Dynamic Preferences in Multi-Criteria Reinforcement Learning

Sriraam Natarajan NATARASR@EECS.ORST.EDU
Prasad Tadepalli TADEPALL@EECS.ORST.EDU
School of Electrical Engineering and Computer Science, Oregon State University, USA

Abstract

The current framework of reinforcement learning is based on maximizing the expected returns based on scalar rewards. But in many real world situations, tradeoffs must be made among multiple objectives. Moreover, the agent's preferences between different objectives may vary with time. In this paper, we consider the problem of learning in the presence of time-varying preferences among multiple objectives, using numeric weights to represent their importance. We propose a method that allows us to store a finite number of policies, choose an appropriate policy for any weight vector and improve upon it. The idea is that although there are infinitely many weight vectors, they may be well-covered by a small number of optimal policies. We show this empirically in two domains: a version of the Buridan's ass problem and network routing.

1. Introduction

Reinforcement Learning (RL) is the process by which an agent learns an approximately optimal behavior 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. In this work, we consider average reward optimization, which aims to optimize the expected average reward per time step over an infinite horizon. Traditional RL techniques are essentially scalar-based, i.e., they aim to optimize an objective that is expressed as a function of a scalar reinforcement. In many real world domains, however, the reward may not be a single scalar function. For example, in network routing the goals might be to reduce the end-to-end delay in routing packets as well as to reduce the power loss in the routers. In inventory control, one might have the goal of minimizing the shortage costs while also keeping the inventory costs low. A government may have to decide between how much to spend on national defense vs social welfare. Also, in manufacturing, there are competing goals: increase the profits and improve the quality of the products.

The problem of conflicting goals is well-captured by the Buridan's ass problem. There is a donkey at equal distance from two piles of food. The problem is that if it moves towards one of the piles, the food in the other pile can be stolen. If it stays in the center to guard the food, it might eventually starve to death. It is clear that the agent has to find a reasonable compromise between the two goals. Classical approaches to solving multi-objective Markov decision problems (MDPs) are based on vector-based generalizations of successive approximation (e.g., see (White, 1982)). In more recent work on multi-criteria reinforcement learning (MCRL), there is a fixed lexicographic ordering between different criteria, such as staying alive and not losing food (Gabor et al., 1998). This approach is based on abstract dynamic programming using component-wise reinforcement learning operators. Mannor and Shimkin take a geometric approach to MCRL and view the objective as approaching an increasingly smaller nested target sets of states (Mannor & Shimkin, 2004). Their method is applicable to the more general setting of stochastic games as well as average-reward optimization in MDPs. In the second, "weighted criterion" setting, the problem is formulated as optimizing a weighted sum of the discounted total rewards for multiple reward types (Feinberg & Schwartz, 1995). They adopt a constrained MDP framework in which one criterion is optimized subject to some constraints on other criteria and solve it using a linear programming formulation (Altman, 1999).

To the best of our knowledge, none of these and other methods in the literature handles the case of dynamic or time-varying preferences, which is the subject of this paper. This is what happens, for example, when a government changes in a country. The priorities and preferences change and so should the policies. We formulate the dynamic multi-criterion RL problem using a weighted average-reward framework. It is easy to see that with fixed weights, the weighted average-reward criterion reduces to

Appearing in *Proceedings of the 22nd International Conference on Machine Learning*, Bonn, Germany, 2005.

the scalar case with the weights incorporated into the reward function. However, time-varying criteria give us an opportunity to exploit the structure of the MDPs to speed-up learning by remembering a small number of previously learned policies. When a change occurs in priorities as reflected by changes in the weights, the agent can start from the best policy it learned so far and improve it further, rather than starting from scratch for each new weight. Moreover, as will be shown in our experimental results, after learning a relatively small number of policies, the agent may not need to learn any more policies since they cover a large part of the relevant weight space.

The rest of the paper is organized as follows. Section 2 provides the background on average-reward reinforcement learning (ARL). Section 3 motivates and presents the dynamic MCRL algorithm schema, which can be used in conjunction with any ARL algorithm. Section 4 presents the results of applying our algorithm schema with model-free R-learning and model-based H-learning in 2 domains: the Buridan's ass domain and network routing domain. Section 5 concludes this paper and outlines some areas for future research.

