Transfer in Variable-Reward Hierarchical Reinforcement Learning

Neville Mehta **Sriraam Natarajan** **Prasad Tadepalli** **Alan Fern**

School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97333
{mehtane,natarasr,tadepall,afern}@eecs.oregonstate.edu

Abstract

We consider the problem of transferring learned knowledge among Markov Decision Processes that share the same transition dynamics but different reward functions. In particular, we assume that reward functions are described as linear combinations of reward features, and that only the feature weights vary among MDPs. We introduce Variable-Reward Hierarchical Reinforcement Learning (VRHRL), which leverages a cache of learned policies to speed up learning in this setting. With suitable design of the task hierarchy, VRHRL can achieve better transfer than its non-hierarchical counterpart.

1 Introduction

Most work in Reinforcement Learning (RL) addresses the problem of solving a single Markov Decision Process (MDP) defined principally by its dynamics (a transition function) and a reward function. The focus on solving individual MDPs makes it difficult, if not impossible, to learn cumulatively, i.e., to transfer useful knowledge from one MDP to another. In this paper, we consider *variable-reward transfer learning* where the objective is to speed up learning in a new MDP by transferring experience from previous MDPs that share the same dynamics but different reward functions. In particular, we assume that reward functions are weighted linear combinations of reward features, and that the *reward weights* vary across MDPs.

MDPs that share the same dynamics but have different reward structures arise in many contexts. For example, different agents operating in a domain such as driving might have different preference functions although they are all constrained by the same physics [1]. Even when the reward function can be defined objectively, e.g., winning as many games as possible in chess, usually the experimenter needs to provide other "shaping rewards", such as the value of winning a pawn, to encourage Reinforcement Learning (RL) systems to do useful exploration. Each such shaping reward function can be viewed as defining a different MDP in the same family. Reward functions can also be seen as goal specifications for agents such as robots and Internet search engines. Alternatively, different reward functions may arise externally based on difficult-to-predict changes in the world, e.g., raising gas prices or declining interest rates that warrant lifestyle changes.

For such variable-reward MDPs, previous work [4] has shown how to leverage the reward structure in order to usefully transfer value functions, effectively speeding-up learning. In this paper, we extend this work to the hierarchical setting, where a single task hierarchy is applicable across the entire variable-reward MDP family. The hierarchical setting provides advantages over the flat RL case, allowing for selective transfer at multiple levels of the hierarchy which can significantly speed up learning.

We demonstrate our results in a simplified real-time strategy (RTS) game domain. In this domain, peasants accumulate gold and wood resources from gold mines and forests respectively, and quell any enemies that appear inside their territory. The reward features include bringing in gold and wood, and damaging the enemy.

The rest of the paper is organized as follows. We provide a synopsis of Variable-Reward RL (VRRL) in Section 2, followed by an explanation of Hierarchical Reinforcement Learning (HRL) in Section 3. In Section 4, we elucidate the central idea of this paper. We present experimental results in Section 5, and conclude in Section 6 by outlining areas of future work.

2 Variable-Reward Reinforcement Learning

A Semi-Markov Decision Process (SMDP) $\mathcal{M}$ is a tuple (S, A, P, r, t), where S is a set of states, A is a set of temporally extended actions, and the transition function $\mathrm{P}(s'|s,a)$ gives the probability of entering state s' after taking action a in state s. The functions $r(s,a)$ and $t(s,a)$ are the expected reward and execution time respectively for taking action a in state s.

Given an SMDP, the average reward or *gain* ρ^π of a policy π is defined as the ratio of the expected total reward to the expected total time for N steps of the policy from any state s as N goes to infinity. In this work, we seek to learn policies that maximize the gain. The *average-adjusted reward* of taking an action a in state s is defined as $r(s,a) - \rho^\pi t(s,a)$. The limit of the total expected average-adjusted reward starting from state s and following policy π as the number of steps goes to infinity is called its *bias* and denoted by $h^\pi(s)$.

In this work, we assume a linear reward function $r(s,a) = \sum_{i=1}^{k} w_i \, r_i(s,a)$, where the $r_i(s,a)$ are reward features, and the w_i are reward weights. In this linear reward setting, the gain and bias are linear in the reward weights $\vec{w}$, that is $\rho^\pi = \vec{w} \cdot \vec{\rho}^\pi$, and $h^\pi(s) = \vec{w} \cdot \vec{h}^\pi(s)$, where the i components of $\vec{\rho}^\pi$ and $\vec{h}^\pi(s)$ are the gain and bias respectively with respect to the ith reward feature.

