AN ABSTRACT OF THE THESIS OF

Sriraam Natarajan for the degree of Master of Science in Computer Science presented on June 2, 2004.

Title: Multi-Criteria Average Reward Reinforcement Learning

Abstract approved: Redacted for Privacy

Prasad Tadepalli

Reinforcement learning (RL) is the study of systems that learn from interaction with their environment. The current framework of Reinforcement Learning is based on receiving scalar rewards, which the agent aims to maximize. But in many real world situations, tradeoffs must be made among multiple objectives. This necessitates the use of vector representation of values and rewards and the use of weights to represent the importance of different objectives.

In this thesis, we consider the problem of learning in the presence of time-varying preferences among multiple objectives. Learning a new policy for every possible weight vector is wasteful. Instead we propose a method that allows us store a finite number of policies, choose an appropriate policy for any weight vector and improve upon it. The idea is that though there can be infinitely many weight vectors, a lot of them will have the same optimal policy. We prove this empirically in two domains: a version of the Buridan's ass problem and network routing. We show that while learning is required for the first few weight vectors, later the agent would settle for an already learnt policy and thus would converge very quickly.

Multi-Criteria Average Reward Reinforcement Learning

by

Sriraam Natarajan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 2, 2004
Commencement June 2005

Master of Science thesis of Sriraam Natarajan presented on June 2, 2004

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Associate Director of the School of Electrical Engineering and Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Sriraam Natarajan, Author

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major professor, Dr.Prasad Tadepalli for his patience, guidance, encouragement, and support during my graduate study. I am indebted to him for funding me for over a year while I was working on this thesis.

I would like to thank Dr.Saurabh Sethia, Dr.Thomas G. Dietterich and Dr. John Bolte for sparing their valuable time and being on my committee. I would also like to thank the CS office staff for their help over the past 3 years.

I am grateful to OPNET Technologies Inc. for providing me with their simulator and support.

My special thanks to my family and friends for their love and support.

This material is based upon work supported by the National Science Foundation under Grant No. 0329278. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. I would also like to acknowledge the support of Defense Advanced Research Projects Agency under grant number HR0011-04-1-0005.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

MULTI-CRITERIA AVERAGE REWARD REINFORCEMENT LEARNING

CHAPTER 1

INTRODUCTION

1.1 Outline

One of the harder problems in the field of artificial intelligence is the design of autonomous agents. These agents must typically interact with a dynamic environment, i.e., perceive the environment and take actions that achieve their goals. A truly intelligent system must learn from its environment.

An autonomous learning agent must interact in its own way with the environment, receive perceptions and take its own actions and learn from its experience what action it needs to take in each situation. Learning from environment is a complex issue. The agent must explore the environment to get a clear picture about the environment and about the effects of its actions [19]. It must use knowledge of these effects to make decisions in the future. If the agent is tracking the model of the environment, it should update the model with every action that it takes.

Reinforcement learning is the process by which the agent learns the correct behavior in an environment through trial and error interactions with the environment. The agent is not told explicitly what actions to take; instead the agent must determine the most useful actions by executing them. Each action would

yield some reward and the agent must try to maximize the rewards. There are many optimization criteria including, the total reward optimization, discounted reward optimization and the average reward optimization. In this work, we consider the average reward optimization, which aims to optimize the expected average reward per time step over an infinite horizon.

Traditional Reinforcement Learning techniques are essentially scalar based, i.e., they aim to optimize a particular criterion that is expressed as a function of a scalar reinforcement. In many real world domains, however, the aim may not be to optimize a single criterion. For example consider the Buridan's ass problem. There is a donkey at equi-distance from two piles of food. It is hungry and so wants to move to one of the piles. The problem is that if it moves towards one of the piles, the food in the other pile can be stolen. So, if the donkey is very greedy, then it would stay at the center and would eventually die. If it chooses to move to one of the piles every time it is hungry, it would lose a lot of food.

So it is clear that there are two goals here. The agent has to find a reasonable compromise in this case. One such compromise would be to minimize the number of piles of food stolen per unit time while satisfying its hunger. The other more reasonable compromise would be to maximize a weighted sum of piles guarded and piles eaten. Here the weights represent the importance of each goal, maximizing the intake of food, and minimizing the amount of food stolen. Both these cases present the need of vector-valued representations in Reinforcement Learning. We consider the vector based reinforcement learning techniques.

Consider the fact that these weights can vary with time. One day the donkey may be famished and so its aim would be to eat as much as possible and

then guard the food. Other times, it could wait for sometime till it becomes hungry and then could eat. So the importance of the criteria could vary in an unpredicatble fashion. If the agent is going to learn from scratch for every weight vector (a weight vector represents the importance of different criteria), then learning would be too slow. Instead, the agent could use the best policy learnt so far for that weight and could begin learning from it. The intuition is that this would need lesser amount of learning as it already has a good policy. Another idea is that after a certain number of policies the agent learns, it need not learn any more policies. This is to say that though there can be an infinite number of weight vectors, there are only a finite number of policies that the agent needs to learn.

1.2 Organization of the thesis

Chapter 2 provides the background of this thesis. It introduces Reinforcement Learning and Markov Decision Processes. It compares in brief the 3 different optimization techniques: Total reward, discounted reward, and average reward optimization techniques. It then gives an overview of model-based and model-free Reinforcement Learning agents. It then presents the two already existing average reward reinforcement learning techniques: R-Learning and H-Learning.

Chapter 3 begins with the motivation of the idea of vector based reinforcement learning. The previous work is discussed, followed by the idea of learning from prior polices. The multi-criteria reinforcement learning algorithm for learning from previous policies is presented. Then the model-based and model-free versions of the algorithms are explained.

Chapter 4 presents the results of applying our algorithm to 2 domains: a toy

domain and a real-world domain. Initially, the modified version of Buridan's problem is shown. Then the implementation details of the domain are provided. Finally, the results of the implementation of the Multi-Criteria versions of R and H learning algorithms are presented.

The second half of chapter 4 deals with the second domain: Network Routing. This section provides a background on network routing and an overview of the hierarchy in OPNET and presents the three models that we created, the network model, the node model and the process model. Then the implementation details are provided. As with the previous section, the results of the implementation of the two algorithms are presented.

Chapter 5 concludes this thesis. It discusses the results and our contribution. It then outlines some areas for future research.

CHAPTER 2

BACKGROUND

This chapter outlines the background of Reinforcement Learning and also provides an insight into Markov Decision Processes. We also present in brief the ideas of discounted and average reward reinforcement learning methods. We then summarize the ideas of model-based and model-free reinforcement learning methods.

2.1 Reinforcement Learning

Reinforcement Learning is learning what to do [16]. Reinforcement Learning agents learn to act in an environment through trial and error. As can be seen from Figure 2.1, an agent interacts with the world (environment) through actions and percepts and receives rewards or penalties (reinforcements) for the same.

The reinforcement learning agents are not told which actions to take. They would instead determine the best actions by executing them. For each action that the agent executes, the environment responds by giving reinforcements (rewards and penalties). The agent, upon receiving the reinforcements, incrementally learns value functions for states or state-action pairs. The value function for a state action pair is the best long term "value" of taking that action in that state.

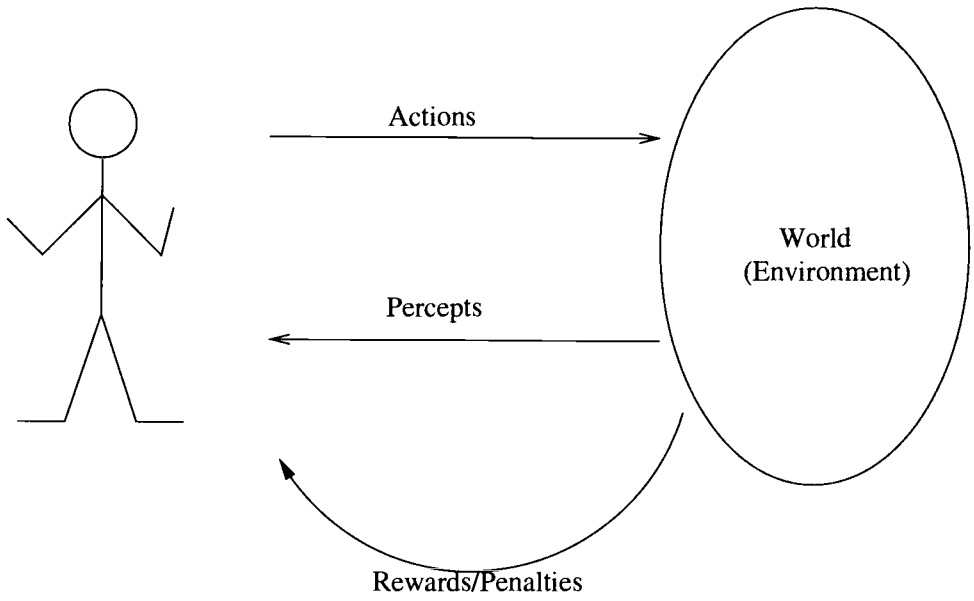

FIGURE 2.1: Interaction of an agent with the environment

Reinforcement learning is useful in domains where explicit supervision from a teacher is unavailable [13]. For example, in many gaming environments, it is virtually impossible for a teacher to provide accurate information about positions of other agents. So in these cases, the agent should know whether it has won or not and use that to learn a function that gives a reasonable idea about the chances of winning. In some cases, there can be an immediate feedback that could let the agent know how it is doing without waiting till the end of the game. This would help the agent learn the domain faster [13].

The goal of the reinforcement learning agent is then to determine a policy, i.e., a mapping from states to actions, by executing actions and receiving reinforcements. This policy should aim to maximize the "returns" or some measure of the returns that are accumulated over time. There are different ways of measuring these returns which are explained in section 2.3.

2.2 Markov Decision Process

In reinforcement learning, during each step of interaction with the environment, the agent perceives the current state of the environment. The decisions are made as a function of the state. A state signal that retains all the relevant information is said to satisfy the Markov property. A task that satisfies the Markov property is called as Markov Decision Process (MDP). An MDP is described by:

- A set of discrete states S
- A set of actions A
- A reward function $r_i(u)$ that describes the reward of action u in state i
- A state transition function $p_{ij}(u)$ that describes the transition probability from state i to state j under action u

The state transition function specifies the next state as a function of the current state and action. The set of actions that are applicable in a state are called admissible actions. The actions are stochastic and Markovian, i.e., there is a certain fixed probability $p_{ij}(u)$ that in state i, upon executing the action u, the next state would be j. The state transitions are independent of the previous environment states or actions. So these models are Markov models. Also the reward function specifies the immediate reward as a function of the current state and action.