2. Average Reward Reinforcement Learning

An MDP is described by a set of discrete states S, a set of actions A, a reward function $r_s(a)$ that describes the expected immediate reward of action a in state s, and a state transition function $p^a_{ss'}$ that describes the transition probability from state s to state s' under action a. A policy is defined as a mapping from states to actions, i.e., a policy π specifies what action to execute in each state. An optimal policy in average reward setting is one that maximizes the expected long-term average reward per step from every state. Unlike in discounted learning, here the utility of the reward is the same for an agent regardless of when it is received. The Bellman equation for average reward reinforcement learning for a fixed policy π is:

$$h^\pi(s) = r_s(\pi(s)) + \sum_{s'} P^{\pi(s)}_{ss'} h^\pi(s') - \rho \qquad (1)$$

where ρ is the average reward per time step of the policy π. Under reasonable conditions on the MDP structure and the policy π, ρ is constant over the entire state-space. The idea behind the Bellman equation 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. The difference between $r_s(a)$ and ρ is called the average-adjusted reward of action a in state s. $h^\pi(s)$ is called the bias or the value function of state s for policy π and represents the limit of the expected value of the total average-adjusted reward over the infinite horizon for starting from s and following π (Puterman, 1994).

H-Learning, is a model-based version of average reward reinforcement learning (Tadepalli & Ok, 1998). The optimal policy chooses actions that maximize the right hand side of the above Bellman equation. Hence, H-learning also chooses greedy actions, which maximize the right hand side, substituting the current value function for the optimal one. It then updates the current value function as follows:

$$h(s) \longleftarrow max_a\{r_s(a) - \rho + \sum_{s'=1}^{n} p_{ss'}(a)h(s')\} \qquad (2)$$

The state-transition models p and immediate rewards r are learned by updating their running averages. The average reward ρ is updated using the following equation over the greedy steps.

$$\rho \longleftarrow \rho(1-\alpha) + \alpha(r_s(a) - h(s) + h(s')) \qquad (3)$$

R-Learning is a model-free version of H-learning that uses the action-value representation and is an analogue of Q-learning (Schwartz, 1993). The action value $R^\pi(s,a)$ represents the value of executing an action a in state s and then following the policy π. 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 update equation for R-Learning is:

$$R(s,a) \longleftarrow R(s,a)(1-\beta) + \beta(r_{imm} - \rho + max_{a'}R(s',a')) \qquad (4)$$

Although, there is no proof of convergence of R-Learning or H-Learning, they are experimentally found to converge robustly to optimal policies on reasonably sized problems with sufficient exploration (Tadepalli & Ok, 1998; Mahadevan, 1996).

3. Dynamic Multi-Criteria Average Reward Reinforcement Learning

In many real world situations, it is natural to express the objective as making appropriate tradeoffs between different kinds of rewards. As a motivating example, consider a modified version of the Buridan's ass problem (Figure 1). In our version of this example, there is a 3x3 grid. The animal is originally in the center square and is hungry. Food is present at the two diagonally opposite squares as indicated in the figure. But if it moves towards one of the piles, the food in the other pile can be stolen. It has to compromise between the two competing goals of eating and guarding the food. We introduced the third criterion, which is to minimize the number of steps it walks.

As discussed before, static versions of the multi-criterion RL problems are studied in the literature (Gabor et al., 1998; Altman, 1999; Feinberg & Schwartz, 1995; Mannor & Shimkin, 2004). In particular, (Feinberg & Schwartz, 1995) uses a weighted optimization criterion, which we will be adopting. We divide the rewards into k reward

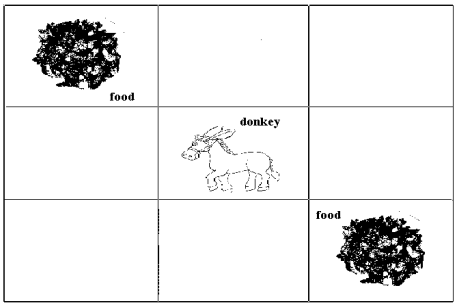

Figure 1. Buridan's ass problem in a 3 x 3 grid. The donkey needs to eat to survive while gurding the food and not walking too much.

types, and associate a weight with each reward type, which represents the importance of that reward. Given a weight vector $\vec{w} = \langle w_1, \ldots, w_k \rangle$ and an MDP as defined in the previous section, a new "weighted MDP" is defined where each reward $r_s(a)$ of type i is multiplied by the corresponding weight w_i. We call the average-reward per time step of the new weighted MDP for a policy, its "weighted gain." We want to find a policy that optimizes the weighted gain.

