method is that initially we can run DG only for a short interval and get approximate distances, use those distances to bootstrap HDG, and, then onwards run DG and HDG in parallel. The disadvantage is that we do have to run DG and allocate the time and space required by DG.

8.2 HDGM

HDGL incurs all the time and space costs required for running DG. However, our entire motivation in developing hierarchical algorithms is to reduce the time and space requirements. We would like to have the hierarchical algorithms start learning given negligible information, induce a suitable structure on the domain and not incur too much additional cost. To achieve these goals we propose the HDGM algorithm.

We can bootstrap HDGM with arbitrary mappings. As learning progresses, we can incrementally *modify* these partitions. The HDGM algorithm is given below:

1. Construct N and NL mappings with initial partitions (which may be random allocations)
2. Use the N and NL mappings to construct a landmark network
3. Run the HDG algorithm for some set interval.
4. Modify the N and NL mappings as follows:
   - For each state, s with $NL(s) = l_{old}$, set $NL(s) = ClosestNeighbor(s)$ where the closest neighbor of s is the l_i corresponding to
     $$\min_{l_i \in (N(l_{old}) \cup l_{old})} \min_{a \in A} DG(s, a, l_i)$$
   - For each landmark l, let $l_i \in N(l)$ if there was a transition from s_l to s_{l_i}, where $NL(s_l) = l$ and $NL(s_{l_i}) = l_i$.
     Note that in practice having too many neighbor landmarks increases time and space requirements. Hence, in recent versions, we update the transition frequency from landmark l to landmark l_i each time there is a transition from the region of landmark l to that of l_i, and then let the n (some prespecified constant) landmarks to whose regions states of l most frequently transition be in $N(l)$.
5. Goto step 2.

8.3 Empirical Results

Intuitively, we felt that starting HDGM with somewhat reasonable partitions (HDGM-GOOD) as opposed to completely random partitions (HDGM-RANDOM) should enhance performance. We performed experiments to test

out this intuition. We constructed a grid-world domain with several walls in it. We bootstrapped HDGM-GOOD with partitions that would be optimal if there were no walls in the domain. Hence the initial partitions were good approximations for the domain, but, were not optimal. HDGM-RANDOM was bootstrapped by randomly allocating states to regions.

While referring to figures 8.1 to 8.4 please note that the thick black lines are the walls. The states with black squares in them are landmarks. Each landmark has a different color, and, states allocated to that landmarks region have the same color as the landmark.

Our experiments were performed in a $10X10$ grid-world with 10 landmark states and several walls. The domain is illustrated in figure 8.1. Figure 8.1 also depicts the initial partitions for HDGM-GOOD. We performed 10 runs of the algorithms of length 30,000 each. For these algorithms the learning rate was set at 0.75 and eventually decayed to 0.4. The transition probability parameter of the domain was set at 0.2. We observed that the modified partitions found by the algorithm were impressive in all those runs. Figure 8.2 illustrates a sample partition found during one of those runs after 20,000 steps. Although we ran these algorithms for 30,000 steps and continued modifying the partitions at each update interval throughout the run, the partitions stabilized and rarely changed after about 20,000 steps. Figure 8.3 illustrates the initial partition for HDGM-RANDOM. Note that the domain was exactly the same as that of figure 8.1, even though some of the landmark states have not been plotted in the figure 8.3 picture. Figure 8.4 illustrates a partition found at the end of one such run of HDGM-RANDOM. In all runs, we noted that

the partitions found by this algorithm were sub-optimal, and, the partitions did not stabilize at the end of 30,000 steps. As expected the performance of HDGM-GOOD was tremendously better than that of HDGM-RANDOM. The average learning curves for the two algorithms are plotted in figure 8.5. With HDGM-GOOD, the average performance factor was 0.05632 with a standard deviation of 0.00514507. HDGM-RANDOM produced a performance factor of 0.02991 with a standard deviation of 0.001452176.

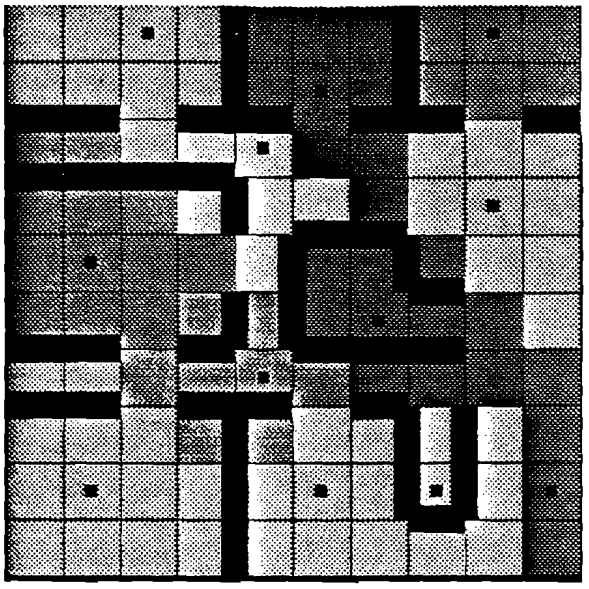

Figure 8.1: This figure illustrates the initial partitions provided to HDGM-GOOD in a $10X10$ grid-world with walls and 10 landmark states.

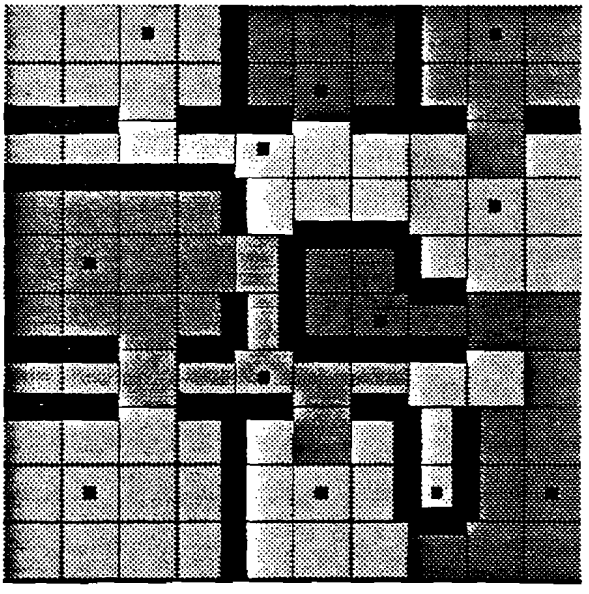

Figure 8.2: This figure illustrates the modified partitions formed on the domian from figure 8.1 during a run of HDGM after 20,000 steps.