Time is modeled as a discrete sequence of steps. A policy is defined as a mapping from states to actions, i.e., a policy specifies what action to execute in each state. A deterministic policy is one that prescribes the same action in

the same state. An optimal policy is one that maximizes the expected long-term returns from every state. The value of a state under a particular policy π is denoted by $V^{\pi}(s)$, which is the expected returns starting from state s and following the policy π.

In many versions of reinforcement learning, instead of learning the policy directly, the agent learns the value function [4]. So for choosing the best action in each state, the values of the next states are predicted using some value function. An action that is chosen to maximize the value of the next state, as predicted by the value function plus any expected immediate reward, is called a greedy action. In addition to greedy actions, the agent also takes exploratory actions. The need for the exploratory actions lies in the fact that they enable the agent to learn new information that might improve its policy. If sufficient exploration is allowed, then all the states would be visited and the agent would have a true estimate of the values of the different states in the environment.

2.3 Optimization schemes

In this section, we provide a brief overview of three kinds of reward optimizations: total reward optimization, discounted reward optimization and average reward optimization.

2.3.1 Total Reward Optimization

As the name indicates, the value of a state for a policy ($V^{\pi}(s)$) is the sum of the rewards that would be received when starting from state s and executing

the policy π, i.e.,

$$R^{\pi}(s) = \sum_{k=0}^{t-1}(r_{s_k}(\pi(s_k)))$$

where s_k is the state at time k.

This approach would make sense in problems where there is a final state or absorbing state, upon reaching which the agent remains there forever. But this is not the case in many domains. There are a lot of cases when the agent's interaction with the environment does not stop, but continues without any limit. This is called as an infinite horizon MDP, where there is no terminal state and the t in the above equation would approach ∞. In such cases the values of states also approach infinity and this criterion would not be appropriate.

2.3.2 Discounted Total Reward Optimization

The problem with the earlier method was that the value functions could approach infinity. Discounted optimization is one way to make this value finite. The discounted total reward of a state is given by

$$R^{\pi}(s) = \sum_{k=0}^{t-1}(\gamma^k r_{s_k}(\pi(s_k)))$$

where $\gamma < 1$, is a discounting factor. As can be seen, the rewards that are obtained immediately are given more importance when compared to rewards that are obtained after a long time. The optimal value of a state is given by the following Bellman equation,

$$V^*(s) = max_a\{r_s(a) + \gamma \sum_{s'} P_{ss'}(a) \cdot V^*(s')\}$$

where s is the current state, a is the action to be executed in the current state and $r_s(a)$ is the immediate reward obtained on executing the action a in the current state.

As had been mentioned earlier, this method gives importance to immediate rewards at the expense of future rewards. Unfortunately, this is not appropriate or justified in most domains[9].

2.3.3 Average Reward Optimization

This method, as the name indicates, aims to optimize the average reward per time step computed as t $\longrightarrow \infty$. This approach would eliminate the problem of giving undue importance to the immediate rewards and considers all the rewards as important.

$$R^{\pi}(s) = \lim_{t \to \infty} \frac{1}{t} \sum_{k=0}^{t-1} r_{s_k}(\pi(s_k))$$

where r_{s_k} refers to the reward obtained in state s at time k and $R^{\pi}(s)$ is the value of state s under policy π. Using the same notation as the previous section, the Bellman equation for the optimal value of a state is

$$V^*(s) = max_a\{r_s(a) + \sum_{s'} P^a_{ss'} V^*(s')\} - \rho^*$$

A gain-optimal policy π^* is a stationary policy that maximizes the average reward for all the states. In this thesis, we consider the average-reward optimization via reinforcement learning methods.

2.4 Model Based Vs Model Free methods

Model-Based Reinforcement Learning (shown in Figure 2.2) refers to learning the transition probabilities and the reward models from experience and using them to learn optimal policies [4]. Basically the agent believes that the models that it learns are the true models. This is called "the certainty equivalence

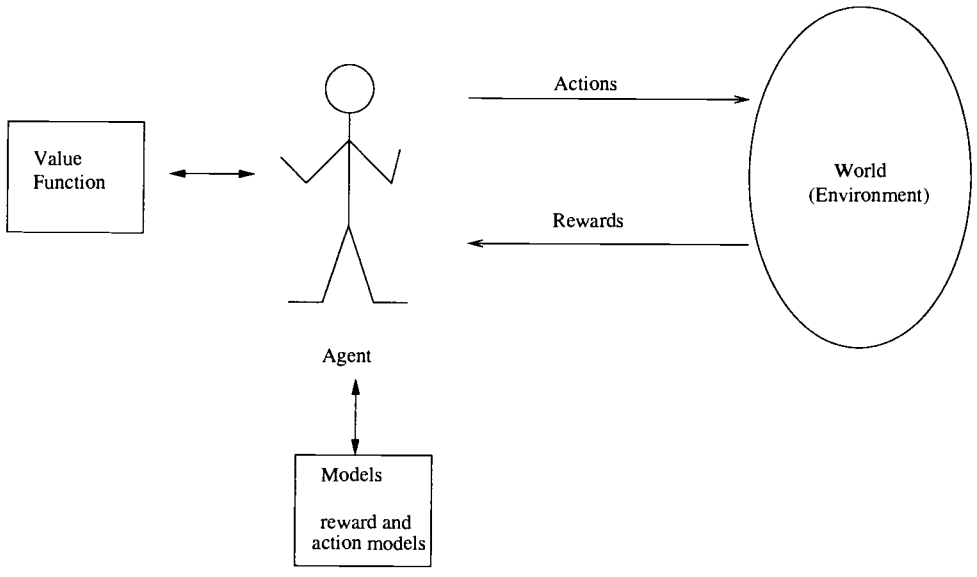

FIGURE 2.2: Schematic of Model based Learning

principle". However, it is easy to see that in the beginning, the models will not even be close to the true model since they are based on limited experience. So if the agent is going to follow some greedy policy, it might converge to a sub-optimal policy and thus could settle for a local maximum. To overcome this problem, exploration must be employed. Random exploration on the other hand could be too inefficient as the agent may take exponentially longer time to converge to a good policy. So, in most cases, an ϵ-greedy strategy is employed [4]. The idea behind the ϵ-greedy strategy is that the agent would take an exploratory action with a probability of ϵ, and would execute a greedy action with a probability of $1 - \epsilon$. The main advantage of the model-based agent is the fact that, since it learns the transition probabilities, it can generalize more rapidly and converge quickly.

The model-free learning agent is shown in Figure 2.3. This type of reinforcement learning is primarily concerned with the goal of learning an optimal policy

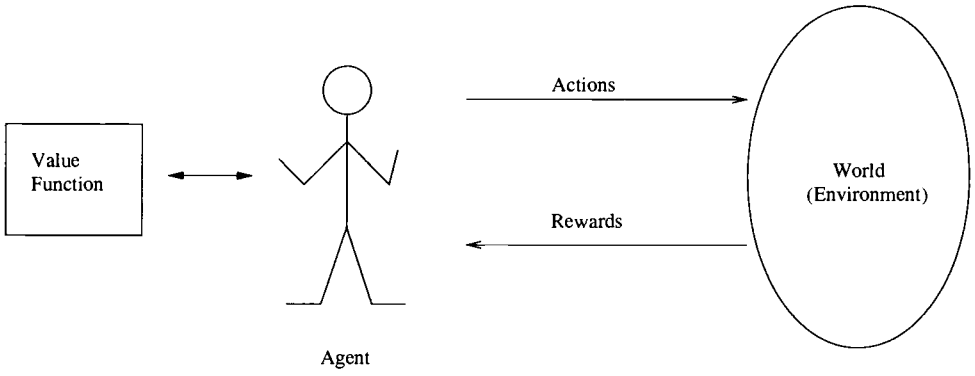

FIGURE 2.3: Schematic of Model Free Learning

without the model knowledge which is mainly the transition probabilities and the immediate reward function. Since the agent does not learn a model of the environment, it learns the values of state-action pairs instead of just the values of states. The idea is that when the values of the state-action pairs are nearly converged to their optimal values, it is appropriate for the agent to take an action with the highest value for each state[8]. Hence while updating the value of an action in a state, the maximum value of the next state over all actions admissible in that state is chosen to be the value of the next state. We will discuss a model-free average-reward reinforcement learning method in the next section.

2.5 Model Free Average Reward Reinforcement Learning - R Learning

Schwartz proposed R-Learning that uses the action-value representation [9]. It is the average-reward version of Q-Learning. The action value $R^{\pi}(s, a)$ represents the value of executing an action a in state s and then following the policy π. We have to associate values with state-action pairs as we do not learn the model

explicitly in this version of reinforcement learning.

If we consider starting from any state, the long run average reward remains constant, but there is a transient [16]. What this means is that there are a few states that yield better than the average reward for a brief period of time and some states that may yield rewards less than the average reward for some time [16]. This expected value of the difference between the total rewards starting from different states, over the infinite horizon is called the bias of a state(or the bias value of the state-action pair).

$$R^{\pi}(s,a) = \sum_{k=1}^{\infty} E_{\pi}\{r_{s_k}(\pi(s_k)) - \rho^{\pi}\}$$

These values are essentially the *relative values*, since they are relative to the average reward. Let us assume that the agent chooses action a in state s. Let s' be the next state and r_{imm} be the immediate reward obtained. The Bellman equation is:

$$V^*(s) = max_a(r_s(a) - \rho + V^*(s'))$$

and hence, the update equation for R-Learning is:

$$R(s,a) = R(s,a)(1-\beta) + \beta(r_{imm} - \rho + max_{a'}R(s',a'))$$

The basic algorithm is shown in Table 2.1. As can be seen, in a particular state, the agent chooses an action that has the maximum R-value for that state action pair(or a random action) and executes it. It receives the immediate reward and reaches the next state. Then it updates the R-values of the previous state-action pair. Also, it updates the average reward if the action executed was an optimal one.

In this algorithm, β is the learning rate for R-values and α is the learning rate for ρ. This is basically used to control the speed of correction of the error. As can be seen, α is updated only if the action is non-exploratory [15].