If the weights never change, the problem reduces to that of solving the weighted MDP and can be done using scalar reinforcement learning. But we are interested in the *dynamic* case, where the weights change in a hard-to-predict fashion and modeling the weights as part of the state is not realistic, e.g., this might require predicting an election result to decide which cereal to buy. In this case, we show that using vector-valued representations of average rewards and value functions would help in exploiting the structure of the MDP, and achieve faster learning.

Before we describe our approach, consider a few alternatives. Learning the best policy from scratch for each weight vector is possible but too inefficient. Another solution is to store all the policies learned so far with their weighted gains, initialize the policy to the one, say π, with the highest weighted gain, and to improve upon it. Unfortunately, the current weight vector could be very different from the weight vector of π. Hence π may not be the best for the current weight vector among all the stored policies.

To solve the problem, we need to understand how the weights affect the average rewards of policies. With a fixed policy, an MDP gives rise to a fixed distribution of sequences of rewards of each type. Hence the average reward of the weighted MDP under that policy is a weighted sum of the average rewards of the component MDPs, each corresponding to a single reward type. If we represent the average rewards of the component MDPs as an average-reward vector $\vec{\rho}$, the weighted gain of a fixed policy is $\vec{w} \cdot \vec{\rho}$. Similarly, the value function or bias of the weighted MDP under the given policy is a weighted sum of the value functions of component MDPs under that policy and can be

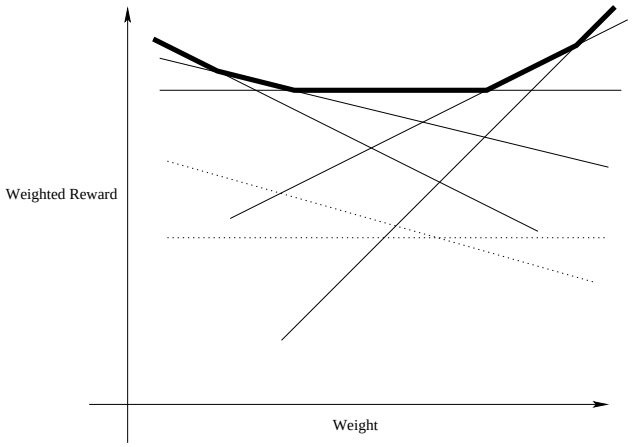

Figure 2. The weighted gains for a few policies for a weight vector with 2 components. The weighted gain of each policy is represented by a single straight line. The dark line segments represent the best weighted gains for any given weight. The dotted lines represent the weighted gains of the dominated policies.

expressed as $\vec{w} \cdot \vec{h}(.)$.

Let us, for simplicity, consider a weight vector $\vec{w}$ with two components such that the sum of the individual components is 1. Now the weighted gain of a given policy varies linearly with either weight (Figure 2). For each weight vector, the optimal policy is the one that maximizes the weighted gain. In the figure, the weighted gain of the optimal policy for any given weight is shown in dark. A policy which is optimal for some weight is called an "un-dominated" policy; the others are "dominated." The un-dominated policies trace a weighted gain function that is convex and piecewise linear. This would be a convex (bowl-shaped) piecewise planar surface in multiple dimensions[1].

The algorithm schema for dynamic multi-criteria reinforcement learning (DMCRL) is presented in Table 1. A key idea behind our approach is that we only need to learn and store the un-dominated policies, which are in general far fewer than all possible policies. We represent a policy π indirectly as a value function vector and an average reward vector $\vec{\rho}_\pi$, both of dimension k, the number of different reward types. Note that the value function vector is a function of states $\vec{h_\pi}(s)$ in the case of model-based learning and a function of state-action pairs $\vec{R_\pi}(s,a)$ in the case of model-free learning, while $\vec{\rho}_\pi$ is a constant. The policy π represented by the value functions is a greedy policy with respect to the weighted value function, and can be computed if we also store the weight vector $\vec{w_\pi}$ from which the value func-

[1]The reasoning here is similar to that of policies over belief states in POMDPs (Kaelbling et al., 1998).

tion has been learned (and the action models in the case of the model-based learning).