We consider transfer learning in the context of families of MDPs that share all components except for the reward weights. After encountering a sequence of such MDPs, the goal is to transfer the accumulated experience to speed up learning in a new MDP given its unique reward weights. For example, in our RTS domain, we would like to consider varying the reward weighting for bringing in units of wood, gold, and damaging the enemy, but still leverage prior experience.

A previous approach to this problem [4] is based on the following idea. The reward weights are assumed to be between 0 and 1. Since the above value functions are linear in the reward weights, policies can be represented indirectly as a set of parameters of these linear functions, i.e., the gain and the bias functions are learned in their vector forms, where the components correspond to the expected value of reward features when the weight is 1. Thus, the set of optimal policies for different weights forms a convex and piecewise linear average reward and bias functions. If a single policy is optimal for different sets of weights, it suffices to store one set of parameters representing this policy. Furthermore, if $\mathcal{C}$ represents a set of all optimal policies, then given a new weight vector $\vec{w}_{\text{new}}$, we might expect

the policy $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}}\{\vec{w}_{\text{new}} \cdot \vec{\rho}^{\pi}\}$ to provide a good starting point for learning. Thus, transfer learning is conducted by initializing the bias and gain vectors to those of π_{init} and then further optimizing it via average-reward reinforcement learning (ARL). After convergence, the newly learned bias and gain vectors are only stored in $\mathcal{C}$ if the gain of the new policy with respect to $\vec{w}_{\text{new}}$ improves by more than a satisfaction threshold δ. With this approach, if the optimal polices are the same or similar for many weight vectors, only a small number of policies are stored and significant transfer could be achieved.

3 Hierarchical Reinforcement Learning

In MAXQ-based Hierarchical Average-Reward Reinforcement Learning (HARL) [2, 5], the original MDP $\mathcal{M}$ is split into sub-SMDPs $\{\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$, where each sub-SMDP represents a subtask. Subtasks that represent the actions of the original MDP are called primitive; every other subtask is called composite. Solving the composite root task $\mathcal{M}_0$ solves the entire MDP $\mathcal{M}$. The task hierarchy is represented as a directed acyclic graph known as the *task graph* that represents the subtask relationships. The task hierarchy for the RTS domain is shown in Figure 1(b). The leaf nodes are the primitive subtasks.

Formally, each composite subtask $\mathcal{M}_i$ is defined by the tuple $(\mathrm{B}_i, \mathcal{A}_i, \mathrm{G}_i)$:

- **State Abstraction** B_i: A function that selects a subset of the original state variables to comprise an abstracted state space sufficient for $\mathcal{M}_i$'s value function to represent the local policy. For example, Goto(l)'s abstracted state space ignores every state variable but the location of the agent.
- **Actions** $\mathcal{A}_i$: The set of subtasks that can be called by $\mathcal{M}_i$. For example, Root can call either Harvest(l), Deposit, Attack, or `idle`.
- **Termination predicate** G_i: The predicate that partitions the subtask's abstracted state space into active and terminated states. When $\mathcal{M}_i$ is terminated, control returns back to the calling subtask. The probability of the eventual termination of a subtask (other than the root task) is 1. For example, Deposit is terminated if the peasant is not carrying anything.

A subtask is *applicable* iff it is not terminated. The root task's termination predicate is always false (an unending subtask). Primitive subtasks have no explicit termination condition (they are always applicable), and control returns to the parent task immediately after their execution.

A local policy π_i for the subtask $\mathcal{M}_i$ is a mapping from the states abstracted by B_i to the child tasks of $\mathcal{M}_i$. A hierarchical policy π for the overall task is an assignment of a local policy π_i to each sub-MDP $\mathcal{M}_i$. A *hierarchically optimal policy* is a hierarchical policy that has the best gain. Unfortunately, hierarchically optimal policies tend to be context-sensitive and transfer-resistant in that the best policy for the lower level subtasks depend on the higher level tasks, e.g., the best way to exit the building might depend on which airport one wants to drive to. To enable subtask optimization without regard to the supertasks, Dietterich introduced the notion of *recursive optimality*, which optimizes the policy for each subtask assuming that its children's policies are in turn recursively optimized [2]. Seri and Tadepalli [5] show that recursive optimality coincides with hierarchical optimality when the hierarchy satisfies a condition called "result distribution invariance" (RDI) which means that the terminal state distribution of a task does not depend on the policy used to accomplish it. Often, the tasks in the hierarchy can be designed in such a way that RDI is satisfied. In the above example, we might design a hierarchy with each possible exit out of building as a different subtask so that the correct exit is chosen based on the airport one wants to get to.

