Mach Learn (2008) 73: 289–312
DOI 10.1007/s10994-008-5061-y

Transfer in variable-reward hierarchical reinforcement learning

Neville Mehta · Sriraam Natarajan · Prasad Tadepalli · Alan Fern

Received: 22 February 2007 / Revised: 21 April 2008 / Accepted: 8 May 2008 /
Published online: 3 June 2008
Springer Science+Business Media, LLC 2008

Abstract Transfer learning seeks to leverage previously learned tasks to achieve faster learning in a new task. In this paper, we consider transfer learning in the context of related but distinct *Reinforcement Learning* (RL) problems. In particular, our RL problems are derived from Semi-Markov Decision Processes (SMDPs) that share the same transition dynamics but have different reward functions that are linear in a set of reward features. We formally define the transfer learning problem in the context of RL as learning an efficient algorithm to solve any SMDP drawn from a fixed distribution after experiencing a finite number of them. Furthermore, we introduce an online algorithm to solve this problem, Variable-Reward Reinforcement Learning (VRRL), that compactly stores the optimal value functions for several SMDPs, and uses them to optimally initialize the value function for a new SMDP. We generalize our method to a hierarchical RL setting where the different SMDPs share the same task hierarchy. Our experimental results in a simplified real-time strategy domain show that significant transfer learning occurs in both flat and hierarchical settings. Transfer is especially effective in the hierarchical setting where the overall value functions are decomposed into subtask value functions which are more widely amenable to transfer across different SMDPs.

Keywords Hierarchical reinforcement learning · Transfer learning · Average-reward learning · Multi-criteria learning

Editors: Daniel L. Silver, Kristin Bennett, Richard Caruana.

N. Mehta (✉) · S. Natarajan · P. Tadepalli · A. Fern
School of Electrical Engineering and Computer Science, Oregon State University, Corvallis 97330, USA
e-mail: mehtane@eecs.oregonstate.edu

S. Natarajan
e-mail: natarasr@eecs.oregonstate.edu

P. Tadepalli
e-mail: tadepall@eecs.oregonstate.edu

A. Fern
e-mail: afern@eecs.oregonstate.edu

1 Introduction

Most work in transfer learning is in the supervised setting where the goal is to improve the classification performance by exploiting inter-class regularities. In this paper, we consider transfer learning in the context of Reinforcement Learning (RL), i.e., learning to improve performance over a family of Semi-Markov Decision Processes (SMDPs) that share some structure. In particular, we introduce the problem of *Variable-Reward Transfer Learning* where the objective is to speed up learning in a new SMDP by transferring experience from previous MDPs that share the same dynamics but have different rewards. More specifically, reward functions are weighted linear combinations of reward features, and only the *reward weights* vary across different SMDPs. We formalize this problem as converging to a set of policies which can solve any SMDP drawn from a fixed distribution in close to optimal fashion after experiencing only a finite sample of SMDPs.

SMDPs that share the same dynamics but have different reward structures arise in many contexts. For example, while driving, different agents might have different preferences although they are all constrained by the same physics (Abbeel and Ng 2004). 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 RL systems to do useful exploration. Any such shaping reward function can be viewed as defining a different SMDP 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., rising gas prices or declining interest rates that warrant lifestyle changes. There is a large literature on multi-criteria decision problems where each criteria corresponds to an individual reward signal, and one of the standard ways of approaching this problem is to solve these decision problems with respect to the single linear combination reward signal. An example of such a reward decomposition would be in a logistics domain where there are trade-offs between fuel consumption, delivery time, and number of drivers. Different companies might have different coefficients for these items and would require different solutions. Another example is that of trading where the prices of different commodities might change from day to day but the actions involved in trading them all have the same dynamics.

In this work, we use the Average-reward RL (ARL) framework where the goal is to optimize the average reward per step. However, our general approach can be easily adapted to both the discounted and the total-reward settings. The key insight behind our method is that the value function of a fixed policy is a linear function of the reward weights. Variable-Reward Reinforcement Learning (VRRL) takes advantage of this fact by representing the value function as a vector function whose components represent the expectations of the corresponding reward features that occur during the execution of the policy. Given a new set of reward weights and the vectored value function of a policy stored in a cache (value function repository), it is easy to compute the value function and average reward of that policy for the new weights. VRRL initializes the value function for the new weights by comparing the average rewards of the stored policies and choosing the best among them. It then uses a vectorized version of an ARL algorithm to further improve the policy for the new weights. If the average reward of the policy is improved by more than a satisfaction constant γ via learning, then the new value function is stored permanently in the cache. We derive an upper bound for the number of policies that will be stored in the worst case for a given γ and the maximum values of different reward weights.