Suppose that Π represents the set of all stored policies. The question is how to choose an appropriate policy for a new weight vector $\vec{w}_{new}$. Here we exploit the fact that each policy π also stores its average reward vector $\vec{\rho}_{\pi}$. Inner product of these two vectors gives the weighted gain of π under the new weight vector. To maximize the weighted gain, we need to pick the policy π_{init} that maximizes the inner product.

$$\pi_{init} = Argmax_{\pi \in \Pi}\{\vec{w}_{new} \cdot \vec{\rho}_{\pi}\}$$

The value function and average reward vectors are initialized to those of π_{init} and are further improved with respect to the new weights $\vec{w}$ through vector-based reinforcement learning. Note that during this process, the greedy actions are taken with respect to $\vec{w}_{new}$, and not with respect to $\vec{w}_{\pi_{init}}$, the weights for which π_{init} may have been learned. Indeed, our algorithm actually does not store $\vec{w}_{\pi_{init}}$, as there is no need for it and the current weight vector $\vec{w}_{new}$ is the only one that matters. Thus, the trajectories, reward sequences, and value function vectors could deviate from those of π_{init} towards a policy that is better suited for the new weights. We only store the new value function and average reward vectors in Π if the weighted gain of the new policy with respect to $\vec{w}_{new}$ improves by more than a tunable parameter δ. The idea is that the optimal polices are going to be the same for many weight vectors and hence they have identical average reward vectors and weighted gains with respect to $\vec{w}_{new}$. Thus, they will not be duplicated. If each weight is given sufficient time to converge to the optimal policy, then only the un-dominated policies will be stored. Hence, although there are an infinite number of weight vectors, and an exponential number of policies, the number of stored policies may be small.

We could use any vector-based average reward reinforcement learning algorithm in step 2*c* of the algorithm. We verify our result empirically by using vectorized versions of R-learning and H-learning. In both the versions, the value functions, immediate rewards and the average rewards are vectors. In vector R-Learning, the agent chooses an action that maximizes the value of $\vec{w}.\vec{R}(s',a')$ and executes it and obtains the immediate reward $\vec{r}_{imm}$. The update equation is:

$$\vec{R}(s,a) \longleftarrow \vec{R}(s,a)(1-\beta) + \beta(\vec{r}_{imm} - \vec{\rho} + \vec{R}(s',a')) \quad (5)$$

where

$$a' = argmax'_a[\vec{w} \cdot \vec{R}(s',a')] \quad (6)$$

and $\vec{\rho}$ is the average reward vector which is updated using the equation:

$$\vec{\rho} \longleftarrow \vec{\rho}(1-\alpha) + \alpha(\vec{r}_{imm} + \vec{R}(s',a') - \vec{R}(s,a)) \quad (7)$$

Table 1. Algorithm schema for Dynamic Multi-Criteria Reinforcement Learning

1. Obtain the current weight vector $\vec{w}_{new}$
2. For the current weight compute, $\pi_{init} = \arg\max_{\pi \in \Pi}(\vec{w}_{new} \cdot \vec{\rho}_{\pi})$
   (a) Initialize the value function vectors of the states to those of π_{init}
   (b) Initialize the average reward vector to that of π_{init}
   (c) Learn the new policy π' through vector-based reinforcement learning
3. If $(\vec{w}_{new} \cdot \vec{\rho}_{\pi'} - \vec{w}_{new} \cdot \vec{\rho}_{\pi_{init}} > \delta$,add π' to the set of stored policies.

Since the transition probability models do not depend on the weights, they are not vectorized in H-Learning. The update equation for vector-based H-Learning is:

$$\vec{h}(s) \longleftarrow \vec{r}(s,a) + \sum_{s'=1}^{n} p_{s,s'}(a)\vec{h}(s') - \vec{\rho} \quad (8)$$

where

$$a = argmax\{\vec{w} \cdot (\vec{r}(s,a) + \sum_{s'=1}^{n} p_{s,s'}(a)\vec{h}(s')\} \quad (9)$$

and $\vec{\rho}$ is updated using

$$\vec{\rho} \longleftarrow \vec{\rho}(1-\alpha) + \alpha(\vec{r}(s,a) - \vec{h}(s) + \vec{h}(s')) \quad (10)$$

4. Implementation and Results

Having proposed the algorithm for DMCRL, we provide the empirical verification of its performance. We tested our algorithms on the modified version of Buridan's ass problem and a network routing domain. We explain the experimental setup and the results.