4 Variable-Reward Hierarchical Reinforcement Learning Framework

In the non-hierarchical (flat) setting, every policy is represented by a monolithic value function and one gain. This means that even if the rewards only change for a small subset of the states, but the resultant policy belongs in the convex set of optimal policies, the entire value function will need to be cached. A monolithic value function could also lend itself to negative transfer where the initial biased value function takes longer to converge to the optimal policy than if it was unbiased to begin with.

We seek to incorporate the transfer mechanism into a hierarchical framework to benefit from value function decomposition. Every subtask has a local value function, and local changes in rewards can be better handled. For instance, if none of the reward variations affect navigation, the Goto subtask need only learn its local value function once; perfect transfer across the MDPs is achievable for this subtask (and consequently everything below it in the task hierarchy).

In the Variable-Reward HRL framework, every subtask i (but the root) stores the total expected reward vector $\vec{V}_i$ during that subtask, and the expected duration t_i of the subtask for every state. The bias vector $\vec{h}_i$ can be calculated by subtracting from $\vec{V}_i$, t_i times the global gain vector $\vec{\rho}$. Storing the bias vector indirectly in a form that is independent of the gain allows for the transfer of any subtree of the task hierarchy across MDPs with different global gain vectors. Storing the value functions as vectors facilitates transfer across different MDPs in the variable-reward family just as in the non-hierarchical variable-reward RL. For action selection, the objective is to maximize the weighted gain (the dot product of the gain vector with the weight vector).

More formally, the value function $\vec{V}_i(s)$ for a non-root subtask i represents the total expected reward during task i starting from state s for a recursively optimal policy π. Hence, the value function decomposition for a non-root subtask satisfies the following equations:

$$\vec{V}_i(s) = \vec{r}(s) \quad \text{if } i \text{ is a primitive subtask} \tag{1}$$

$$= 0 \quad \text{if } s \text{ is a terminal/goal state for } i \tag{2}$$

$$= \vec{V}_j(\mathrm{B}_j(s)) + \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s,j) \cdot \vec{V}_i(s') \quad \text{otherwise,} \tag{3}$$

$$\text{where } j = \operatorname*{argmax}_a \left\{ \vec{w} \cdot \left(\vec{V}_a(\mathrm{B}_a(s)) - \vec{\rho}^{\pi} \cdot t_a(\mathrm{B}_a(s)) + \mathrm{E}\left[\vec{V}_i(s') - \vec{\rho}^{\pi} \cdot t_i(s') \right] \right) \right\} \tag{4}$$

The primitive subtasks just keep track of the reward received (equation 1), and the elapsed time for the atomic MDP actions they represent. For the composite tasks, the value of every terminal state is 0 (equation 2). The value of a non-terminal state in a composite task is the sum of the total reward achievable by the best child task followed by the total reward till the completion of the task (equation 3), where the best child task is chosen to maximize the weighted bias (equation 4). In general, this action choice leads to a recursively optimal policy which coincides with the hierarchically optimal policy when the task hierarchy satisfies result distribution invariance. Similar Bellman equations can be written for computing the total expected time of a subtask, which is a scalar.

Since the root task never terminates, we cannot store the total expected reward as we do for the terminating composite subtasks. Instead, the value function $\vec{V}_{root}(s)$ for the root task

directly represents the bias of state s:

$$\vec{V}_{\text{root}}(s) = \max_j \left\{ \vec{w} \cdot \left(\vec{V}_j(\mathrm{B}_j(s)) - \vec{\rho}^{\pi} \cdot \vec{t}_j(\mathrm{B}_j(s)) + \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s,j) \cdot \vec{V}_{\text{root}}(s') \right) \right\} \quad (5)$$

The VRHRL agent has three components: the task hierarchy with the current subtask value functions and the global gain, the task stack, and a cache of previously learned optimal policies $\mathcal{C}$ that comprise the convex piecewise function. Note that the policies in the cache are indirectly represented by the subtask value and duration functions, and the global gain. The policy cache $\mathcal{C}$ is specific to the hierarchical transfer mechanism while the other components are part of a basic hierarchical agent.