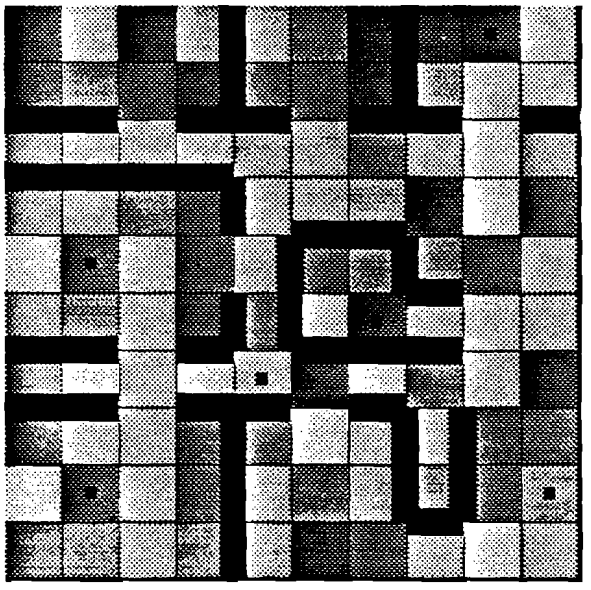

Figure 8.3: This figure illustrates the initial partitions provided to HDGM-RANDOM in a $10X10$ grid-world with walls and 10 landmark states.

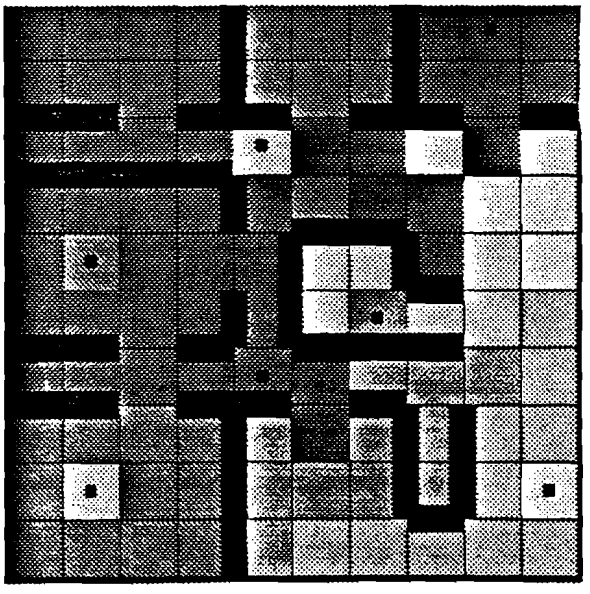

Figure 8.4: This figure illustrates the modified partitions formed on the domian during a run of HDGM-RANDOM after 30,000 steps.

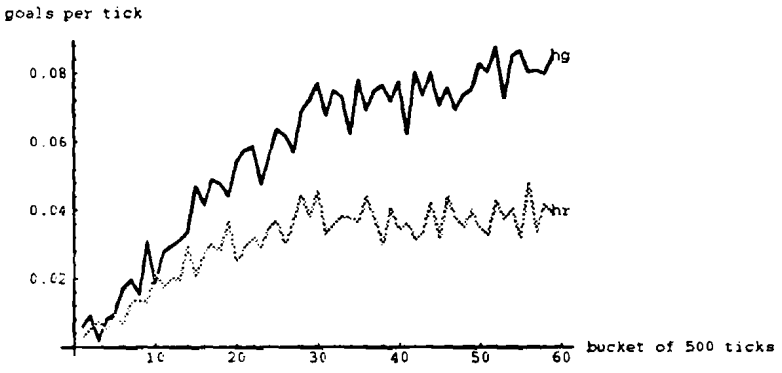

Figure 8.5: In a 10x10 grid-world with 10 landmark states and with walls. hg and hr denote learning curves for HDGM-GOOD and HDGM-RANDOM respectively. Each of these curves is an average of 10 runs; each point represents the average goals achieved per tick for that bucket, where 500 ticks comprise a bucket.

Next, we compared HDGM-GOOD with HDGL. We expected that initially HDGM-GOOD would perform better but in the asymptote both algorithms would display similar performance.

We tested the two algorithms in a domain that was rather similar to that of figure 8.1 and had the same landmark states but the layout of the walls was somewhat different. For these algorithms the learning rate was set at 0.75 and eventually decayed to 0.4. The transition probability parameter of the domain was set at 0.2. The N and NL mappings were modified at update intervals

of 500 steps. Our results indicated that the partitions formed by HDGL were also very nearly optimal although not perfect. The learning curves for the two algorithms are displayed in figure 8.6. The performance factor for HDGL averaged over 10 runs of length 50,000 each was 0.02466 with a standard deviation of 0.001105328. HDGM had a performance factor of 0.0647 with a standard deviation of 0.006027418.

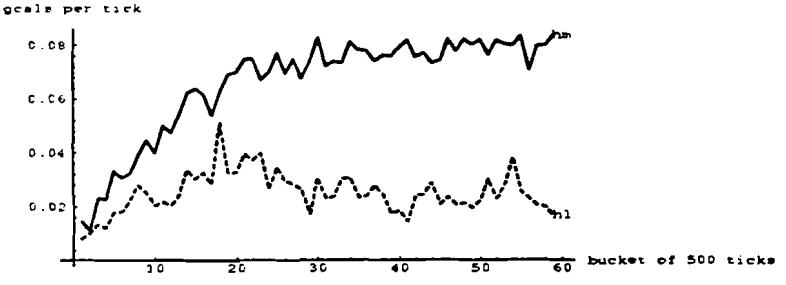

Figure 8.6: In a 10x10 grid-world with 10 landmark states and with walls. ghg and hghg denote learning curves for HDGL-Boltzmann and HDGM-Boltzmann respectively. Each of these curves is an average of 10 runs; each point represents the average goals achieved per tick for that bucket, where 500 ticks comprise a bucket.