4.1. Buridan's ass domain

This modified version of the Buridan's ass domain is shown in Figure 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 neighboring squares in the eight directions. If the donkey moves away from the neighboring square of a food pile, there is a certain probability p_{stolen} with which the food is stolen. Food is regenerated once every N_{appear} time-steps. The donkey has to strike a compromise between minimizing the three different costs: hunger, lost food, and walking.

4.1.1. EXPERIMENTAL SETUP

A state is a tuple $\langle s, f, t \rangle$, where s stands for the square in which the donkey is present, f for food in the two piles, and t for the time since the donkey last ate food. If $t = 9$, it is not incremented and the donkey incurs a penalty of -1 per time step till it eats the food when t is reset to 0. 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. p_{stolen} is set to 0.9. N_{appear} is set to 10. The stolen penalty is -0.5 per plate and walking penalty is -1 per step.

The dynamic multi-criteria versions of R-Learning and the H-Learning presented in the previous sections were implemented. The weights were generated at random and the three components were normalized so that they add up to 1. For each weight, the programs were run for 100,000 time-steps. The agent learns for 1000 steps and is evaluated for the next 1000 steps. 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 small number of weights, the agent need not learn for a new weight vector, and instead can use a previously stored policy.

The correctness of the policies learned by both the agents were verified manually for the extreme cases of weights. 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. Hence, the donkey would 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 learned for different weights are presented in Figure 3.

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 3. Policies for R and H Learning corresponding to different weights

4.1.2. EXPERIMENTAL RESULTS

In this section, we present two graphs for each version of the algorithm, one showing the learning curve for a few weights and the other showing the number of steps required for convergence vs the number of weights. Figures [4, 5] present the learning curves for three weight vectors for dynamic multi-criteria R-learning and dynamic multi-criteria H-learning. The three weight vectors in Figures [4, 5] are $\langle 1,0,0 \rangle$,$\langle 0,1,0 \rangle$ and $\langle .33,.33,.33 \rangle$. It can be seen that the weighted gain $\vec{w} \cdot \vec{\rho}$ converges to zero in the first two cases for both the versions. 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.

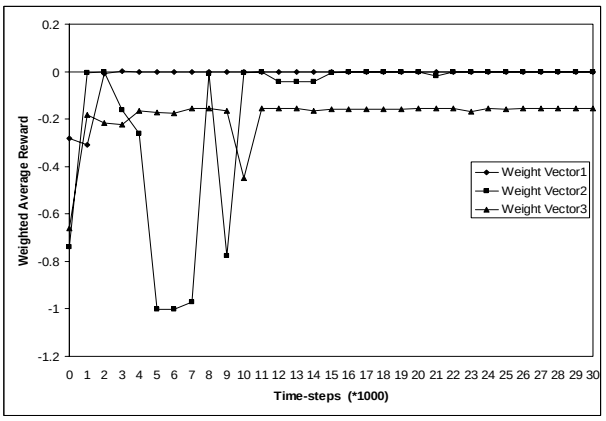

Figure 4. Learning curves for 3 weights using R-Learning

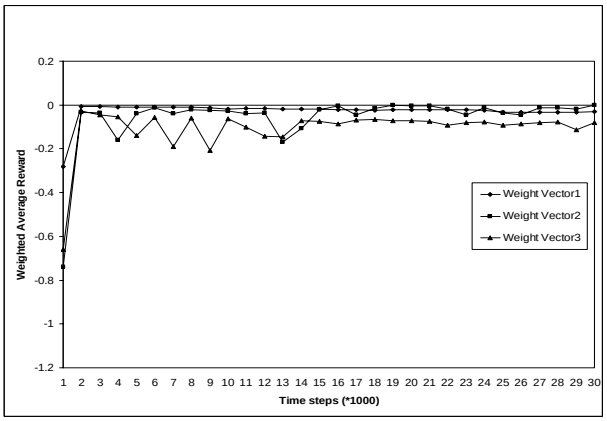

Figure 5. Learning curves for 3 weights using H-Learning

The second set of graphs for the model-free and model-based versions of the algorithms are shown in Figures [6,7]. Each of these figures compares our agent with one that learns from scratch for every weight. Each plot presents

the number of time-steps required to converge to the optimal policy against the number of weight vectors. For each weight vector, we record the weighted gain after every 1000 time-steps. We determine the convergence by looking at a window of 20 consecutive recordings. If their weighted gains lie within δ of each other in the 20 recordings, the algorithm is assumed to have converged. The plots show averages over 15 different runs of 100 random weight vectors each.