TABLE 2.1: R-Learning

Initialize the values and rewards

Repeat,

1. Let the current state be s
2. Choose an action a that has the maximum $R(s,a)$ value or choose an exploratory action
3. Execute the action. Let the next state be s' and the reward be r_{imm}.
4. $R(s,a) = R(s,a)(1-\beta) + \beta(r_{imm} - \rho + max_{a'}R(s',a'))$
5. If a is an optimal action,

$$\rho = \rho(1-\alpha) + \alpha[r_{imm} + max_{a'}R(s',a') - max_a R(s,a)]$$

6. $\alpha \longleftarrow \frac{\alpha}{\alpha+1}; s \longleftarrow s'$

There is no proof of convergence of R-Learning, but it is experimentally found to converge to optimal policies on reasonably sized problems with sufficient exploration.

2.6 Model Based Average Reward Reinforcement Learning - H Learning

H-Learning introduced by Tadepalli and Ok [17], is the model based version of R-Learning. The models that are learnt are the transition probabilities and the reward models. The reward model is the average immediate reward that is obtained in a state s upon executing an action a and is represented by $r_s(a)$. The transition probability $p_{ss'}(a)$ represents the probability that the next state is s' given the current state is s and the action is a. This method also uses an $\epsilon-$greedy method to explore the environment.

The bias of the state s here is similar to that of R-Learning and is the expected long-term reward starting from state s over and above ρ^{π}. So the bias must satisfy the equation [4],

$$h(s) = max_a\{r_s(a) - \rho + \sum_{s'=1}^{n} p_{ss'}(a)h(s')\}$$

The idea is that if the agent moves from the state s to the next state s' by executing an action a, it has gained an immediate reward of $r_s(a)$ instead of the average reward ρ. Once the program converges, the expected long-term reward for being in state s relative to being in the next state s' is the difference between $r_s(a)$ and ρ. This forms the basis of the average reward reinforcement learning methods. So the difference between the bias value of state s and the expected bias value of the next state s' is $r_s(a) - \rho$ [17]. Setting the h-value of an arbitrary reference state to 0 guarantees a unique solution for unichain

MDPs [4]. Since only the relative values matter, one method to satisfy the above equations would be to set the h-value of an arbitrarily chosen state to 0 and the resulting equations can be solved. H-Learning estimates ρ from on-line rewards [4].

H-Learning converges faster than the discounted methods due to an important factor. In H-learning while updating the h-value of a state, three values are used: immediate reward, the h-values possible for the next state, and the average reward of the current greedy policy ρ[4]. The discounted methods use only two of these: immediate reward and the values of possible next states. For states without any immediate reward, the discounted methods will have to wait for back-propagation of values, while H-Learning would use the average reward and update the h-values. Using ρ for updates makes the algorithm converge faster.

When the algorithm runs for a large number of steps, it will accurately learn the values of $p_{ss'}(a)$ by visiting all the states and executing all the actions admissible in those states. The estimation of the average reward in H-Learning is similar to that of R-Learning. H-Learning was found to perform better than R-Learning and other RL methods such as Q-learning and ARTDP in many domains [17].

H-Learning is presented in table 2.2. The agent chooses an action a that maximizes the sum of the reward obtained and the h-value of the next state s' (or a random action) and executes the action. It then obtains the immediate reward and reaches the next states'. It updates the models i.e., the transition and the reward models based on the rewards and s'. It also updates the average reward if the action chosen is optimal. Finally, it updates the h-value of the state.

TABLE 2.2: H-Learning

Initialize the values and rewards

Repeat,

1. Let the current state be s

2. Choose an action a such that

$$a = \arg\max_a \{r_s(a) + \sum_{s'=1}^{n} p_{s,s'}(a)h(s')\}$$

   or choose an exploratory action

3. Execute the action. Let the next state be s' and the reward is r_{imm}.

4. $N(s,a) \longleftarrow N(s,a) + 1$; $N(s,a,s') \longleftarrow N(s,a,s') + 1$

5. $p_{s,s'}(a) \longleftarrow N(s,a,s')/N(s,a)$

6. $r_s(a) \longleftarrow r_s(a) + (r_{imm} - r_s(a))/N(s,a)$

7. If a is a greedy action,

   - $\rho \longleftarrow \rho(1-\alpha) + \alpha(r_s(a) - h(s) + h(s'))$
   - $\alpha \longleftarrow \frac{\alpha}{\alpha+1}$

8. $h(s) \longleftarrow max_a(r_s(a) + \sum_{s'=1}^{n} p_{s,s'}(a)h(s')) - \rho$

CHAPTER 3

MULTI-CRITERIA REINFORCEMENT LEARNING

In the previous chapter, the basics of Reinforcement Learning were presented. Also the average-reward and the discounted methods were discussed. We also introduced the idea of model-based and model-free methods. In this section, we explain the idea of Multi-Criteria Reinforcement learning as well as introduce the model-free and model-based multi-criteria average-reward reinforcement learning algorithms.

3.1 Decomposition of values and rewards

Reinforcement Learning (RL) algorithms that we have seen so far utilize scalar valued reinforcements. In many real world situations, it is not possible to express the optimization criteria as maximizing a single scalar-based reinforcement value function. The rewards that are obtained also may not be scalar. One example would be the product delivery domain. There is a warehouse that has to service a few shops. Trucks are used to deliver the products to the shops from the warehouses. Here the goals could be: to make sure that the shops' inventory levels do not become empty and to minimize the transportation costs of the trucks. A government may have to decide between how much to spend on national defense vs stimulating the economy. Also, in a manufacturing plant, there are competing goals: increase the production and reduce the costs.

As another example, consider a modified version of the Buridan's ass problem discussed in Gabor et.al [6] (Figure 3.1). In this example, there is a 3x3 grid. The animal is placed in the center square. Food is present at the two diagonally opposite squares as indicated in the figure. So the animal is equi-distant from the two piles of food. It is hungry, and so it feels like moving towards one of the food piles. But the problem is that if it moves towards one of the piles, the food in the other pile can be stolen. One of the goals is to make sure that the food is not stolen. So it has to somehow compromise between the two goals that are competing with each other.

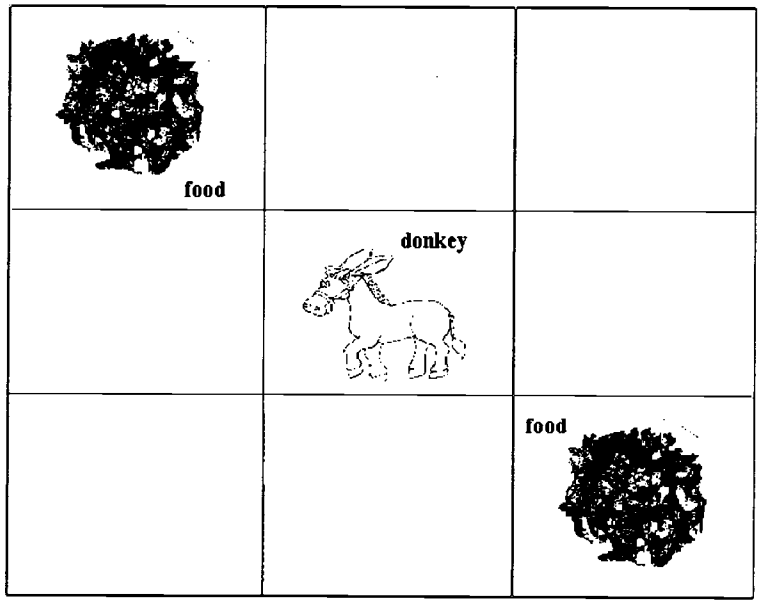

FIGURE 3.1: Buridan's ass problem in a 3 x 3 grid

One reasonable solution would be to do constrained optimization: minimize

the number of piles stolen per unit time while satisfying its hunger [6]. But consider the case where it is famished. Here the priority changes. It has to eat without worrying about the food in the other pile. If the animal is full, then it need not eat and can guard the food. Basically, there are two criteria, one is to satisfy the hunger and the other is to protect the food from being stolen. We introduced the third criterion, which is walking. The animal does not want to spend its energy unnecessarily and seeks to minimize the number of steps it walks.

Since the decision should be based on the amount of food eaten, the amount of food stolen, and the number of steps that the animal has walked, a scalar representation of value functions and rewards is not sufficient. This requires a vector-valued representation of the values, $\overrightarrow{V}$ = (E,S,W). Here E refers to the hunger component, S to the stolen component, and W to the walk component.

The idea is that if the immediate reward is a vector, then the long-term average rewards are also vectors. As explained by Gabor et.al, now the comparison of policies becomes problematic [6]. We must be able to compare any pair of policies, and also we need a reflexive and transitive comparison operator. An optimal policy can now be defined as a policy that compares favorably with any policy.

Gabor, Kalmar and Szepesvari presented a framework based on abstract dynamic programming models [6] and suggested an approach based on the notion of reinforcement-propagating operators. These operators act on function spaces defined over an abstract return space with a given ordering [6]. So, in essence, they consider these problems as constrained problems with lexicographic criteria. But it is not clear how to use this idea when the importance of the different criteria change.

We can also individually compare each component of the reward vector to determine the best policy for that criterion. And now to compare the policies, one way is to sum the two components of the rewards of each policy and compare them. This was proposed by Russell and Zimdars [14]. Their idea was that the overall reward function can be additively decomposed into separate rewards for each sub-agent. In their implementation, the assumption was that there are different sub-agents that aim to solve different goals and there is an arbitrator that combines the sub-agents' recommendations. The combination is done by adding the rewards obtained by each sub-agent. The sum of the different value components is used to determine the best action for a given state. A similar idea is proposed by Guestrin, Koller, and Parr in their paper on Multiagent planning [3]. They view the multiagent system as a single, large MDP, which they represent in a factored way. The action space of the resulting MDP is the joint action space of the entire set of agents [3]. The factored value functions are then solved by a simple linear program.

This approach is similar to Parr's earlier work on policy caches [11]. His idea is to decompose a large stochastic decision problem into smaller pieces. He then proposes two methods: One to build a cache of policies for each sub-problem and then combine the pieces, the other to make the sub-problems communicate with one another. For the former case, he uses a value space search algorithm that is inspired by treating the policies as linear functions, the maximum over which forms a convex surface [7]. All the methods described earlier for solving the multi-criterion problem assume that all the criteria are equally important, which may not always be true. A more general method is to include some weights for computing the weighted average reward for that policy.