Hierarchical Reinforcement Learning (HRL) makes it possible to scale RL to large tasks by decomposing them into multiple smaller subtasks (Dietterich 2000; Sutton et al. 1999;

Andre and Russell 2002). In the MAXQ framework of HRL, the overall value function is additively decomposed into different task-specific value functions. We extend VRRL to the MAXQ setting where a single task hierarchy is used for all SMDPs. Vectored value functions are stored for the subtasks, and represent the subtask policies. Given a new weight vector, our method initializes the value functions for each subtask by comparing the overall average rewards for the stored value functions and choosing the best. If the average reward improves during learning, then every subtask whose value function changes significantly is cached. In this hierarchical version, we expect the subtask policies to be optimal across a wide spectrum of weights because the number of reward components affecting a particular subtask's value function is normally less than that affecting the non-hierarchical value function, and this results in significant transfer.

We demonstrate our approach empirically in a simplified real-time strategy (RTS) game domain. In this domain, a peasant must accumulate various resources (wood, gold, etc.) from various locations (forests, gold mines, etc.), quell any enemies that appear inside its territory, and avoid colliding into other peasants. The reward features are associated with bringing in the resources, damaging the enemy, and collisions. The actual value of these features to the agent is determined by the feature weights which are different in each SMDP. The goal is to learn to optimize the average reward per time step. We show that there is significant transfer in both flat and hierarchical settings. The transfer is much more prominent in the hierarchical case mainly because the overall task is decomposed into smaller subtasks which are optimal across a wide range of weights.

To cope with the huge state space (36.6×10^{15} states), we employ value-function approximation. The combination of hierarchical task decomposition and value function approximation allows our method to be applicable to a broad class of problems. Moreover, we show that standard perceptron-style learning can be used to induce the weight vector from a scalar reward function and the reward features.

The rest of the paper is organized as follows. In Sect. 2, we introduce the Variable Reward Transfer Learning problem followed by our algorithm for it, its analysis, and some experimental results in the RTS domain. Section 3 is a review of our version of hierarchical reinforcement learning. In Sect. 4, we describe our Variable-Reward Hierarchical Reinforcement Learning (VRHRL) algorithm, and present experimental results in a more complex version of the RTS domain. The related work is discussed in Sect. 5, and conclusions and future work in Sect. 6.

2 Variable-reward transfer learning

In this section, we start with some background on Average-reward Reinforcement Learning. We then introduce the *variable-reward transfer learning problem*, and present an approach to solve it in a non-hierarchical setting.

2.1 Average-reward reinforcement learning

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 information relevant to future actions is said to satisfy the Markov property. Thus, given the current state, all future states are independent of past states.

A semi-Markov Decision Process (SMDP) $\mathcal{M}$, an extension of the Markov Decision Process, 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 $P(s,a,s') = \Pr(s'|s,a)$ 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 the expected execution time respectively for taking action a in state s. A policy π is a mapping from states S to actions A, specifying which action to execute in every state. Given an SMDP, the *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, and these are called *gain-optimal* policies.

The intuition behind ARL 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 $\rho^\pi T(s,a)$. Given a single SMDP and a policy π, we define the *average-adjusted reward* of taking an a in state s as $R(s,a) - \rho^\pi T(s,a)$, i.e., the difference between the expected immediate reward for the state-action pair (s,a) and the expected total reward (on average) for the policy π in the duration of a in s. The limit of the total expected average-adjusted reward for π starting from state s and action a as the number of steps goes to infinity is called its *bias* and denoted by $h^\pi(s)$

$$h^\pi(s) = \lim_{N\to\infty} \mathrm{E}\left[\sum_{i=0}^{N}(r_i - \rho^\pi t_i)\right].$$

Under some reasonable conditions on the SMDP structure (Puterman 1994), there exist a scalar ρ and a real-valued function $h = h^{\pi^*}$ that satisfy the following Bellman equation:

$$h(s) = \max_{a\in A}\left(R(s,a) - \rho T(s,a) + \sum_{s'\in\mathcal{S}} \Pr(s'|s,a)h(s')\right). \tag{1}$$

The policy π^* that selects actions to maximize the right-hand side of (1) attains the optimal gain $\rho^{\pi^*} \geq \rho^\pi$ over all policies π.