As can be observed, the number of steps to converge for the DMCRL agent drops down quickly in both H-Learning and R-Learning. The total number of policies learned was between 15 and 20 in both cases. In many cases, these are learned within the first 50 weight vectors. In contrast, the plots for the *learning-from-scratch* agents wander about at a much higher value of average number of steps for convergence.

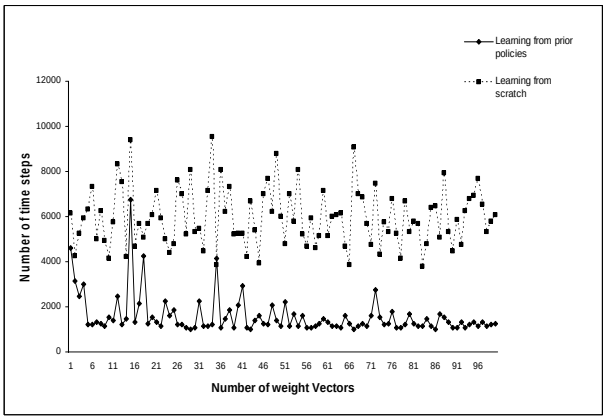

Figure 6. Convergence graph for R-Learning

Though previous work showed that the model-based learning performed better than the model-free learning (Tadepalli & Ok, 1998), the results were not significantly different for the two versions in our experiments. This is because the domain is almost deterministic and one sample of execution is as good as many. Hence explicit learning and using of models did not have a significant impact on the speed of learning.

4.2. Network Routing Domain

In this section, we describe the application of distributed versions of our algorithms to network routing. 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. At the heart of any routing protocol is the routing algorithm that determines the path of a packet from the source to the destination (Kurose & Ross, 2003). The problem of network routing was studied earlier in the reinforcement learning paradigm (e.g., (Tao et al., 2001;

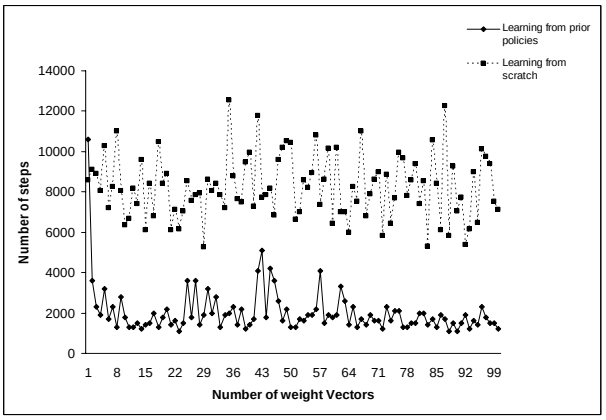

Figure 7. Convergence graph for H-Learning

Stone, 2000)). These approaches, however, focussed on a single criterion.

There are multiple criteria to evaluate the routing algorithms (Lu et al., 2003). We consider three of them. The end-to-end delay is the time required for a packet to travel from the source to the destination. The other two criteria are the loss of packets due to congestion or router/link failure and the power level associated with a node. Each criterion has a weight associated with it and the goal is to optimize the weighted gain.

4.2.1. Implementation details

The network that we used to test our algorithms is shown in Figure 8.

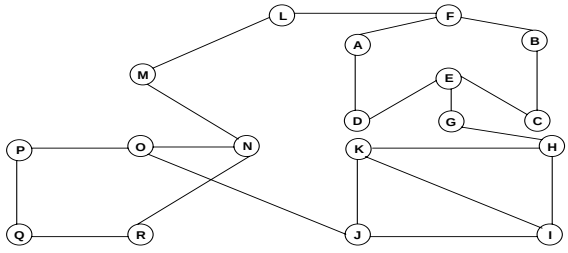

Figure 8. Network Model

The weights are generated uniformly at random. Each node generates some packets for random destinations. Every node sends a control packet that contains the value functions to its neighbors once every $T_{control}$ seconds. Upon receiving the packet, each node processes the packet. If it is

a data packet, it sends an acknowledgment to the source. If it is an acknowledgment, it updates the values and rewards. 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. The nodes explore for T_{learn} seconds and evaluate the policy for the next $T_{evaluate}$ seconds.