Each criterion thus has a weight associated with it that represents the im-

portance of that criterion. For example, if the hunger criteria has a weight of '1' while the other two have a weight of '0', it means that the animal does not have to care about the other two and can concentrate only on eating. So the weights are also vectors $\overrightarrow{W} = (W_1, W_2, W_3)$. Since the rewards are also vectors, the weighted reward can be used for comparing policies. This is more general than assuming that all the criteria are equally important. The optimization criterion would then be a dot product of the weight and reward function vectors. Hence the main idea of Multi-criteria RL are

- The reward and value functions are vectors
- Weights represent the relative importance of different criteria
- The objective function is expressed as a weighted sum of multiple sub-value functions, i.e., $\overrightarrow{W} \cdot \overrightarrow{V}$

We consider the case where the weights themselves are changing. For instance, at 1 PM the animal may be hungry, and so the main objective would be to eat. But at 2 PM, it may be full, and so the goal would be to guard the food. Now the problem becomes a slightly different one. Now there are not only different criteria, but the criteria themselves change. How do we adapt to these changes? Do we learn from scratch for each weight vector? Doesn't that mean that the learning process is redundant and wasteful?

3.2 Learning from prior policies

As has been explained in the previous section, vector representations are imperative for handling multiple criteria. Now the focus is on changing criteria.

As has been pointed out, this is true in many real-world situations. The main question now is: how do we adapt to the changes?

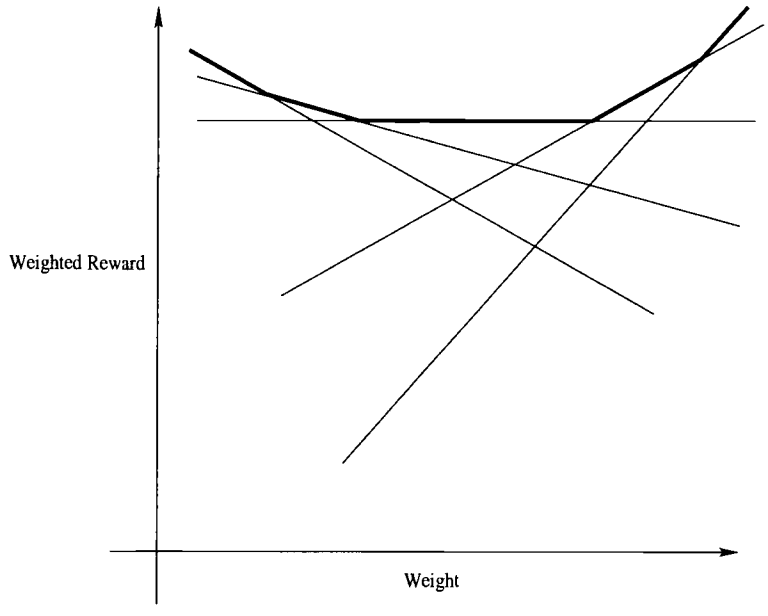

FIGURE 3.2: A few policies for a weight vector with 2 components. The dark lines represent the best policies for any weight.

One solution is to learn the best policy from scratch. Obviously, this is inefficient and does not take advantage of the structure of the problem. Another solution is to learn a policy for a particular weight and use it for all new weights. This is a bad idea, as the two weights may have contrasting goals. For instance, in our grid domain, one weight vector could be $\langle 1, 0, 0 \rangle$ and the new one could be $\langle 0, 1, 0 \rangle$. If we are to use the same policy, then it may not be optimal for atleast one of the weights.

A third solution could be to store all the policies learnt so far with their

rewards obtained and to use one of them (or use the one with the highest average reward). Once again, there is a very good chance that the policy that has the highest average reward may not be the best for the new weight vector. Hence this is not a good solution either.

This problem can be solved by understanding the role of policies in weight space. Let us for simplicity consider a weight vector ($\vec{W}$) with 2 components such that the sum of the individual components would be 1. So one component is enough to describe the weight. Every policy would then have an average reward vector ($\vec{\rho}$) which would also have 2 components. Now if we plot the value of $\vec{W}.\vec{\rho}$ against one component of $\vec{W}$, it would be a straight line. So each policy would generate a line in this weight space (Figure 3.2).

When the weight consists of three components, it can be described by two values (by the simplex constraint, which requires the individual values to sum to 1). Now the weight space can be seen as a triangle in two-space with vertices (0,0),(0,1) and (1,0). Now the $\vec{W}.\vec{\rho}$ is a plane in three space, and the global optimal policy is a bowl-shaped one. The optimal policy (global) is highlighted in black in the Figure 3.2 and is convex and piecewise linear. This would correspondingly be a convex bowl shaped piecewise planar surface in three dimensions[1].

Thus if a new weight is considered, then we could start learning from the best policy (which is a point in the highlighted convex function in Figure3.2). The bold piecewise linear and complex function in the figure represents the best reward possible for each weight. This will assure that the agent would learn a

---

[1] The reasoning provided here is similar to that of policies over belief states in POMDPs [7].

policy that is atleast as good as the best policy that it has already learnt. Note that while evaluating a policy, we consider the $\vec{W}.\vec{\rho}$ and not just the average reward $\vec{\rho}$.

Hence we propose to search the space of policies that have been learnt so far for the best policy corresponding to the new weight. We start from the value function of this policy and continue improving it through reinforcement learning. The process of selecting the best policy for the current weight is by obtaining the dot product of the weight vector and the average reward vector that was obtained for that policy. Hence

$$\pi_{opt} = \text{Argmax}_{\pi}\{\vec{W} \cdot \vec{\rho_{\pi}}\}$$

where π_{opt} is the optimal policy for the weight vector $\vec{W}$. $\vec{\rho_{\pi}}$ is the average reward vector for a particular policy π.

Once the current best policy for the new weight vector is obtained, we can initialize the values of the states to the values corresponding to the best policy. Then we can learn starting from these values.

This initialization of the values and rewards to the best policy would guarantee that we would find a policy that is at least as good as the best policy from the set of policies that we have. Since we are learning using an old policy, we might be able to find a better policy for the current weight through reinforcement learning. If the policy learnt is better than the one that we started out with, then we add the new policy to the set of policies that we have.[2]

The algorithm for learning from prior policies is presented in Table 3.1. As can be seen, we search the space of policies already learnt for the best policy

---

[2] Note that our policies are implicitly represented through the corresponding value function vectors.

TABLE 3.1: Algorithm for learning from prior policies

1. Obtain the current weight

2. For the current weight compute, $\pi_{opt} = \arg\max_{\pi}(\overrightarrow{W} \cdot \overrightarrow{\rho_{\pi}})$ from the current set of policies.

   (a) Initialize the value vectors of the states to the values corresponding to π_{opt}

   (b) Initialize the average reward vectors to the average rewards corresponding to π_{opt}

   (c) Learn the new policy π' through reinforcement learning beginning from these values and rewards

3. If $(\overrightarrow{W} \cdot \overrightarrow{\rho'_{\pi}} - \overrightarrow{W} \cdot \overrightarrow{\rho_{\pi}}) > \delta$,add π' to the set of stored policies.

4. Goto step 1.

corresponding to the current weight and then learn beginning from this best policy. We store the new policy learnt if it is atleast δ better than the one we started out with. By doing so, we can reduce the number of policies that are needed to be stored. The idea is that for similar weight vectors, the polices would not differ much and hence need not be duplicated. So though there are infinite number of weight vectors, the number of policies that need to be stored may be quite low. We verify this result empirically.

3.3 Multi-Criteria Model-free Average Reward Reinforcement Learning

The algorithm that we present here is a vector-based version of the algorithm proposed by Schwartz [15]. R-learning, as explained earlier, is an off-policy control method[16]. Here the goal is to maximize the reward obtained per time step. When we are operating in a multi-criteria domain, the goal is going to be to maximize the weighted reward per time step.

The algorithm for Multi-Criteria R-Learning is shown in Table 3.2. As explained earlier, the agent learns the state-action values instead of the state values, as this is a model-free setting. In our algorithm, the agent receives a set of weights. If it is the first weight, then the state value vectors and the reward vectors are set to $\overrightarrow{0}$ (step 2a). If it is not the first weight, then the list of policies that have already been learnt is searched to find the best policy for the current weight vector ($\overrightarrow{W}$), by taking the dot product of the average reward vector and the current weight vector for all the policies and selecting the one with the maximum value(step 2b).

$$\pi_{opt} = \arg\max_{\pi} \{\overrightarrow{W} \cdot \overrightarrow{\rho}^{\pi}\}$$

TABLE 3.2: Multi-Criteria R Learning

1. Obtain the current weight vector $\vec{W}$
2. Initialize the values and rewards:
   (a) If the current weight is the first weight,
   - Set $\vec{\rho} = \vec{0}$
   - Set $\vec{R}(s,a) = \vec{0} \forall s, a$

   (b) Else,
   - Find the best policy (π_{opt}) for the current policy from the set of current policies $\pi_{opt} = \text{argmax}_\pi \{\vec{W} \cdot \vec{\rho}_\pi\}$
   - Set $\vec{\rho} = \vec{\rho}_{\pi_{opt}}$
   - Set $\vec{R}(s,a) = \vec{R}_{\pi_{opt}}(s,a)\ \forall s, a$
3. Learn the best policy for the current weight:
   (a) Let the initial state be s
   (b) Choose an action a such that $a = argmax_a(\vec{W}.\vec{R}(s,a))$ or choose an exploratory action
   (c) Execute the action. Let the next state be s' and the reward is $\vec{r}_{imm}$.
   (d) $\vec{R}(s,a) = \vec{R}(s,a)(1-\beta) + \beta(\vec{r}_{imm} - \vec{\rho} + max_{a'}[\vec{W}.\vec{R}(s',a')])$
   (e) If a is an optimal action,

$$\vec{\rho} = \vec{\rho}(1-\alpha) + \alpha(\vec{r}_{imm} + \vec{R}(s',a') - \vec{R}(s,a)) \text{ where,}$$

$$a = \arg\max_a[\vec{W} \cdot \vec{R}(s,a)] \text{ and } a' = \arg\max_{a'}[\vec{W} \cdot \vec{R}(s',a')]$$

   (f) $s = s'$. goto step(b).
4. If $(\vec{W} \cdot \vec{\rho_\pi'} - \vec{W} \cdot \vec{\rho_\pi}) > \delta$, where π' and π are the new policy and the old policy respectively, add π' to the set of policies.

Once the best policy is determined, the $\overrightarrow{R}(s,a)$ values and the ρ vectors for the new weight vector are set to the corresponding values of the best policy. The agent then begins to optimize the policy for the new weight vector (step 3). The R-value vectors are updated using the immediate reward and the action a' that maximizes the value $\overrightarrow{W}.\overrightarrow{R}(s',a')$ (step 3d). The value of the average reward vector ($\overrightarrow{\rho}$) is updated only if the action executed in a state is a non-exploratory one (step 3e). The learning rate for the value vector β, is generally higher than that of the learning rate α for the average reward vector. After it has converged on the correct policy for the given weight, it stores the new policy if it is a better policy than the one the agent started out with (step 4).