Figure 8.6 indicates that HDGM performs asymptotically better than HDGL. Since we have observed that the partitions found by HDGL are very close to optimal this is rather surprising. We speculate that the learning rate might be a relevant factor for this difference. In our current implementation, with both algorithms, we start with a high learning rate, linearly decay it for

World Size	Landmarks	Algorithm	Update Interval	Run Length	CPU Time
10 x 10	10	HDGL	500	30000	61
10 x 10	10	HDGM	500	30000	61
20 x 20	20	HDGL	500	30000	184
20 x 20	20	HDGM	500	30000	168

Table 2: Comparing Running time of the algorithms. The Update Interval of an algorithm denotes the number of steps after which the algorithm modifies its partitions. The CPU time is given in seconds.

a while, and, thereafter logarithmically decay it. With HDGL, since initially the partitions are arbitrary the learning rate ought to be low. At some stage, when reasonable partitions have been formed the learning rate should get really high and then again be gradually decayed. Hence a Gaussian function might be appropriate. We have not yet tested out such a function for the learning rate. But we further speculate that if such a mechanism enhances the performance of HDGL then it should propably cause HDGM-RANDOM to learn better also.

Table 2 gives the CPU times comparing HDGM and HDGL algorithms. In the $10X10$ world both algorithms take the same time to complete a run of length $30,000$. The partitions are updated every 500 steps. In the larger $20X20$ domain HDGM runs faster.

8.4 Computational Complexity

With HDGL, if we run DG and HDG in parallel then our space and time requirements are similar to simply running DG.

With HDGM, we require the same time and space as HDG for learning. While updating the partitions, for each state, we require $O(D)$ time to decide

what region the state belongs in, and hence we need $O(SD)$ time for all states. Also, we require $O(L^2)$ time to construct the neighborhood mappings. Hence we need $O(SD + L^2)$ time to modify partitions, and this time is amortized over a moderately large number of steps.

8.5 Discussion

We noted that we would prefer the hierarchical algorithms to bootstrap given little or no information about the domain. We proposed the HDGL algorithm that runs DG and HDG in parallel and bootstraps from scratch. We also proposed the HDGM algorithm that bootstraps with arbitrary initial partitions and then modifies them to converge towards optimal partitions. Our experimental results verfified the intuition that bootstrapping HDGM with reasonable partitions as opposed to totally random partitions should enhance performance. We further compared HDGM with HDGL and were surprised that the two algorithms do not appear to converge asymptotically. We speculated on reasons for this and proposed a way of resolving the problem. It should be interesting to further investigate this issue in the course of future work.

Chapter 9

Walls and Membranes

We can simulate complex domains by introducing walls. Since the DG values maintained by the algorithms are asymmetric, slight modifications to the algorithms enable us to tackle learning in domains with membranes or one-way walls.

9.1 Modifications to the algorithm

We have to tweak our algorithms to incorporate asymmetric distances.

9.1.1 Mappings

We maintain two sets of nearest landmark mappings, NL_{FROM} and NL_{TO}. In the FROM mappings, each state is in the region of the nearest landmark that can be reached *from* that state, whereas TO mappings place a state in the region of the landmark from which there is an optimal path *to* the state. The N mapping is the almost same as before. That is $l_i \in N(l_1)$ if there is a transition from some state in $NL_{TO}(l_1)$ to some state in $NL_{FROM}(l_i)$.

9.1.2 DG values

We maintain DG values from every state s to its corresponding $FROM$ landmark, $NL_{FROM}(s)$, and, to every neighbor of that landmark, all l_i such that $l_i \in N(NL_{FROM}(s))$. Also, we maintain DG values from every state s to every other state in the region of its NL_{TO} landmark, and, to all the neighbors of its NL_{TO} landmark.

The Γ values are maintained exactly as in the earlier versions of the algorithm.

9.1.3 The algorithm

The HDG algorithm is now given as:

Given a landmark network, the current state $s \in S$ and the goal state g,

1. Find $NL_{FROM}(s)$ and $NL_{TO}(g)$, i.e., the nearest landmarks corresponding to s and g.
2. If $NL_{FROM}(s) = NL_{TO}(g)$, then execute the best local action to reach from s to g.
3. Else, let l_i be the second landmark on the shortest high-level path from $NL_{FROM}(s)$ to $NL_{TO}(g)$.
4. Perform the best local action to reach from s to l_i.

9.2 Empirical Results

We have obtained various interesting pictures of mappings learned by the HDGL and HDGM algorithms in such domains. In the FROM mappings, each state is in the region of the nearest landmark that can be reached *from*

that state, whereas TO mappings place a state in the region of the landmark from which there is an optimal path *to* the state.

Figure 9.1 displays the FROM mappings learned by the DG-HDG algorithm (a version of HDGL that runs DG and HDG in parallel) in a $20X20$ grid-world with walls and membranes during a run of length 30,000. For this run the learning rate was initially 0.4 and eventually decayed to 0.3. The transition probability parameter of the domain was set at 0.2. Each little grid in that figure represents a state of the world. As indicated in the legend, the states with large black circles in them are the landmarks. All states have the same color and pattern combination as their corresponding nearest landmark (it may be somewhat difficult to discriminate regions if you have a black and white as opposed to a colored copy). Clearly, the learned mappings are very close to optimal, and, the partitions respect membranes and walls.