The immediate reward components, r_{ete}, r_{pl}, r_{pow} correspond 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 simulated time. Also for every packet that is lost, the immediate reward was -1. A node would wait for a certain time period T_{pl} to determine if the packet is lost. The power value received from the neighbor was used as the immediate reward for the action that chose that neighbor.

The "state" in this domain 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 is to be sent. The value function is represented in a distributed way, in that each node stores its value function for each destination node ($R_{curr\_node}(destination, neighbor)$ or $h_{curr\_node}(destination)$). The goal is to maximize the weighted gain computed over the entire network.

4.2.2. Experimental Results

In our experiments $T_{control}$ was set to 400, while T_{learn} and $T_{evaluate}$ were both set to 500. The set of actions is the set of neighbors to choose from. All the nodes accessed the global average reward using mutual exclusion constraints and updated them. The agent learned for 10,000 seconds of simulated time and then would read in a new weight vector. Since the models do not depend on weights, they were learned from scratch for the first weight and were incrementally updated for the later weights in the case of H-Learning. Both the methods followed an ε-*greedy* strategy with $\varepsilon = 0.10$.

The results were recorded after every 1000 seconds, where the agent learned for the first 500 seconds using ε-greedy strategy and was evaluated for the next 500 seconds using purely greedy strategy. We verified the policies manually for a few weights. The algorithm is assumed to converge if the weighted average rewards lie within δ of each other in 5 consecutive recordings. The convergence graphs for R-Learning and H-Learning are presented in Figures [9, 10]. The data were collected from 15 runs and averaged. For each run, the program was executed for 10 simulated days. The convergence curve was plotted for these 86 weight vectors. As before, the dynamic multi-criteria approach is compared to learning from scratch for each weight. As expected, in the multi-criteria approach, the convergence time for the few initial weights is high, but goes down quickly with the number of weights. New policies are learned in-

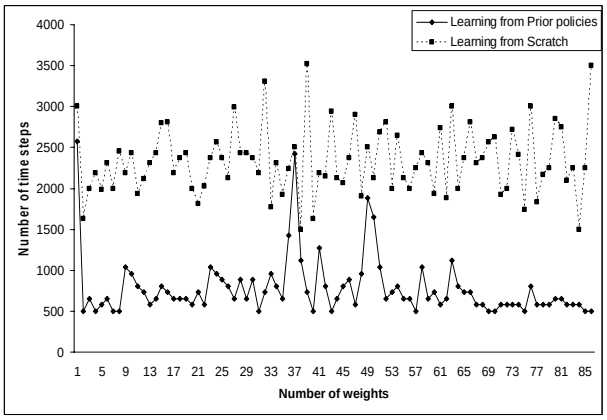

Figure 9. Convergence graph for R-Learning in the Network Routing Domain

frequently after about 60 weight vectors. The results are similar for both the model-free and model-based versions. In contrast, the number of steps for convergence for the *learning-from-scratch* agents is much higher.

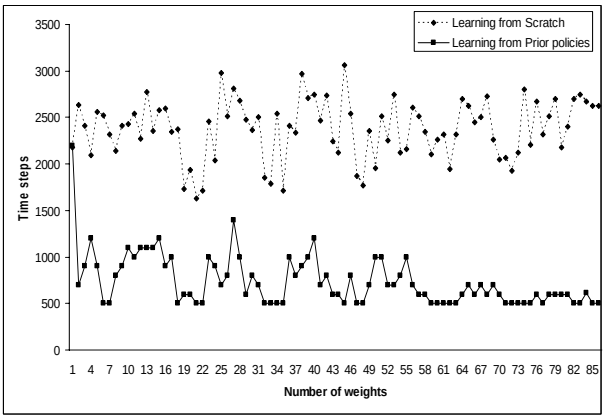

Figure 10. Convergence graph for H-Learning in the Network Routing domain

5. Conclusion and Future work

The basic premise of the paper is that many real-world problems have multiple objectives, where the importance of the objectives is time-varying. In such cases, the normal scalar based reinforcement learning techniques do not suffice. It becomes imperative that the value functions and rewards are decomposed and adapted to new preferences. We showed that by exploiting the structure of the MDP and introducing a vector-based approach, we can learn faster by storing the previously learned value functions and reusing them. Similar ideas on decomposing rewards and value functions have been explored in related work. For example, Russell and Zimdars decomposed the value functions so that they are more smoothly approximated (Russell &

Zimdars, 2003). Guestrin et al. used the decomposition idea to make multi-agent coordination tractable (Guestrin et al., 2001). This approach is similar to Parr's earlier work on "policy caches," which consists of decomposing a large MDP into smaller MDPs, solving them and combining the solutions (Parr, 1998).