Since the agent starts learning using the best among the old policies for the new weight vector, it is expected to start from a reasonably good policy and converge quickly on the optimal policy. This new optimal policy is expected to be atleast as good as the current best policy. Thus the agent learns a better policy in a shorter time.

3.4 Multi-Criteria Model-based Average Reward Reinforcement Learning

In the previous chapter, the Model-Based Average-Reward Reinforcement Learning was introduced. The algorithm that we now describe is a multi-criteria adaptation of the H-Learning algorithm proposed by Tadepalli and Ok [17]. As in the case with the Multi-Criteria R Learning, the values and rewards are vectors. H-Learning learns the models by maximum likelihood estimation.

The algorithm for the model-based average-reward multi-criteria reinforcement learning is shown in Table 3.3. As mentioned before, the $r(s,a)$, $h(s)$ and ρ values are turned into vectors, $\overrightarrow{r}(s,a)$, $\overrightarrow{h}(s)$ and $\overrightarrow{\rho}$, in this method. Similar to

TABLE 3.3: Multi-Criteria H Learning

1. Obtain the current weight vector $\overrightarrow{W}$
2. Initialize the values and rewards:
   (a) If the current weight is first weight,
   - Set $\overrightarrow{\rho} = \overrightarrow{0}$
   - Set $\overrightarrow{r}(s,a) = \overrightarrow{0}\ \forall s,a$
   - Set $\overrightarrow{h}(s) = \overrightarrow{0} \forall s$
   - $N(s,a,s') = 1; \forall s,a,s'$
   - $N(s,a) = n; \forall s,a$
   - $p_{s,s'}(a) = 1/n; \forall s,a,s'$

   (b) Else,
   - Find the best policy(π_{opt}) for the current policy from the set of current policies $\pi_{opt} = \text{Argmax}_\pi\{\overrightarrow{W} \cdot \overrightarrow{\rho}_\pi\}$
   - Set $\overrightarrow{\rho} \longleftarrow \overrightarrow{\rho}_{\pi_{opt}}$
   - Set $\overrightarrow{r}(s,a) \longleftarrow \overrightarrow{r}_{\pi_{opt}}(s,a)\ \forall s,a$
   - Set $\overrightarrow{h}(s) = \overrightarrow{h}_{\pi_{opt}}(s) \forall s$
3. Learn the best policy for the current weight:
   (a) Let the initial state be s
   (b) Choose an action a such that $a = max_a(\overrightarrow{W} \cdot (\overrightarrow{r}(s,a) + \sum_{s'=1}^{n} p_{s,s'}(a)\overrightarrow{h}(s')))$ or choose an exploratory action
   (c) Execute the action. Let the next state be s' and the reward is $\overrightarrow{r}_{imm}$.
   (d) $N(s,a) \longleftarrow N(s,a) + 1$; $N(s,a,s') \longleftarrow N(s,a,s') + 1$
   (e) $p_{s,s'}(a) \longleftarrow N(s,a,s')/N(s,a)$
   (f) $\overrightarrow{r}(s,a) \longleftarrow \overrightarrow{r}(s,a) + (\overrightarrow{r}_{imm} - \overrightarrow{r}(s,a))/N(s,a)$
   (g) If a is a greedy action,
   - $\overrightarrow{\rho} \longleftarrow \overrightarrow{\rho}(1-\alpha) + \alpha(\overrightarrow{r}(s,a) - \overrightarrow{h}(s) + \overrightarrow{h}(s'))$
   - $\alpha \longleftarrow \frac{\alpha}{\alpha+1}$

   (h) $\overrightarrow{h}(s) \longleftarrow max_a\{\overrightarrow{W} \cdot (\overrightarrow{r}(s,a) + \sum_{s'=1}^{n} p_{s,s'}(a)\overrightarrow{h}(s')\} - \overrightarrow{\rho}$
   (i) $s = s'$. goto step(b).
4. If $(\overrightarrow{W} \cdot \overrightarrow{\rho'_\pi} - \overrightarrow{W} \cdot \overrightarrow{\rho_\pi}) > \delta$,where π' and π are the new policy learnt and the old policy respectively, add π' to the set of policies.

the model-free case, we initialize the value vectors and average reward vectors to zero (step $2a$). In addition, we also initialize the transition model values. Once again, we check for the policy that maximizes the $\{\overrightarrow{W} \cdot \overrightarrow{\rho}\}$ value and use this policy to bootstrap learning (step $2b$).

Instead of the state-action pairs, we associate values with states and the rewards with state-action pairs. The corresponding vectors are initialized to those of the best policy. Then the agent learns starting from that policy.

Note that the model is not re-learnt for each weight (steps $3d$, $3e$). It continues to update the model that it has learnt. So in some sense, while it receives a new weight, it already has a good model of the environment which further speeds up the learning beyond the model-free method. The agent would be expected to converge on the optimal policy faster as it starts from the best policy currently available for that weight and learns from it. We verify this empirically in the next section. It updates its $\overrightarrow{r}(s,a)$ values based on the immediate reward (step $3f$). Also if the executed action is a greedy one, it updates the average reward vector values and the h-value vectors else it just updates the h-value vectors (steps $3g$, $3h$). The idea of storing the policies is same as that of the model-free case (step 4).

CHAPTER 4

IMPLEMENTATION AND RESULTS

Having proposed the algorithm for Multi-Criteria RL, we provide the empirical verification of our hypotheses. This chapter deals with the two domains that we tested our algorithms on the modified version of Buridan's ass problem and the network routing domain. We explain the experimental set up and the implementation details. We also outline the results that were obtained for the Multi-Criteria versions of R and H learning.

4.1 Grid world domain

This modified version of the grid world domain is shown in Figure 3.1. As can be seen, the donkey is in the center square of the 3 x 3 grid. There are food piles on the diagonally opposite squares. The food is visible only from the adjacent squares, i.e., the squares that are next to it in the eight directions. If the donkey moves away from the adjacent square of a food pile, there is a certain probability p_{stolen} with which the food will be stolen. Food is regenerated once every N_{appear} time-steps.

The donkey is a glutton. It has to eat atleast once in ten time-steps. Otherwise it is penalized with a hunger penalty. Also the donkey is greedy in the sense that it doesn't want to lose the food. So it is trying to minimize the amount of food being stolen per unit time. Every time a food pile is stolen, the

donkey is penalized. The other quality of the donkey is that it is lazy. Hence it wants to minimize the number of steps it walks per unit time. For every step it takes, the donkey is penalized with a walking penalty.

Basically the donkey has to strike a compromise between optimizing the three different criteria: hunger, food being stolen, and walking.

4.1.1 Experimental Setup

A state is a tuple $\langle s, f, t \rangle$, where s stands for square, f for food and t, time. s can take 9 values between 0 and 8 corresponding to the square in which the donkey is currently in. f can take 4 values corresponding to whether food is present in both piles or absent in both or present in one of them. t is the time counter, which basically takes 10 values (0-9) to determine whether the donkey is hungry. Once the donkey eats the food, t will be reset to 0. If t reaches 9 and the donkey hasn't eaten the food, t is not incremented or reset. Instead until it eats, t stays at 9 and the donkey is penalized with -1 per time step until it eats the food. The actions are move up, down, left, right, and stay. It is assumed that if the donkey chooses to stay at a square with food, then it eats the food.

The probability of food being stolen(p_{stolen}) is 0.9. N_{appear}, the number of time steps when the food would reappear is 10. The number of time-steps that the donkey is allowed to stay without eating is 10. When the donkey becomes hungry, it is penalized with -1 every time step until it once again eats. Also, when each food pile is stolen the donkey is penalized with a negative reward of -0.5 per pile. For every step it walks, it is penalized with a penalty of -1.

The R-Learning and the H-Learning versions presented in the previous chapters were implemented. The weights were generated at random and the three

components were normalized so that they add up to 1. These were then stored in an array. For each weight, the programs were run for 100,000 time-steps. The program was allowed to run for 1000 time steps and the policy was evaluated for the next 1000 time-steps to determine the convergence. While the agent was learning, an ϵ-greedy policy was followed, and during evaluation, the agent was allowed to choose only greedy actions and accumulate the rewards. As stated earlier, we predicted that after a certain number of weights, the agent need not learn for a new weight vector, and instead use an already existing policy. What this means is that after a certain number of weights, the agent must converge to a correct policy in 1000 time steps. This is due to the fact that we are evaluating the policy once every 1000 time-steps.

The correctness of the policies learnt by both the agents were verified manually. The extreme cases of weights were verified. For instance, the weight vector contains the following components: $\langle W_{hunger}, W_{stolen}, W_{walking} \rangle$. So if the weight vector $\vec{W} = \langle 1, 0, 0 \rangle$, it means that hunger is the most important criterion. So the donkey would then walk to one of the food piles and stand there. Whenever the food is re-generated, the donkey would eat it. For the vector $\langle 0, 1, 0 \rangle$, the donkey would not move out of the square.

The set of policies learnt for different weights are presented in 4.1.

4.1.2 Multi-Criteria R Learning

The R-learning algorithm was presented in the previous chapter. The α value was set to .01 and β was set to .05.

We have presented two graphs: One showing the learning curve for a few weights(Figure 4.2) and the other showing the number of steps required for

Weights			Policy
H	S	W	
1	0	0	Go to one of the plates and stay there
0	1	0	Stay at the center square
.5	0	.5	Go to one of the plates and stay there
0	.5	.5	Stay at the center square
.33	.33	.33	Go to one of the plates and stay there
0	0	1	Stay at some square(center/food)
.5	.5	0	Alternate b/w food plates and eat

FIGURE 4.1: Policies for R and H Learning corresponding to different weights

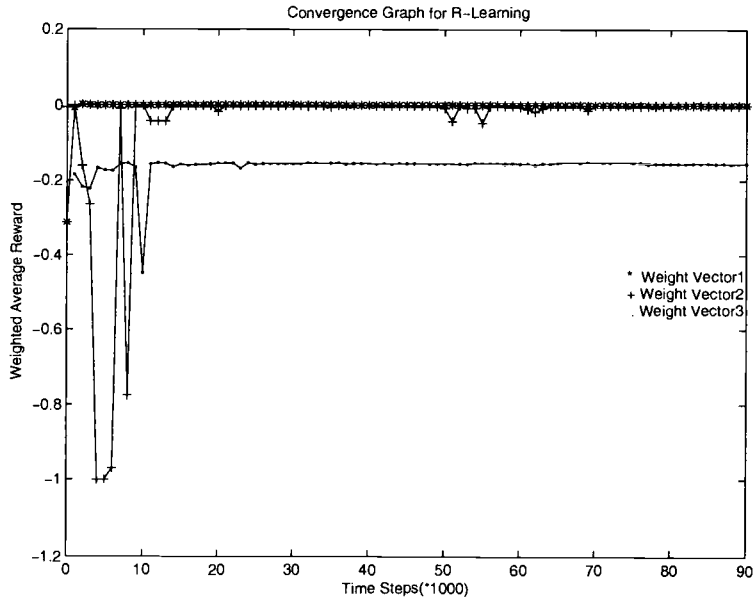

FIGURE 4.2: Learning curves for 3 weights using R-Learning

convergence vs the number of weights(Figure 4.3). The three weight vectors in Figure 4.2 are $\langle 1,0,0\rangle$,$\langle 0,1,0\rangle$ and $\langle .33,.33,.33\rangle$. It can be seen that the weighted average reward$\{\vec{W}\cdot\vec{\rho}\}$ converges to zero in the first two cases. In the third case, the agent chooses to guard one food pile and keeps eating it. The agent cannot alternate between piles, as it obtains a penalty of -1 for every step that it walks.