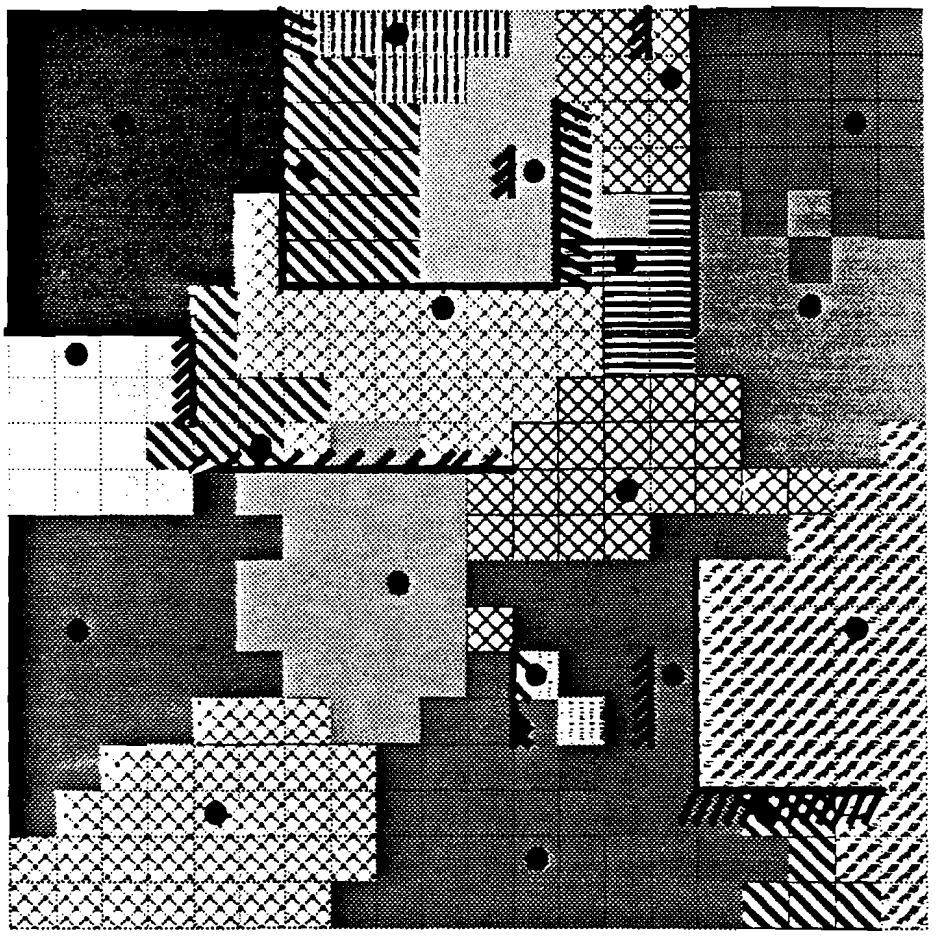

Figure 1

Maze or wall

Membrane or one-way wall - the short slanted lines indicate the porous direction (for example, the membrane shown in this sample is porous in the South to North direction, but impenetrable while going from North to South)

Landmark state

Figure 1: Mappings learned by DG-HDG in 20 x 20 grid-world
Run Length: 30,000

Figure 9.1: Mappings Learned by HDGL in a $20X20$ grid-world.

Chapter 10

Multiple Levels of Hierarchy

As mentioned in [7], the hierarchical process can be continued by introducing additional levels of hierarchy.

10.1 The Landmark Network

Given the set S of states and h hierarchical levels, let $L_{h'}$ represent the set of landmark states at level h'. At each hierarchical level h' , where $1 \leq h' \leq h$, we have a tuple $\langle L_{h'}, NL_{h'}, N_{h'} \rangle$. At level 1, $NL_1 : S \mapsto L$ is a mapping from each state to its nearest landmark, and at every other level h', $NL_{h'} : L_{h'-1} \mapsto L_{h'}$ is a mapping from each level $h'-1$ landmark to its corresponding level h' landmark. For each hierarchical level, $N_{h'} : L_{h'} \mapsto 2^{L_{h'}}$ is a mapping from each landmark to a set of neighbor landmarks at the same level.

10.2 The Algorithm

The algorithm with h hierarchical levels, HDG-h, is a straightforward extension of the HDG algorithm with a single level of hierarchy.

10.2.1 Executing Actions

To get from current-state s to goal-state g,

1. let $g_{int} = g$ and $h' = 1$.
2. if $NL_{h'}(s) = NL_{h'}(g_{int})$ or $g_{int} \in N_{h'}(NL_{h'}(s))$ then execute best level $h' - 1$ action to get from s to g_{int}.
3. else while $NL_{h'}(s) \neq NL_{h'}(g_{int})$ and $g_{int} \notin N_{h'}(NL_{h'}(s))$ and $h' \leq h$ $h' = h' + 1$
4. Let g_{int} = the level $h' - 1$ intermediate goal to get from s to g_{int}.
5. goto step 2.

10.2.2 Data Structures

The distance values at the lowest level are maintained exactly as before. At the highest level of hierarchy we maintain Γ values between the L_h landmarks exactly as the Γ values between L landmarks were maintained in HDG. At all intermediate hierarchical levels, the Γ values are maintained analogous to the way the DG values are maintained. That is, for $1 \leq h' < h$, we maintain $\Gamma_{h'}$ values from every landmark $l_i \in L_{h'}$ to every other landmark $l_j \in L_{h'}$ such that $NL_{h'+1}(l_i) = NL_{h'+1}(l_j)$ and to every level $h' + 1$ landmark $l_{h'+1,n}$ such that $l_{h'+1,n} \in N_{h'+1}(NL_{h'+1}(l_i))$.

10.2.3 Learning

The DG values and the Γ values at each hierarchical level are learned incrementally exactly as they were with the HDG algorithm.

10.3 Empirical Results

We have a generic implementation that enables us to specify and construct an arbitrary number of hierarchical levels. Until now we have experimented with one and two levels of hierarchy.

For the experiments discussed in this section the transition parameter of the world was set at 0.5. Policy iteration was run off-line to determine the distance values between states and then those values were used to form the NL and N mappings for bootstrappping the hierarchical algorithms.

Figure 10.1 displays the learning curves for DG, HDG and HDG-2 in a $10X10$ grid-world. For HDG there were 10 randomly selected landmark nodes and for HDG-2 there were the same 10 level 1 landmarks, and 5 of those were hand-picked to be level 2 landmarks. For DG learning, the learning rate was set at 0.75 and then decayed to 0.3. For the hierarchical algorithms, at the lowest level the learning rate was 0.6, and at higher levels of hierarchy the initial learning rate was 0.75. At each level, the learning rate was eventually decayed to 0.3. In these experiments, the performance factor for DG learning averaged over 5 runs was 0.07873 with a standard deviation of 0.0009419162. The average performance factor for HDG was 0.04824 with a standard deviation of 0.0006456659. HDG-2 had an average performance factor of 0.04217 with a standard deviation of 0.001701859. DG learning performed significantly better than the two hierarchical versions but the difference between HDG and HDG-2 was not significant.

We also experimented in a $30X30$ domain. There were 30 landmark states at the first level of hierarchy, and, 10 of those states were also landamrks

at the second level of hierarchy. The partitions induced on the domain with 1 level of hierarchy are illustrated in figure 10.3. The learning rates of all algorithms were maintained constant at 0.4. The learning curves comparing DG, HDG and HDG-2 are plotted in figure 10.2. The performance factor of the optimal learner was 0.0339. DG had a performance factor of 0.023366. HDG had a performance factor of 0.015196 and HDG-2 had a performance factor of 0.010878. It took DG learning a little over 4 hours to finish this run of length 500,000. The same run was completed by HDG in 80 minutes, and, HDG-2 took 60 minutes to finish.