An interesting direction for future research is to investigate the number of different weight vectors needed to learn all the optimal policies within a desired degree of accuracy. This in turn depends on the structure of the MDPs, which needs to be carefully characterized. Function approximation would be useful and important to scale the problems to large domains.

Another interesting problem is to infer the weights when the user is unable to provide them, but simply controls the agent in a near-optimal manner. This problem resembles the "inverse reinforcement learning" problem or "preference elicitation" problem (Ng & Russell, 2000; Boutilier, 2002). Chajewska et al. use a Bayesian approach to solve this problem and learn to predict the future decisions of the agent from the past decisions (Chajewska et al., 2001). More recently, Ng et al. formulate the problem of "apprenticeship learning," which includes learning from observations and reinforcements (Abbeel & Ng, 2004). Observed behavior is used to elicit the weights of the user, which are in turn used to find an optimal policy for those weights. It would be interesting to see if we can improve apprenticeship learning by storing previously learned policies and reusing them as in the current paper.

6. Acknowledgement

We are grateful to OPNET Technologies Inc. for providing us with their network simulator and support. This material is based upon work supported by the National Science Foundation under Grant No.IIS-0329278 and the support of DARPA under grant number HR0011-04-1-0005. We thank Tom Dietterich, Alan Fern, Neville Mehta, Aaron Wilson, Ronald Bjarnason and the anonymous reviewers for their valuable suggestions.

References

Abbeel, P., & Ng, A. (2004). Apprenticeship learning via inverse reinforcement learning. *In Proceedings of ICML-04*.

Altman, E. (1999). *Constrained markov decision processes*. Chapman and Hall. First edition.

Boutilier, C. (2002). A POMDP formulation of preference elicitation problems. *In Proceedings AAAI-02*.

Chajewska, U., Koller, D., & Ormoneit, D. (2001). Learning an agent's utility function by observing behavior. *In Proceedings of ICML-01*.

Feinberg, E., & Schwartz, A. (1995). Constrained markov decision models with weighted discounted rewards. *Mathematics of Operations Research*, *20*, 302–320.

Gabor, Z., Kalmar, Z., & Szepesvari, C. (1998). Multi-criteria reinforcement learning. *In Proceedings of ICML-98*.

Guestrin, C., Koller, D., & Parr, R. (2001). Multiagent planning with factored MDPs. *In Proceedings NIPS-01*.

Kaelbling, L. P., Littman, M., & Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101*.

Kurose, J. F., & Ross, K. W. (2003). *Computer networking - a top-down approach featuring the internet*. Pearson Education. Second edition.

Lu, Y., Wang, W., Zhong, Y., & Bhargava, B. (2003). Study of distance vector routing protocols for mobile ad hoc networks. *In Proceedings of PerCom-03*.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, *22*, 159–195.

Mannor, S., & Shimkin, N. (2004). A geometric approach to multi-criterion reinforcement learning. *JMLR*, *5*, 325–360.

Ng, A. Y., & Russell, S. (2000). Algorithms for inverse reinforcement learning. *In Proceedings of ICML-00*.

Parr, R. (1998). Flexible decomposition algorithms for weakly coupled markov decision problems. *In Proceedings UAI-98*.

Puterman, M. L. (1994). *Markov decision processes*. J.Wiley and Sons.

Russell, S., & Zimdars, A. L. (2003). Q-decomposition for reinforcement learning agents. *In Proceedings of ICML-03*.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. *In Proceedings of ICML-93*.

Stone, P. (2000). TPOT-RL applied to network routing. *In Proceedings of ICML-00*.

Tadepalli, P., & Ok, D. (1998). Model-based average reward reinforcement learning. *AI Journal*, *100*, 177–223.

Tao, N., Baxter, J., & Weaver, L. (2001). A multi-agent policy-gradient approach to network routing. *In Proceedings of ICML-01*.

White, D. (1982). Multi-objecticve infinite-horizon discounted markov decision processes. *Journal of Mathematical Analysis and Applications*, *89*, 639–647.