Initially, the agent starts out with an empty policy cache. The agent proceeds to learn a recursively optimal policy π_1 for the first weight $\vec{w}_1$. When the agent senses a new weight $\vec{w}_2$, it first caches π_1 (the subtask value functions and the global gain achieved for $\vec{w}_1$) into the policy cache. Next, it determines $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}} \{\vec{w}_2 \cdot \vec{\rho}^{\pi}\}$ (which in this case is π_1), and initializes its subtask value functions and global gain based on π_{init}. It then improves the value functions using vectorized version of model-based hierarchical RL, which works as follows.

At any level of the task hierarchy, this algorithm chooses an exploratory subtask or a greedy one according to Equation 4. It learns the transition model $\mathrm{P}(s'|s,a)$ for each subtask by counting the number of resulting states for each state-task pair and estimating the transition probabilities. In doing so, the states are abstracted using the abstraction function defined at that subtask so that the transition probabilities are represented compactly. Every time a child task terminates, Equations 2 and 3 are used to update the total expected reward vector $\vec{V}_i$ of task i, and similar equations are used to update the duration function $t_i(s)$ (a scalar). The global average-reward vector $\bar{r}^{\pi}$ is updated at the root level by averaging the immediate reward adjusted by the difference in the bias vectors of the two end states. Additionally, the average-time scalar $\bar{t}^{\pi}$ is updated by averaging the immediate duration of the root task's subtasks.

$$\bar{r}^{\pi} \leftarrow (1-\alpha)\bar{r}^{\pi} + \alpha(\vec{V}_a(s) + \vec{V}_{\text{root}}(s') - \vec{V}_{\text{root}}(s)) \quad (6)$$

$$\bar{t}^{\pi} \leftarrow (1-\alpha)\bar{t}^{\pi} + \alpha\, t_a(s) \quad (7)$$

where α is the learning rate, and s and s' are the states before and after executing the highest level subtask a of the root task. The updates in equations 6 and 7 are performed only a is selected greedily. Finally, $\vec{\rho}^{\pi} \leftarrow \bar{r}^{\pi} / \ \bar{t}^{\pi}$. α is decayed down to an asymptotic value of 0 during the course of the learning algorithm.

On sensing a new weight $\vec{w}_3$, the agent only caches the learned hierarchical policy π_2 for $\vec{w}_2$ if $\vec{w}_2 \cdot \vec{\rho}^{\pi_2} - \vec{w}_2 \cdot \vec{\rho}^{\pi_{\text{init}}} > \delta$. If this condition is not satisfied, then the newly learned policy is not sufficiently better than the cached version. However, if it is satisfied, then the newly learned policy must be added to the cache. When adding π_2 to the policy cache, we could just store the value function of every subtask in the task hierarchy. However, although the hierarchical policy has changed, many of the local subtask policies could still be the same. To leverage this fact, for every subtask being stored, we check the policy cache to see if any of the previously stored versions of the subtask is similar to the current one; if so, then we need only store a reference to that previously stored version. Two versions of a subtask are similar if none of the values for the vector components of the value and duration functions for any state differ by more than ε, a similarity constant. Once caching is complete, the policy that maximizes the weighted gain w.r.t. $\vec{w}_3$ is chosen from the policy cache for initialization. This process is repeated for every new weight encountered by the system. Thus, every weight change is accompanied by the internal process of caching and initialization for the agent.

5 Experimental Results

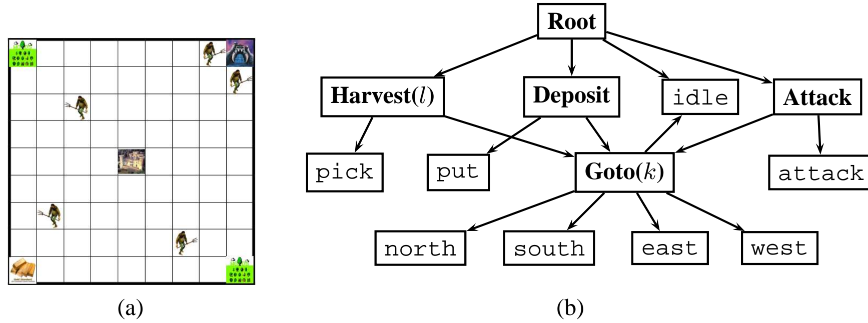


Figure 1: Simplified RTS domain and the corresponding task hierarchy.