10.4 Computational Complexity

Consider a domain with $|S|$ states partitioned into h hierarchical levels. Let $|S_0|$ be the number of states at the lowest level, and r_0 be an upper-bound on the region-size (i.e., the number of states in each region) at the lowest level. Let $|S_{h'}|$ be the total number of landmark states at level h', and, $r_{h'}$ be an upper-bound on the number of states in each region at level h' (At level h, we will get a single region with $|S_h| = r_h$).

Further assume, for simplicity, that $|D|$ is an upper bound on the number of neighbors that a landmark state can have at each hierarchical level.

Then, in the worst case, executing a single step can cost $O(|A|) + O(h|D|)$, since we require $O(h)$ time to look up the nearest landmarks at each of the hierarchical levels, $O(|D|)$ time to find the next best landmark at each hierarchical level, and $O(|A|)$ time to select an action. Learning can cost $O(|A|(r_0 + |D|) + |D|(r_1 + |D|) + ... + |D|(r_h + |D|))$, since the cost of updating the DG values at the lowest level is $O(|A|(r_0 + |D|))$, and the cost of updating the Γ values at each hierarchical level h' is $O(|D|(r_{h'} + |D|))$. Assuming that $|A|$ and $|D|$ are small, we have worst-case of $O(hr_{largest})$, where $r_{largest}$ is the largest regional size amongst all the levels. To optimize this cost, we must have the same upper bound on regional sizes at each level. Then $r = r_{largest} = r_0 = r_1 ... = r_h$, and we have, $|S| = r^{h+1}$.

If S is of the form k^n, we want $h = n - 1$ with no more than k states in each region at each level to get the optimal cost of $O(nk)$.

If we can empirically determine a suitable region size $r_{largest}$ then we can set $k = r_{largest}$, and, determine $h = \lg_k S$. Alternatively, if we have a pre-

fixed number of hierarchies that we want to implement then, we can find the corresponding region size, $r_{largest} = S^{\frac{1}{1+h}}$.

In really large domains introducing several hierarchical levels will optimize the time requirements. Also, each additional level of hierarchy will add to the sub-optimality of the paths found by the algorithm.

10.5 Discussion

We presented a starightforward extension of the HDG learning algorithm that enables us to implement several levels of hierarchy. Our empirical results in the $10X10$ domain indicate that DG performs better than HDG and HDG performs better than HDG-2 although the difference in performance factors of these algorithms is not that large. Even in the $30X30$ domain the results are similar. Whereas, in the $30X30$ experiment presented in chapter 6 we had obtained learning curves indicating that HDG initially performs better than DG. In these two sets of experiments the run lengths were different, the algorithms had varying learning rates and the transition probability parameters of the domain were different.

Our computational complexity results indicate that it would be very efficient in terms of time and space to implement several hierarchies in larger domains. We indeed discovered that the hierarchical algorithms run substantially faster in the $30X30$ domain.

Our results to date suggest that it is worth implementing several levels of hierarchy in larger domains and exploring how the algorithms perform upon varying the number of landmarks and the learning parameters across hierar-

chical levels in larger domains.

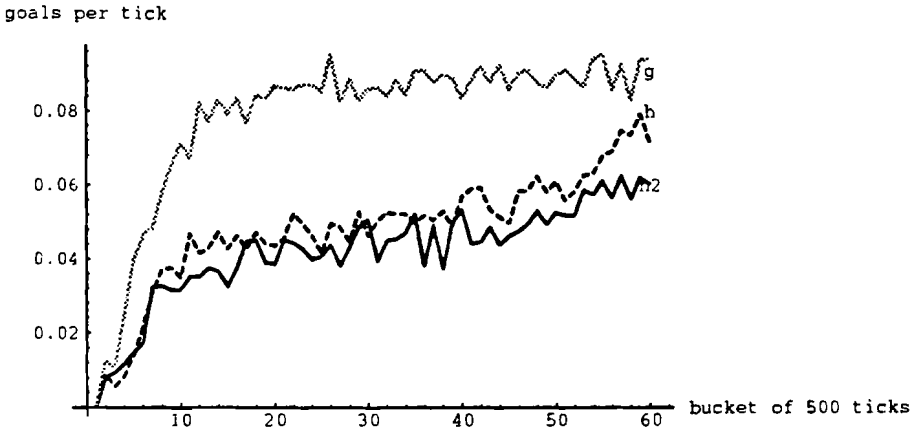

Figure 10.1: The gray line, the dashed line and the black line denote learning curves for DG, HDG and HDG-2 respectively. Each curve is averaged over 5 runs of length 30,000

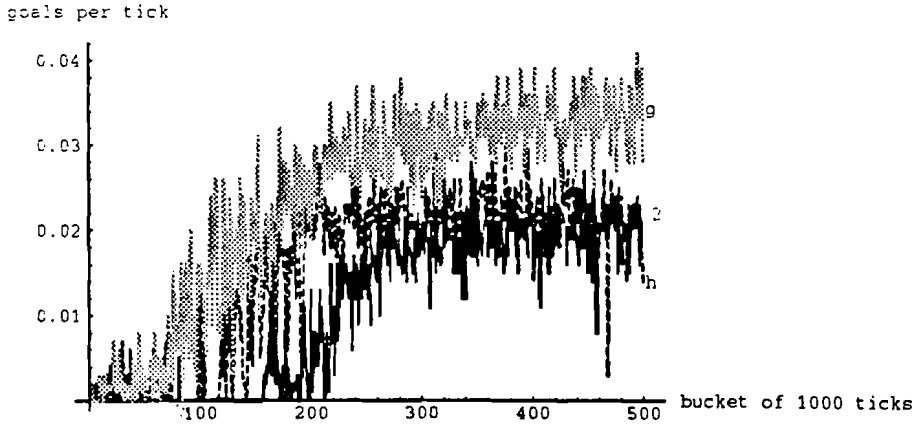

Figure 10.2: The gray line, the dashed line and the black line denote learning curves for DG, HDG and HDG-2 respectively. Each curve is over a single run of length 500,000.