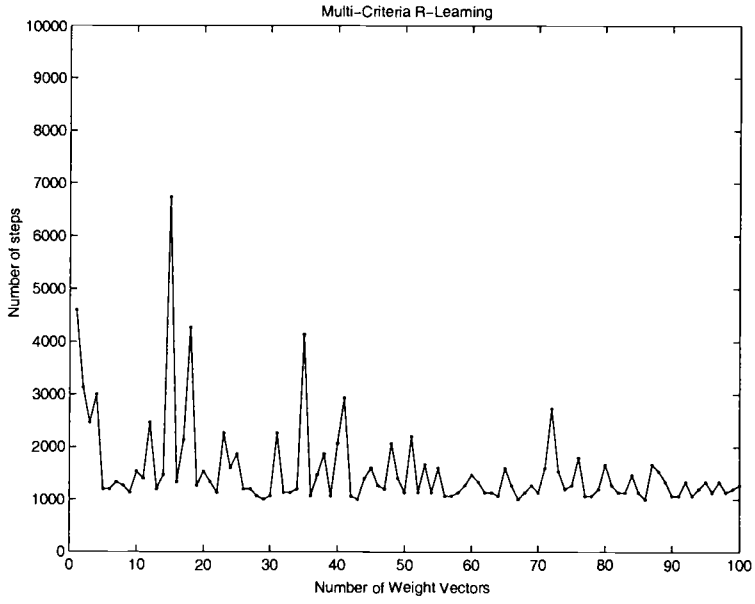

FIGURE 4.3: Convergence graph for R-Learning

The second graph for R-learning is the one that shows the number of time-steps required to converge to the correct policy against the number of weight vectors. The policies were verified manually for a few weights(eg. Figure 4.1). This is presented in Figure 4.3. The agent was allowed to learn for 15 different

runs, each of 100 weight vectors. For each weight vector, we obtain the number of steps required for convergence of each run and calculate the average number of steps. The graph is then plotted with the average number of steps required for convergence of the algorithm against the number of weight vectors. As can be observed, the agent settles for one of the learnt policies after about 50 weight vectors. The number of peaks that are high in the first half of the graph indicates that the agent is learning a policy that is different from the one it started out with. The total number of polices that are stored was between 15 and 20. Thus the agent used about 20 different policies for 100 weight vectors.

4.1.3 Multi-Criteria H Learning

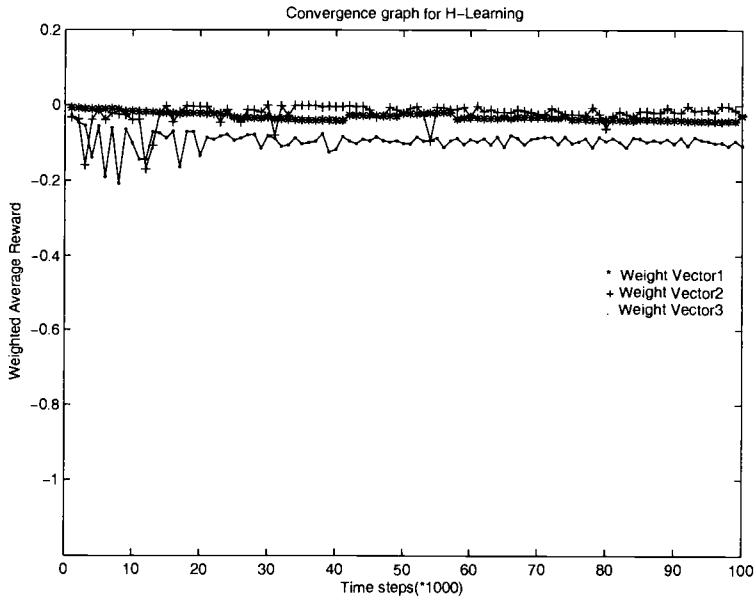

FIGURE 4.4: Learning curves for 3 weights using H-Learning

Chapter 3 described the Model-Based Multi-Criteria Average-Reward RL algorithm in detail which is a Multi-Criteria version of H-Learning. We present 2 graphs similar to those of R-Learning. One is the learning curve for 3 weights and the other is the graph showing the number of steps required for convergence. The learning curve is presented in Figure 4.4. The weights are $\langle 1, 0, 0 \rangle$, $\langle 0, 1, 0 \rangle$ and $\langle .33, .33, .33 \rangle$. The agent converges pretty quickly in all the cases.

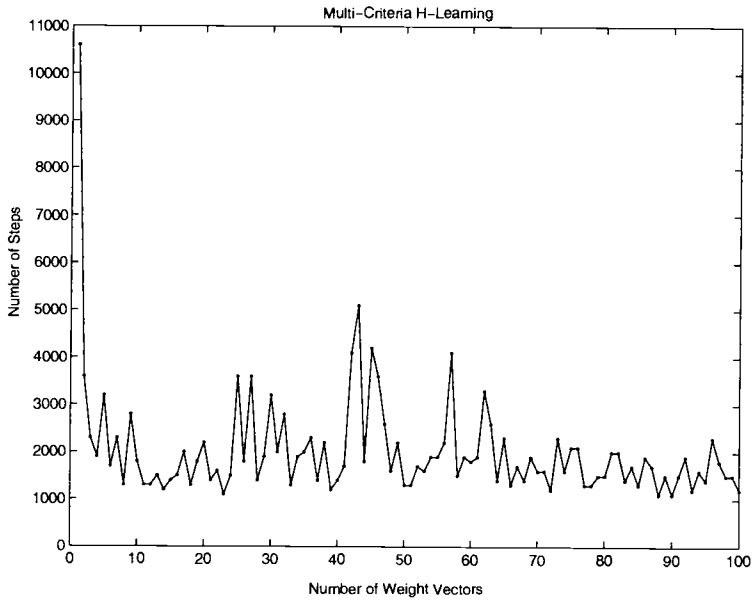

FIGURE 4.5: Convergence graph for H-Learning

The graph for the convergence of H-Learning in the grid world domain is shown in Figure 4.5. The program was run for 15 different sets of 100 weights each that were generated at random. The weighted average ($\vec{W} \cdot \vec{\rho}$) was obtained for each weight. Then the average number of steps were plotted against

the number of weights. The results are similar to that of R-Learning. Initially the agent took sometime to learn the model and then after some number of weight vectors, uses the model and the prior policies learnt to operate in the domain for new sets of weight vectors. So the graphs have a few peaks in the initial stages and later settles down to around 1000 time-steps. The number of policies learnt is also similar to that of R-Learning.

4.2 Network Routing Domain

The other domain in which we tested our algorithm was the network routing domain. In order to transfer packets from a source host to a destination host, the network layer must determine the path or route that the packets are to follow [5]. At the heart of any routing protocol is the routing algorithm that determines the path of a packet from the source to the destination [5].

In network routing there are three main criteria:

- end-to-end delay - Time required for a packet to travel from the source to the destination. This includes the propagation delay, the transmission delay, and the processing delay
- packet loss - The loss of packet due to congestion or router/link failure
- power - The power level associated with a node

The multiple objectives have some weights associated with them, and the weights change frequently. For instance, during war scenarios in mobile networks, power is the main concern while in some other situations, packet loss or delay would be the main issue. So in all the cases, apart from optimizing the

main parameter which could be end-to-end delay or packet loss or the power, it would be desirable to optimize the other parameters as well.

4.2.1 Experimental Setup

Optimized Network Engineering Tools (OPNET) is a comprehensive engineering system capable of simulating large scale communications networks [1]. OPNET features include: graphical specification of models; a dynamic, event-scheduled simulation kernel; integrated data analysis tools; and hierarchical, object-based modeling. It is specially suited to our environment, as it provides a hierarchical modeling structure that enables the development of distributed algorithms easily.

We used OPNET to design and implement our algorithm on the network domain. It has been chosen as a modeling environment for this work due to its modular architecture and flexibility. In addition, it provides an impressive library of protocols and devices to facilitate the study and development of networks at any level.

OPNET has a three-tired Network Hierarchy:

- Network Model - Specifies the overall network topology
- Node Model - Specifies the object in the network model
- Process Model - Specifies the object in the node model

4.2.2 Network Model

The network that we used to test our algorithms is shown in Figure 4.6

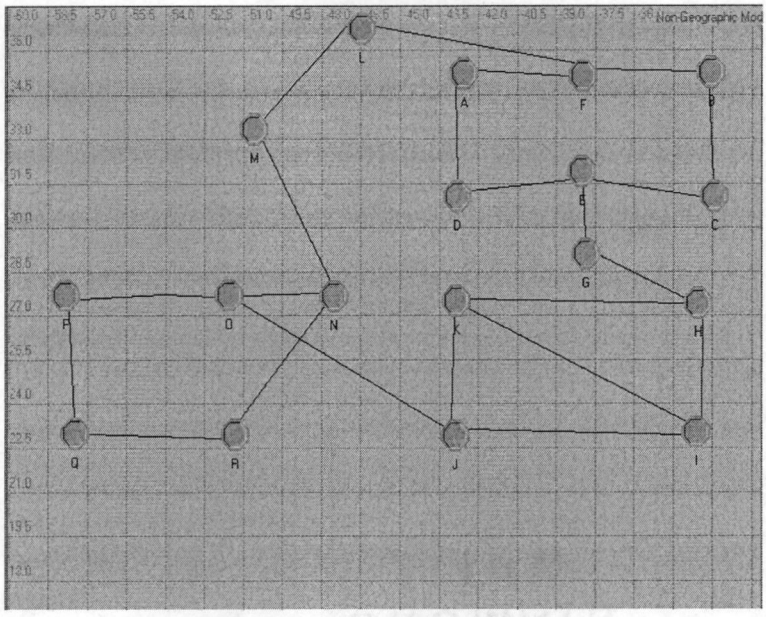

FIGURE 4.6: Network Model

The network model in OPNET consists of nodes, links, and subnets. We have created our own node model which we would describe in the next section and used the point-point duplex link provided by the OPNET. The data rate for the link was set at 9600bps. Basically each node has a probability (P_{drop}) of dropping the packet. The packets that are received from the other nodes are the ones that could be dropped. A node does not drop the packet that it creates. A special packet format was created. These packets had the destination, source, packet number and a table field.