For our experiments, we consider a simplified RTS game shown in Figure 1(a). It is a grid world that contains peasants, the peasants' home base, resource locations (forests and goldmines) where the peasants can harvest wood or gold, and an enemy base which can be attacked. The state variables in this domain are the peasant's location, what it is currently carrying (gold or wood or nothing), and the resources available at each of the resource locations. The primitive actions available to a peasant are moving one cell to the `north`, `south`, `east`, and `west`, `pick` a resource, `put` a resource, `attack` the enemy base, and `idle` (no-op). In this domain, the resources get regenerated stochastically and are inexhaustible. The enemy also appears stochastically and stays until there is a peasant in the same cell that is attacking it.

The basic reward setup is as follows: -1 for every action, $+100$ and $+200$ for depositing wood and gold at the home base respectively, and $+300$ for damaging the enemy. Each of the weight components $w_i \in [0, 1)$; they dictate the relative value of collecting the various resources and attacking the enemy. For example, if the peasant is at its home base with a load of gold (state s_1) and is executing the `put` action, the reward vector $\vec{r}(s_1, \texttt{put}) = (-1, 0, 200, 0)$. If the weight $\vec{w} = (0, 0.5, 0.2, 0.9)$, then the scalar reward $= \vec{w} \cdot \vec{r}(s_1, \texttt{put}) = 40$.

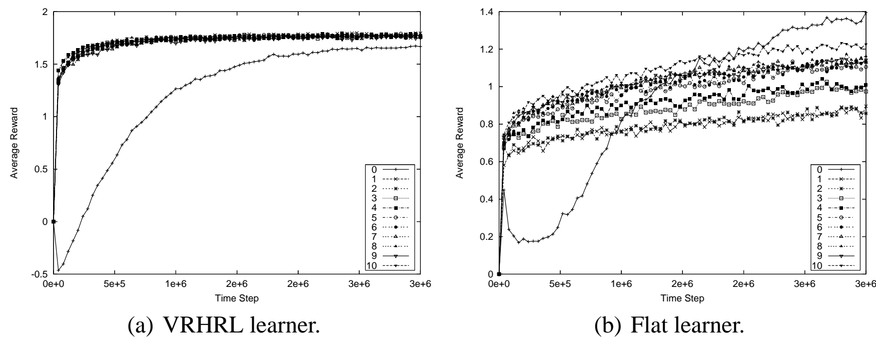
The following results are based on a single-peasant game in a 25×25 grid with 3 forests cells, 2 goldmines, a home base, and an enemy base. Figures 2(a) and 2(b) show the learning curves for both the flat and VRHRL learners for one test weight (0.5 for every component) after having seen 0 through 10 previous training weights, averaged over 10 different weight sets. All training weight vectors are generated by drawing every weight component from a Uniform(0,1) distribution.

Both the flat and hierarchical learners use epsilon-greedy exploration. For both learners, the learning curve for the test weight given no training weights (i.e., an empty policy cache) is slow. However, after learning on one training weight, the VRHRL agent converges very quickly for the test weight. This could be attributed to immediate and perfect transfer of composite subtasks such as Goto. The flat learner exhibits negative transfer, i.e., the initialization to the policy for the training weight is hurting the convergence for the test weight. This is unsurprising given that currently we always attempt to transfer past experience, even when experience is limited. VRHRL seems to avoid such negative transfer by clustering experience into similar subtasks. Obviously, as both learners see more training weights, their performance improves.