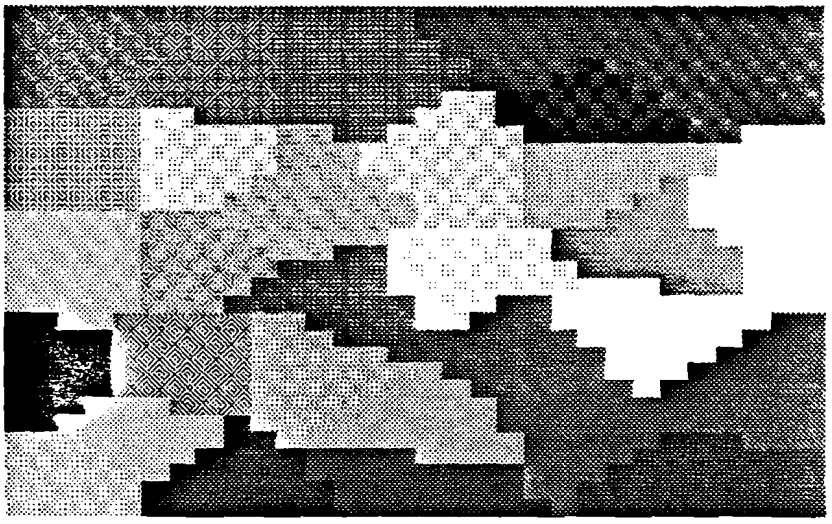

Figure 10.3: A $30X30$ world with 1 level of hierarchy and 30 landmark states.

Chapter 11

Landmark Layout

In our HDG algorithm, we either randomly select the landmark states or hand-pick them so that they are well-distributed over the domain. We must address the question of whether there is such a thing as an optimal landmark layout, that is, a landmark layout that minimizes the additional cost incurred owing to the hierarchical framework.

For simplicity, let us consider a single level of hierarchy. Let k be the total number of landmark states selected. Also, let us assume that we have an oracle that provides us with the correct distance from every state s_1 to every other state s_2.

When the HDG algorithm converges, the distance values found by the algorithm will be given as:

$$D(s_1, s_2) = \begin{cases} \min_{a \in A} DG(s_1, a, s_2) & \text{if } NL(s_1) = NL(s_2) \\ \min_{a \in A} DG(s, a, NL(s)) + \min_{l_i \in L} \Gamma(NL(s), l_i, NL(g)) & \\ \quad + \min_{a \in A} DG(NL(g), a, g) & \text{otherwise} \end{cases}$$

We want to minimize some measure that correlates the above D values with the actual distances d. For example, we could consider minimizing the

sum squared diffrence, that is

$$\sum_{s_1 \in S} \sum_{s_2 \in S} (D(s_1, s_2) - d(s_1, s_2))^2$$

We have so far investigated two different approaches to resolve this problem.

11.1 Optimal Landmark nodes in a planar graph

Kaelbing [7] had suggested that it might be possible to use some recent work on approximation algorithms for all-pairs shortest path problems to provide an upper bound on the suboptimality of paths found by the hierarchical DG algorithms.

We studied an algorithm for maintaining all-pairs shortest-paths in planar graphs [10]. Given k selected nodes all on the boundaries of a constant number of faces in the graph G, and given some $0 < \epsilon <= 1$, the algorithm provided in [10] will find a sparse-substitute graph to approximate all-pairs shortest paths in G to within a $1 + \epsilon$ factor in $O(\epsilon^{-1} n \log^2 n \log D)$ time, where D is the sum of the lengths of all the edges and n is the number of nodes in G. The algorithm mentioned above divides the graph G into several clusters and integrates the partial solutions for different clusters to derive a solution for the entire graph [10]. Note that this approach is very analogous to the approach used by the HDG algorithms, i.e, HDG algorthms impose a partition on the domain, maintain DG values for each region, and use the Γ values to achieve inter-regional travel.

A k-cluster partition can be obtained from a two-connected planar graph G in $O(nlogn)$ time [10]. Essentially a sparse-substitute graph is constructed from G by placing substitute nodes in G. This is analogous to constructing a landmark network by placing landmarks in the world. If we had a domain that satisfied the planarity and two connectivity restrictions, and if we wanted to optimally place k landmark states in such a domain, we could apply the partitioning technique given in [10].

Once we determine the number k, we can select k states/nodes that lie near the boundaries of our domain, select some desired value for ϵ, and construct a face-boundary substitute as shown in [10]. One problem is that [10] assumes that the domain is continuous, and, while constructing a face-boundary substitute, substitue nodes can be placed anywhere in the domain. In our case the substitute nodes (which will become landmark nodes) must be placed only at some state of the world. However, at least in our grid-world domains, for every point that lies within G there will be a corresponding world-state at a distance of at most $\sqrt{2}$ units. Hence given a domain with S states, we can use the above approach to optimally select k landmark states in $O(SlogS)$ time, and then trivially form partitions around those landmarks in $O(kS)$ time.

In order to run the above algorithm to select landmark states, we must have the correct distance values between states available. Hence, there is a boot-strapping problem. We have not as yet implemented this algorithm, but, it might be worth implementing it, so that it can constitute a benchmark while comparing the performance of other partitioning and alndamrk layout methods.

Also, if the domain is planar and we use this approach to find partitions then we get an upper bound on the path degradation incurred by introducing hierarchies.

Note that the above algorithm approximates all-pairs shortest paths in G to within a $1+\epsilon$ factor. Hence even if Zorbi were to travel in the hierarchical world by rigidly following the landmark graph the paths would be approximated to within a $1+\epsilon$ factor. But, recall that Zorbi need only aim towards nearest landmarks and not necessarily reach them. Hence if the graph algorithm is used to select the landmark nodes then when the DG and Γ values converge all paths will be approximated to within a $1+\epsilon$ factor.

11.2 Incrementally adjusting landmark layout

We have been investigating other incremental approaches for adjusting the landmark nodes. The motivation for this is that we might initially start with randomly selected landmark states and approximate distance values, as learning progresses, at intermediate stages we can incorporate the additional knowledge acquired and adjust the positions of the landmark states.

11.2.1 Kohonen Maps

Kohonen maps are self-adjusting feature detectors [5]. As explained in [5], given a collection of points in the input space with neighborhood relations specified on them, the Kohonen algorithm adjusts these units (achieving an effect analogous to the Mexican hat lateral interconnections) such that they

converge towards optimally modeling the entire input space.

Kohonen maps seem to be a promising approach to learn the landmark layout. We can initially start with random or handpicked landmark nodes. The number of landmark nodes at each level must remain fixed. At each level of hierarchy we have a constant number of landmark indices and each index points to a landmark state in the world. We introduce the landmark-position mapping that maps landmark indices to their corresponding world states, $LP : LI \mapsto S$. The state corresponding to a certain landmark index may change with time The landmark indices form the nodes of the Kohonen map, and each node has the appropriate position attributes mapping it to its corresponding world state.