4.2.3 Node Model

Figure 4.7 shows the node model for a node of degree 3. We also have nodes of degree 2, that have one receiver and one transmitter fewer than this model. As

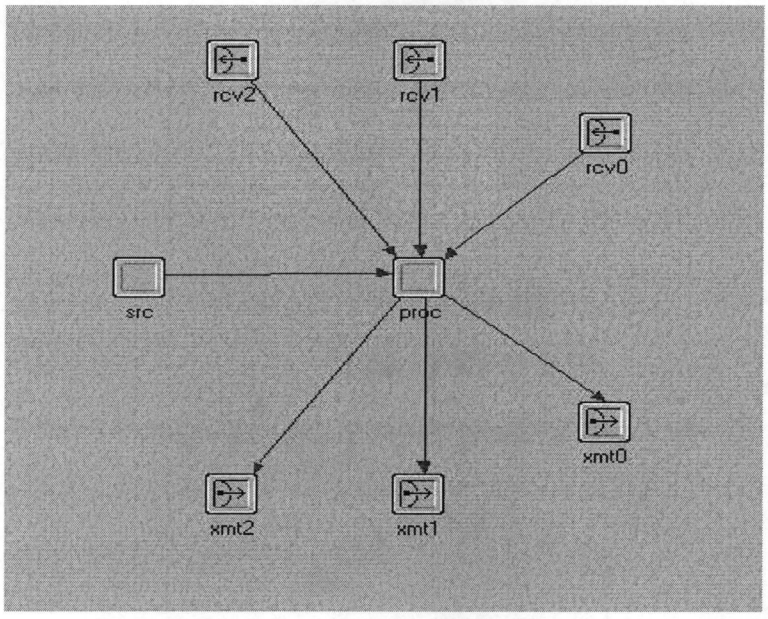

FIGURE 4.7: Node Model for a node of degree 3

can be seen from the figure, each node has a source that generates the packet at a constant rate. This distribution can be changed in OPNET. Also each node has transmitters and corresponding receivers. While creating the network model, the receivers and the corresponding transmitters are attached to the link. Each node has a processor that has incoming links from the receivers and outgoing links to the transmitters. Upon receiving a packet from either the source or the receivers, the processor decides what to do with the packet. The process model of the processor is discussed in the next section.

4.2.4 Process Model

The process model of the processor in the node is shown in Figure 4.8. There are 4 states in the model. The init state contains the code to be executed in

the beginning of the simulation. In our case, we initialized the data structures and read the weights from the file into an array. The next state is the idle state. This state is the default state of a processor. There are two transitions from the idle state. These are the "PK_ARRVL" and "SRC_ARRVL" events corresponding to the packet arriving from the receiver and the source. The corresponding states that are reached are receive and recv_source. Basically if a packet is received from the source, its information is entered into a list. Then it is routed accordingly. If a packet is received from a receiver, then it has to be checked for the destination. If a node receives its packet from a neighbor, then it processes the packet. If the packet is for another destination then it is routed accordingly.

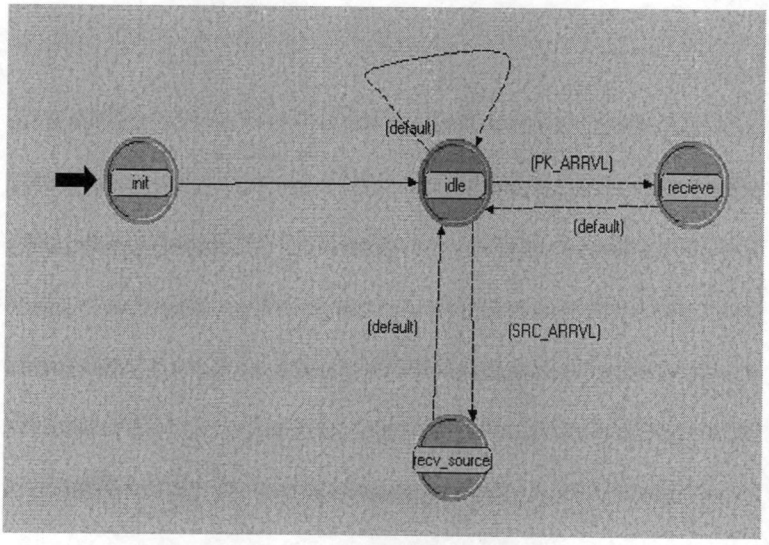

FIGURE 4.8: Process Model for the processor of a node

4.2.5 Implementation details

The network that was shown in Figure 4.6 was used for testing our algorithms. The weights were generated initially at random and were written to a file. When the simulation was started, the weights were read into an array by individual nodes. Then the data structures were initialized accordingly. The lists for the reward and value functions were created. Then the idle state is reached where it waits until an event happens. Since OPNET is an event-driven simulator, the time gets updated only if an event occurs.

If a node is in the idle state and a packet arrives from the source, it enters the recv_source state. Here it appends the source and destination addresses to the packet and then routes the packet. It stores the details of the packet in a list. It also sends a table packet that contains the value functions to its neighbors once every T_{table} seconds. When a packet is received from a neighbor, it checks for the destination. If the destination is its own address, it processes the packet. If it is a data packet, it sends an acknowledgment to the source. If it is an acknowledgment, then it updates the values and rewards.

The node also stops exploring after every T_{learn} seconds and then evaluates its policy for the next $T_{evaluate}$ seconds. Also, each node would send the information about its power to its neighbors every T_{power} seconds. The power level of each node decreases with the increase in the number of packets it processes. This power value is reset once a new weight vector arrives.

The "state" in this domain basically consists of the destination of the current packet and the current node. The action to be executed in a state is the neighbor to which the packet has to be sent. The value function is represented in a distributed way, in that each node stores its value function for each destination node ($R_{curr\_node}(destination, neighbor)$). There is a global reward function

(average reward function in our case) that all the nodes are trying to maximize.

The immediate reward components were: r_{ete}, r_{pl}, r_{pow} corresponding to the end-to-end delay, packet loss and power respectively. The immediate reward values were between 0 and -1. The end-to-end delay was brought between 1 and 0 by a linear transformation of the simulation time. Also for every packet that is lost, the immediate reward was -1. Basically a node would wait for a certain time period T_{pl}, to determine if the packet is lost. The power value that was received from the neighbor was used as the immediate reward for the action that chooses that neighbor.

As stated earlier, each node would generate some packets for random destinations. Once the acknowledgment is received from a destination, it updates the value functions corresponding to that destination. The end-to-end delay and the power level of the neighbors are used as the different immediate reward components. Then the current packet details are removed from the list of packets sent. Also, the list of packets sent is searched for determining if some packets are lost and the value functions of the corresponding neighbor is updated with the packet loss penalty. The R-Learning and the H-Learning agents were allowed to run for 10,000 seconds for each weight.

4.2.6 R-Learning

T_{table} was set to 400, while T_{learn} and $T_{evaluate}$ were both set to 500. Each node had a $\overrightarrow{R}(s,a)$ vector for each state-action pair. The set of actions is the set of neighbors to choose from. All the nodes accessed the global reward using mutual exclusion constraints and updated them. The agent learnt for 10,000 seconds of simulation time and then would read in a new weight vector.

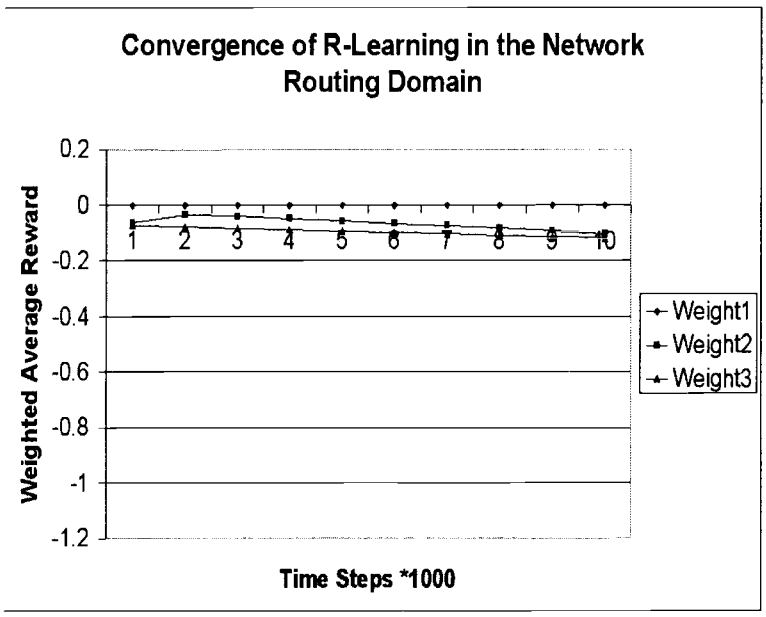

FIGURE 4.9: Learning curves for 3 weights in the Network Routing domain using R-Learning

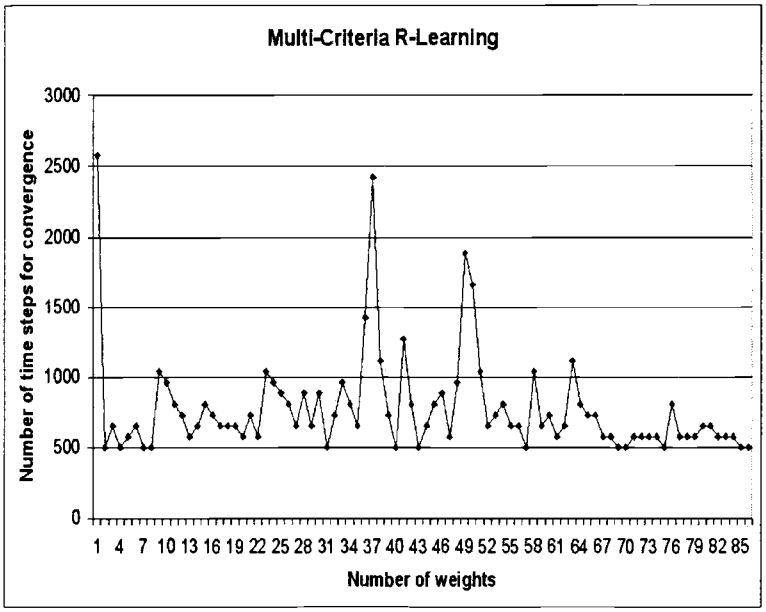

FIGURE 4.10: Convergence graph for R-Learning in the Network Routing Domain

The learnings curve for 3 random weights are presented in Figure 4.9. The results were obtained after every 1000 seconds, when the agent was allowed to learn for the first 500 seconds using an ϵ-greedy strategy with ϵ set to 0.1 and then was evaluated for the next 500 seconds. As can be seen, the system learns the policy in a very short period of time. We verified the policies manually for a few weights.