Since the MDPs have the same dynamics, the learners do not need to relearn the transition

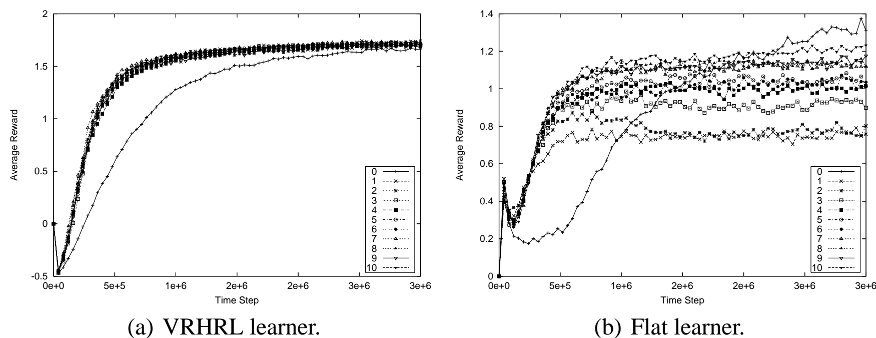
models from scratch when the reward weights change. To determine how much of the transfer benefits can be attributed to the transition models, Figures 3(a) and 3(b) show the results of repeating the above experimental setup with one crucial difference – no policy is ever cached. Thus, the only transfer here is due to the transition models. Obviously, reusing the learned transition models is helpful to both agents. However, the VRHRL learner does perform better with the added mechanism of caching. On the other hand, the flat learner does slightly worse with caching due to negative transfer.

As another measure of transfer, let F_Y be the area between the learning curve and its optimal value for problem Y with no prior learning experience on X, and $F_{Y|X}$ be the area between the learning curve and its optimal value for problem Y given prior training on X. The *transfer ratio* is defined as $F_Y / F_{Y|X}$. When evaluating this metric, the base experimental setup is the same as above except that now the test weight vector for every set is generated randomly just like the training vectors, and the transfer ratio is averaged over these 10 sets. The test vectors need not be the same (as in the case of plotting the learning curves) because the transfer ratio is a metric that is independent of the absolute convergence value of the learning curves. Figure 4 shows the transfer ratios for the flat and VRHRL learners, with and without caching. It is apparent that the VRHRL has benefited the most from the transfer of subtask value functions.



(a) VRHRL learner. (b) Flat learner.

Figure 2: Learning curves with policy caching.



(a) VRHRL learner. (b) Flat learner.

Figure 3: Learning curves without policy caching.

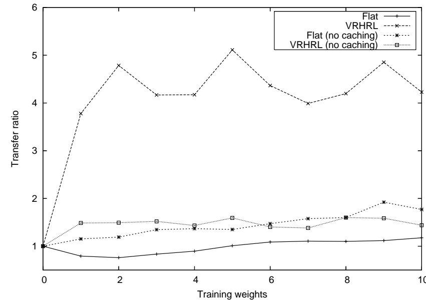


Figure 4: Transfer ratios.

6 Conclusions and Future Work

In this paper, we showed that hierarchical reinforcement learning can accelerate transfer across variable-reward MDPs more so than in the non-hierarchical case. Our results are in the model-based setting with the advantage that the models need not be relearned when the rewards change. While we assumed that the reward weights are given, it is easy to learn them from the scalar rewards since the scalar reward is linear in the reward weights. Extending these results to MDP families with slightly different dynamics would be interesting. Another possible direction is an extension to shared subtasks in the multi-agent setting [3].

Acknowledgments

We thank the reviewers for their helpful comments and suggestions. The idea of using transfer ratio to evaluate the effectiveness of transfer originated in discussions with Stuart Russell and Leslie Kaelbling. We gratefully acknowledge the support of Defense Advanced Research Projects Agency under DARPA grant FA8750-05-2-0249. Views and conclusions contained in this document are those of the authors and do not necessarily represent the official opinion or polices, either expressed or implied, of the US government or of DARPA.

References

[1] P. Abbeel and A. Ng. Apprenticeship Learning via Inverse Reinforcement Learning. In *Proceedings of the ICML*, 2004.

[2] T. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 9:227–303, 2000.

[3] N. Mehta and P. Tadepalli. Multi-Agent Shared Hierarchy Reinforcement Learning. *ICML Workshop on Rich Representations in Reinforcement Learning*, 2005.

[4] S. Natarajan and P. Tadepalli. Dynamic Preferences in Multi-Criteria Reinforcement Learning. In *Proceedings of the ICML*, 2005.

[5] S. Seri and P. Tadepalli. Model-based Hierarchical Average Reward Reinforcement Learning. In *Proceedings of the ICML*, pages 562–569, 2002.