Let $Position(l_i) = LP(l_i)\ \forall\ l_i \in LI$.

The landmark-position mapping maps landmark indices to their corresponding world states, $LP : LI \mapsto S$. We have, $Position(l_i) = LP(l_i)\ \forall\ l_i \in LI$.

We have the new-landmark-position mapping that maps landmark states to new positions that approximate some world state, but, may not precisely correspond to a world state. Initially, $\forall l_i \in LI$, $NewPosition(l_i) = Position(l_i)$.

As learning progresses,

1. at each step
   (a) the learning agent performs some action that causes the world to transition to state s
   (b) look up $NL(s) = l_s$ and its index l_{si}
   (c) Pull l_s towards s. $NewPosition(l_{si}) := (1 - \alpha)NewPosition(l_{si}) + \alpha Position(s)$
   (d) Pull
   all the neighbors of l_s towards s $\forall l_{sn} \in N(l_s) NewPosition(l_{sn}) = (1 - \beta)NewPosition(l_{sn}) + \beta Position(s)$
2. at each partition update interval
   (a) for each landmark index l_i, find a worldstate s_{li} that is nearest to $NewPosition(l_i)$, and let $Position(l_i) = NewPosition(l_i) = s_{li}$.
   (b) Map each state to its corresponding nearest landmark and hence update the NL mappings.
   (c) Update the N mappings.

Since the Kohonen map would learn incrementally, it would only require a constant amount of additional time per step. At fixed intervals, we would need $O(SD + L^2)$ time to update the partitions, and this would be amortized over

a moderately large number of steps. If there are certain states in the domain that are frequently visited, then the landmarks will be drawn towards those states, and hence the Kohonen maps would adapt to achieve a domain-specific landmark layout.

We expect to implement this approach and obtain experimental results in the near future.

11.3 Discussion

We motivated reasons for wanting the algorithm to incrementally learn optimal landmark layouts and sketched some thoughts on how to achieve this. The Kohonen map approach should be implemented and tested out in the future. If it is successfull, we should introduce additional complexity in the domain by non-uniformly selecting gola-state (that is, frequently picking states from certain regions of the domain), and by dynamically placing and removing walls and membranes (that is introducing walls at certain places for a reasonably large interval and the removing the wall etc.).

Chapter 12

Related Work

In this chapter, we review some approaches from literature dealing with hierarchical learning and navigation problems. Whenever possible we discuss how these approaches are similar or differ from our algorithms.

12.1 Feudal Reinforcement Learning

Dayan and Hinton [4] present a feudal reinforcement learning architecture wherein they have a Q learning managerial hierarchy composed of managers, super-managers, sub-managers, sub-sub-managers etc. There is a strict hierarchical division in their system, for example, a sub-manager is merely concerned with satisfying the commands of its manager and is oblivious to the wishes of the super-manager; a sub-manager will be rewarded if and only if it satisfies the command of its manager regardless of whether or not the sub-manager's actions were beneficial to the highest level task. Also, managers only need to know the state of the system at their own level of granularity.

As illustrated by the results in [4], this model of learning works well in a grid-world domain similar to the domains in which we have simulated our HDG

algorithms. However, unlike our domains, the domain in the feudal learning experiments [4] is deterministic. Since a low-level manager concentrates on satisfying the wishes of its immediate superior and is unaware of the global goal it is very crucial that a low-level manager always receive orders from the appropriate superior. In a stochastic domain this might be difficult to ensure. Also, a manager is rewarded by its superior whenever the action performed by the manager is consistent with the superior's orders. But, in stochastic domains it is possible that such a consistent action results in a consequence that is inconsistent with the superior's order.

In the maze task illustrated in [4], managers are assigned to separable portions of the maze at each level. As noted in [4], while the separation into quarters at the various levels is fairly arbitrary, the system would not perform well if the regions at the high levels did not cover contiguous areas at the lower levels. Hence this approach demands a plausible managerial system preferably based on some domain-dictated partitioning of the state space. Providing such a hierarchical partitioning to the system is analogous to providing hand-coded partitions to the HDG learning algorithm.

12.2 Reinforcement Learning with a hierarchy of abstract models

Singh presents the H-DYNA architecture [15] that learns a hierarchy of abstract models to solve composite sequential tasks. As Kaelbling notes [7], since the tasks are sequntial a natural task decomposition is introduced. Hence while splitting up at hierarchical levels this algorithm does not have a boot-strapping

problem like the HDG or feudal RL algorithms.

12.3 Parti-game Algorithm

The parti-game algorithm induces partitions on the domain using geometric techniques, explicitly retains records of partition, system state, action, and outcome tuples, and, uses a game-theoretic approach to achieve goals via inter-partition travel [11]. This algorithm is applicable for goal-of-achievement tasks in deterministic domains.

The parti-game algorithm induces partitions on the state space using geometric techniques. There is no notion of landmark states with this algorithm, but, given a partition with N neighbors the learning agent can perform N discrete actions to traverse across partitions. During such traversals the agent aims towards the center of the partition that it wants to reach. This is very akin to Zorbi aiming towards the next landmark during inter-regional travel with the HDG algorithm.

A neat feature of this algorithm is that it dynamically and reactively adjusts the partitions such that a high resolution is achieved in relevant areas of the domain. We expect and hope that when Kohonen maps are incorporated into the HDGM algorithm so that the landmarks can be concentrated in critical sections of the domain and the partitions can be suitably modified, a similar behavior will result.

12.4 Ariadne's Clew Algorithm

The Ariadne's Clew algorithm builds a global path planner using two local components called a SEARCH algorithm and an EXPLORE algorithm [12]. The EXPLORE algorithm helps collect information about the environment by placing *landamrks* in the searched space. The SEARCH algorithm is based on genetic programming and determines whether the goal can be reached from any of the placed landmarks. While this approach is rather different than the reinforcement learning algorithms discussed in this thesis, it does achieve the goal of building a path planner for a robot in a dynamic environment, and it does use a landmark-based approach, hence it is worth reviewing in this chapter.

Note that unlike our reinforcement learning algorithms, Ariadne's Clew algorithms have access to a lot of domain-dependent information. The SEARCH algorithm uses a backprojerction method and a genetic algorithm to determine whether a "simple" path exists, i.e., whether some goal target may be reached by performing a few elementary motions from a given starting position [12]. This algorithm may be used by itself, but, it might get stuck in local minima.