The convergence graph for R-Learning is presented in Figure 4.10. As we had predicted, the agent takes some time to learn a bunch of policies and then uses them later to initialize learning. Hence for the initial set of weights, the program takes some time to converge, while it converges very quickly after a certain number of weights. The data were collected from 15 runs and averaged over them. For each run, the program was executed for 10 days of simulation time. The statistics were collected for 86 weights. The convergence curve was plotted for these 86 weight vectors. As can be observed, the program would settle for a learnt policy around 60 weight vectors.

4.2.7 H-Learning

The models were learnt from scratch for the first weight and the learnt model was updated for the later weights. The table time and the exploration strategy were the same as that of R-Learning. This method followed an ϵ-*greedy* strategy for exploration as did the R-Learning with $\epsilon = 0.10$. The results are presented in Figures 4.11 and 4.12. Similar to the previous domain, the results are pretty similar to that of the R-Learning.

The learning curve in Figure 4.11 shows that the program converges very quickly to the optimal policy. Three random weights were used, and the learning

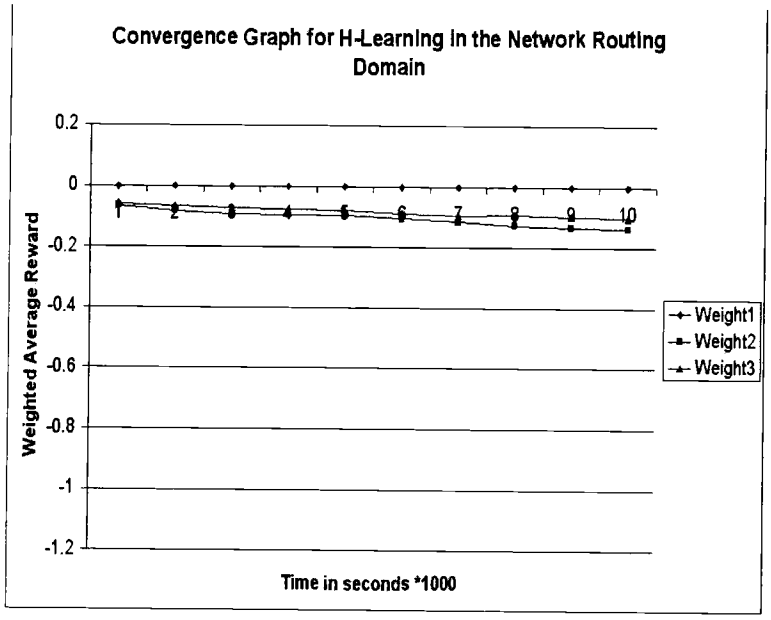

FIGURE 4.11: Learning curves for 3 weights in the Network Routing domain using H-Learning

curves were plotted after the agent learns on those weights. As was the case with R-Learning, the agent was allowed to learn for about 500 seconds of simulation time, and then it was evaluated for the next 500 seconds, and the results are presented. They are very similar to those of R-Learning.

The convergence graph is shown in Figure 4.12. Once again, the agent learns a set of policies when it receives the first set of weights (in this case around 55 weights). Then it uses the model learnt and also the best policy from the set of policies to learn the best policy for a new weight vector. Like R-Learning, the program was executed for 10 days of simulation time and the average number of steps required for convergence was collected for 86 weight vectors. The data were averaged over 15 runs. As predicted, the algorithm converges very quickly

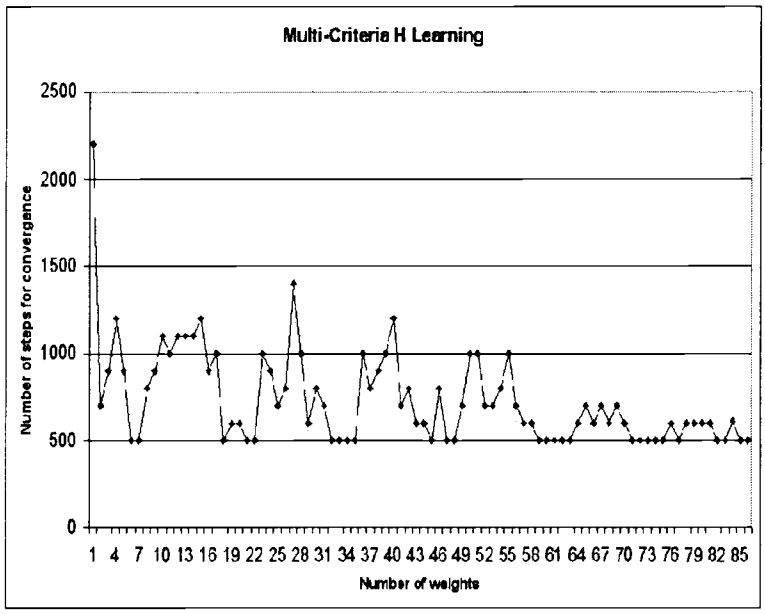

FIGURE 4.12: Convergence graph for H-Learning in the Network Routing domain

for later weight vectors.

CHAPTER 5

CONCLUSION

The basic premise of the thesis is that many real-world problems have multiple goals to be achieved. In such cases, the normal scalar based reinforcement learning techniques may not suffice. It becomes imperative that the value functions and rewards are decomposed. Also the weights governing the criteria may change, and the agent must be able to adapt to this change efficiently.

We motivated the idea of vector based reinforcement learning with an example. We discussed the Q-value decomposition of Russell et.al[14]. and Multi-Criteria Reinforcement Learning technique of Gabor et.al[6]. We explained the need of using weights for representing the importance of different criteria.

We presented an average-reward multi-criteria reinforcement learning algorithm that improved starting from the policies that it has learnt previously. We showed that the agent initially learns a set of policies for different weights and after a few weights was able to use the learned policies and converge quickly.

We empirically verified that a small number of policies can cover most of the weight space. This directly means that after a certain number of policies are learnt, the agent could use the learned policies there resulting in a huge decrease in the learning time.

We demonstrated this idea with experimentation in both a toy domain and a network routing domain. We showed that with a network of about 20 nodes, the agent would converge to a correct policy in the first evaluation phase.

We have not presented a theoretical proof for the convergence of the policies after a few weight vectors. This in turn depends on the convergence proofs of Average-Reward RL methods. The number of distinct optimal policies depends on the structure of the MDPs and needs to be better understood. Also, we haven't explored the idea of using function approximation. In our domains, the state space is not huge, and so we did not use any function approximation techniques.

Another interesting problem is when the user is unable to provide weights but simply controls the agent. For example, he could just route the packets in the network. The question is then: How do we learn the weights that he has in mind? How do we obtain the rewards given the policy? Russell and Ng called this problem as "Inverse Reinforcement Learning" [10]. Koller et.al use the past decisions of the agent to predicts its future decisions [18]. More recently, Ng et.al address this problem as apprenticeship learning [12]. Boutilier studies the problem as Preference Elicitation [2]. Basically, the agent has to recommend courses of actions for a specific user. This requires the knowledge of the user's preferences or the utility function. These functions can vary widely from user to user. He formulates this problem as a POMDP and describes methods that exploit the structure of the preferences and then uses gradient techniques for optimization [2]. It would be interesting to see if and how these ideas can be used to elicit the importance weights from observed behavior in Multi-Criteria Reinforcement Learning.

BIBLIOGRAPHY

[1] *OPNET reference manual, www.opnet.com.*

[2] Craig Boutilier. A POMDP formulation of preference elicitation problems. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002), Edmonton, AB, pp.239–246 (2002)*, 2002.

[3] Ronald Parr Carlos Guestrin, Daphne Koller. Multiagent planning with factored mdps. In *Advances in Neural Information Processing Systems NIPS-14*, 2001.

[4] DoKyeongOk. *A Study of Model-based Average Reward Reinforcement Learning.* Ph.D dissertation, Oregon State University, 1996.

[5] James F.Kurose and Keith W.Ross. *Computer Networking - A Top-Down Approach Featuring the Internet.* Pearson Education, second edition, 2003.

[6] Zs. Kalmar Gabor, Zoltan and Cs. Szepesvari. Multi-criteria Reinforcement Learning. In *In Proc. ICML-98*, 1998.

[7] Michael L. Littman Leslie Pack Kaelbling and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *AI Journal*, 1998.

[8] Michael L. Littman Leslie Pack Kaelbling and Andrew W. Moore. Reinforcement learning: A survey. *Journal of AI Research*, 1996.

[9] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 1996.

[10] Andrew Y. Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[11] Ronald Parr. Flexible decomposition algorithms for weakly coupled markov decision problems. In *UAI*, 1998.

[12] Andrew Y. Ng Pieter Abbeel. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML)*, 2004.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, first edition.

[14] Stuart Russell and Andrew L. Zimdars. Q-decomposition for reinforcement learning agents. In *In Proc. ICML-03*, 2003.

[15] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *In Proc. ICML-1993*, 1993.

[16] Richard S.Sutton and Andrew G.Barto. *Reinforcement Learning - An Introduction.* MIT Press, London, England, first edition, 1998.

[17] Prasad Tadepalli and DoKyeongOk. Model-based average reward reinforcement learning. *AI Journal*, 2000.

[18] Dirk Ormoneit Urszula Chajewska, Daphne Koller. Learning an agent's utility function by observing behavior. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, 2001.

[19] Wei-Men-Shen. *Autonomous Learning From the Environment.* Computer Science Press, 1994.