H-Learning (Tadepalli and Ok 1998) is a model-based reinforcement learning method that solves for the optimal policy. It learns the action models P, R, and T, and uses them to update the h-value of the current state s by applying the right-hand side of (1) at every step. In SMDPs, where actions take a variable amount of time, the gain ρ is not learned directly. Instead, we must learn the average reward ρ_r^π, a measure of the average reward acquired during the execution of a temporally extended primitive action, and the average time ρ_t^π, a measure of the average duration of a primitive action, following policy π:

$$\rho_r^\pi \leftarrow (1-\alpha)\rho_r^\pi + \alpha(R(s,a) + h(s') - h(s)), \tag{2}$$

$$\rho_t^\pi \leftarrow (1-\alpha)\rho_t^\pi + \alpha T(s,a), \tag{3}$$

$$\alpha \leftarrow \alpha/(1+\alpha). \tag{4}$$

The parameter ρ_r^π is updated by the exponential moving average of the immediate rewards. The immediate reward is adjusted by the difference in the h values of s' and s to neutralize the effect of exploratory actions on the states visited by the agent. This adjusted immediate reward gives an unbiased sample of average reward of a single action of the SMDP. The parameter ρ_t^π is updated by the exponential moving average of the immediate durations. These updates are only performed when a greedy action is taken in state s, resulting in an immediate reward $R(s, a)$, taking time $T(s, a)$, and transitioning to the next state s'. Since gain is defined as the total reward obtained divided by the total time taken, ρ^π is updated with ρ^π_r/ρ^π_t. The learning rate α starts at 1 and decays down to an asymptotic value of 0 during the course of learning.

2.2 Variable-reward transfer learning problem

A family of variable-reward SMDPs is defined as $(S, A, P, \mathbf{f}, T)$, where S, A, P, and T are the same as before, but $\mathbf{f}$ is an n-dimensional vector of binary reward feature components $\langle f_1, \ldots, f_n\rangle$. A weight vector $\mathbf{w} = \langle w_1, \ldots, w_n\rangle$ specifies a particular SMDP (S, A, P, R, T) in the family where $R = \mathbf{w} \cdot \mathbf{f}$. Hence, n denotes the size of the minimal specification of any SMDP in the family, and is called the dimension of the SMDP family. The binary feature vector[1] indicates whether or not a particular component is active for (s, a); the weight vector's components are arbitrary real numbers that indicate the importance of the corresponding feature components. All the SMDPs in the variable reward family share the same states, actions, state transition function, and the expected execution times of actions but may have different reward functions based on the particular weight vector.

We now proceed with a formal definition of the Variable Reward Transfer Learning. We begin with some preliminary definitions.

Definition 1 A variable-reward transfer learning problem is a pair $(\mathcal{F}, D)$ where $\mathcal{F}$ is a variable-reward SMDP family, and D is an unknown distribution over weight vectors $\mathbf{w}$.

Note that D defines a distribution over SMDPs in the family. The goal of the transfer learner is to quickly find an optimal policy for SMDPs drawn from the family (based on the weight vectors) given an opportunity to interact with it by taking actions and receiving rewards. A simple strategy would be to treat each SMDP independently, and apply some reinforcement learning algorithm to solve it. While this approach cannot be improved upon if we need to solve only one SMDP in the family, it is inefficient when we are interested in solving a sequence of SMDPs drawn from the distribution over the SMDP family.

After solving a sequence of SMDPs drawn from the distribution, it would seem that we should be able to transfer the accumulated experience to a new SMDP drawn from the same distribution. An obvious improvement to solving each SMDP separately is to learn models for the state-transition function P and the execution times T, collectively called the action models, and share them across different SMDPs. Since all the SMDPs in the family share the same P and T, their models can be incrementally learned from different SMDPs, and hence speeding up learning from successive SMDPs. In the next section, we will see that more efficient approaches are possible.

Definition 2 A γ-optimal policy is any policy whose gain is at most γ less than that of the optimal policy.

Definition 3 An ϵ, γ-approximate cover for a variable-reward transfer learning problem $(\mathcal{F}, D)$ is a set of policies $\mathcal{C}$ such that, given an SMDP M chosen from $\mathcal{F}$ according to D, $\mathcal{C}$ contains a γ-optimal policy for M with probability at least $1 - \epsilon$.

[1]The restriction to being a binary vector is for explanatory purposes only; in practice, this could be an arbitrary real-valued vector.