Adding the EXPLORE component makes the algorithm complete. This component places landmarks in the searched space. A new landmark is placed such that there is a known path from the initial position to that landmark, and, it mainatins the property that all landmarks are placed as far from each other as possible [12]. Also, the number of new landmarks placed depends on the complexity of the domain.

The Ariadne's Clew algorithm, as described in [12], first uses the SEARCH

algorithm to find a "simple" path from the initial position to the goal position. If no such path is found then until such a path can be found the EXPLORE component places a new landmark, and the SEARCH component looks for a simple path.

Chapter 13

Conclusion

We will briefly summarize the work presented in this thesis and conclude by sketching some thoughts for future work.

13.1 Summary and Future Work

We started with an introduction to reinforcement learning, reviewed the Q learning and DG learning algorithms and discussed replicated empirical results comparing these two algorithms on goal-of-achievement tasks. Then we explained the motivations for having hierarchical learning algorithms, reproduced Kaelbling's HDG algorithm and discussed replicated results comparing HDG and DG learning in simulated stochastic domains. We then discussed in detail the various aspects of HDG that we have modified in the course of this work in order to make the algorithm more robust, and, presented experimental results comparing these newer versions of HDG with the older version or with themselves and computational complexity results for the newer versions.

We presented the HDG-Incremental algorithm to learn Γ values between landmarks and presented empirical results and computational complexity fac-

tors determining that this version of the algorithm is preferable over HDDG-Floyd Warshall.

We proposed the UE method of action selection. Our results indicated that DG-UE out-performs DG-Boltzmann in a $10X10$ domain whereas there is no significant difference between the performance factors of HDG-UE and HDG-Boltzmann. We speculate that tweaking the UE parameters might enhance the performance of the hierarchical algorithm.

We implemented programs enabling us to generate arbitrary numbers of hierarchical levels. We presented empirical results comparing DG, HDG and HDG-2 algorithms in $10X10$, $20X20$ and $30X30$ domains. Our results in chapter 6 (over relatively shorter runs) indicated that HDG initially outperforms DG in larger domains. Whereas results in chapter 10 (over very large runs) indicated that DG outperforms the hierarchical algorithms throughout the run. We confirmed that the hierarchical algorithms run much faster as compared with simple DG in larger domains. Our results to date are encouraging but additional experiments should be performed with several levels of hierarchy in larger domains before we can reach any conclusions regarding the hierarchical algorithms. Can we theoretically quantify the efficiency achieved vs. the path degradation incurred by the hierarchical algorithms? Towards the end of section 11.1 we alluded to one method for providing an upper bound on the paths traversed by the hierarchical algorithms as they converge to optimal DG and Γ values. Can a tighter bound be discovered?

We introduced walls and one-way walls or membranes to make our domains more interesting. At present our walls are absolutely impermeable. Potentially

we could make them semi-porous.

Ideally, our hierarchical algorithms should learn to induce partitions on the domain given little or no domain-specific information. To achieve this goal we proposed the HDGL and HDGM algorithms. Empirical results obtained with HDGM (when bootstrapped with reasonable partitions) in a $10X10$ domain with walls are very promising. This suggests that future work should include experiments with this algorithm in huge complex domains with several levels of hierarchy. In several real-world situations, it is possible to obtain approximate information about the domian, and, also, a lot of doamin-specific information is likely to dynamically change. Since our preliminary experimental results do indicate that starting HDGM with reasonably good approximate partitions enhances performance, we expect HDGM to be very robust and to perform well in such scenarios.

Kohonen maps can be implemented to produce a landmark layout that dynamically adapts to generate optimal (we hope) landmark networks. It should be interesting to integrate the Kohonen map method of landmark placement with the HDGM algorithm for modifying partitions.

It will be really interesting to implement the hierarchical algorithms on a real-robot. In order to embark on such an implementation, it might be necessary to integrate HDG at a higher level of abstraction with an approach such as neural networks or hand-coded reactive layers at a lower level.

In the long term, it would be interesting to implement the HDG algorithms on a coarse-grained parallel machine. Each cluster of such a machine can perform DG learning, and there can be landmark clusters that have more

computational power and maintain the Γ values.

When we start thinking of parallel implementations, it is easy to imagine a team of robots working in parallel. Each of these robots can primarily reside in a certain region and perform DG learning, and, some mechanism should enable robots from neighboring regions to communicate and thereby maintain Γ values. In such a scenario, while delivering objects for example, for inter-regional travel, a robot from a certain region would head towards a neighboring region, and the robot from that neighboring region would meet it at the boundary so that they can exchange the package being delivered.

13.2 Perspective

Our dream is to create autonomous and adaptive learning agents that will accomplish dynamically changing goals in stochastic domains.

We realize that learning in huge domains can incur huge time and space constraints and we attempt to resolve these problems using hierarchical constructs. At this stage we have obtained encouraging results with the hierarchical algorithms. Work from other disciplines studying human learning also lends support to the concept of hierarchical learning. Research from Cognitive Science supports the conclusion that people can significantly improve their memory upon hierarchically organizing large amounts of information [1]. Empirical results on simulations of olfactory paleocortex layers of the brain indicate that the learned cues are organized in a hierarchical pattern in that section of the brain [6].

This thesis has extended Kaelbling's hierarchical learning work and pre-

sented interesting new results. As mentioned in the previous section there is tremendous scope for expanding this work.

Bibliography

[1] Lawrence W. Barsalou. *COGNITIVE PSYCHOLOGY An Overview for Cognitive Scientists.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1992.

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

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

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

[5] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, Redwood City, California, 1991.

[6] Gary Lynch Jose Ambros-Ingerson, Richard Granger. Simulation of paleocortex performs hierarchical clustering. *Science*, 247:1344–1348, 1990.

[7] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, Massachusetts, 1993. Morgan Kaufmann.

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

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

[10] Philip Klein and Sairam Subramanian. A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs. *WADS proceedings*, 1993.

[11] Andrew W. Moore. The parti-game algorithm for variable resolution. In *Proceedings of the NIPS 93*, 1993. submitted to NIPS 93.

[12] El-Ghazali Talbi Pierre Bessiere, Juan-Manuel Ahuactzin and Emmanuel Mazer. The "ariadne's clew" algorithm: global planning with local methods.

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

[14] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, Massachusetts, 1993. Morgan Kaufmann.

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

[16] Barbara Tversky. Distortions in cognitive maps. In *Geoforum*, volume 23. Permagon Press Limited, Great Britain, 1992.

[17] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.