We refer to an ϵ, γ-approximate cover as an ϵ, γ-cover for short. This is an approximation in two senses. First, the policies in the cover might be up to γ worse than the optimal. Second, there is some probability ϵ of it not containing a γ-optimal policy for an SMDP in the family $\mathcal{F}$. Ideally, we would like a variable-reward transfer learner to produce an ϵ, γ-cover for any problem $(\mathcal{F}, D)$ after learning from a small number of training SMDPs drawn according to D. However, since the training SMDPs are randomly chosen, we allow the learner to fail with a small probability δ. This is similar to probably approximately correct learning, where we only guarantee to find an approximately correct hypothesis with a high probability.

Definition 4 A learner $\mathcal{A}$ is a *finite sample transfer learner* if for any variable-reward transfer learning problem $(\mathcal{F}, D)$ of dimension n, and any set of parameters ϵ, γ and δ, there exists a finite bound $F(\epsilon, \delta, \gamma, n)$ such that, with probability at least $1 - \delta$, it finds an ϵ, γ-cover for $(\mathcal{F}, D)$ after learning from $F(\epsilon, \delta, \gamma, n)$ SMDPs drawn according to D. In this case, we say that the learner has *sample complexity* $F(\epsilon, \delta, \gamma, n)$. Further, if $F(\epsilon, \delta, \gamma, n)$ is polynomial in n, $\frac{1}{\gamma}$, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$, we call $\mathcal{A}$ a *sample-efficient transfer learner*. If the run-time of $\mathcal{A}$, counting each call to the learner $\mathcal{L}$ as $O(1)$, is polynomial in n, $\frac{1}{\gamma}$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and the sample size $F(\epsilon, \delta, \gamma, n)$, then we say that $\mathcal{A}$ is a *time-efficient transfer learner*.

The above definition can be generalized to a variety of SMDP families that share different kinds of structure. In this paper, we restrict ourselves to the variable-reward family.

2.3 Variable-reward reinforcement learning

Our approach to variable-reward reinforcement learning (VRRL) exploits the structure of the variable-reward SMDP family by caching value functions and reusing them. We begin with the following theorem which states that the value function for any fixed policy is linear in its reward weights.

Theorem 1 *Let $\mathcal{M} = (S, A, P, R, T)$ be an SMDP in the variable-reward SMDP family with a reward weight vector* $\mathbf{w}$, *and π be a policy. The gain ρ^π is $\mathbf{w} \cdot \boldsymbol{\rho}^\pi$, and the bias of any state s is $h^\pi(s) = \mathbf{w} \cdot \mathbf{h}^\pi(s)$, where the i^{th} components of $\boldsymbol{\rho}^\pi$ and $\mathbf{h}^\pi(s)$ are the gain and bias respectively with respect to the i^{th} reward feature for the policy π.*

Proof (sketch) The result follows directly from the fact that the immediate rewards for the SMDP are linear in the reward weights, and the bias and the average rewards are based on the sums of the immediate rewards. □

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 all the weight components are 1. In the following section, we show that the set of optimal policies for different weights forms a convex and piecewise linear gain 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.

2.3.1 Algorithm description

To understand the intuition behind our approach, consider a gain vector $\boldsymbol{\rho}$ for a particular policy. Plotting $\mathbf{w} \cdot \boldsymbol{\rho}$ with respect to the weight components of $\mathbf{w}$ would result in a hyperplane. Every policy generates one such hyperplane in the weight space; Fig. 1 demonstrates this in the case of a 2-component weight vector.

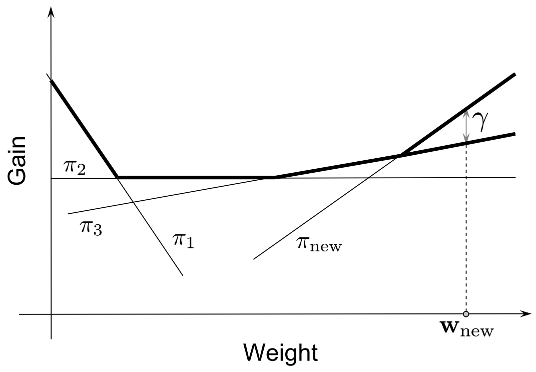

Fig. 1 Every *line* in the above plot represents a single policy whose gain varies linearly with the reward weights. The *dark lines* represent the gain of the optimal policy as a function of weight

The bold piecewise linear and convex function in the figure represents the best weighted gain for each possible weight. Extended to multiple dimensions, this would correspondingly be a convex piecewise planar surface.[2] Thus, when a new weight is considered, the learner might start off with the policy that registers the highest weighted average-reward, represented by a point in the highlighted convex function in Fig. 1. Initializing the value function with that of the dominant policy for the current weight vector will assure that the agent would learn a policy that is at least as good.

Let $\mathcal{C}$ represent the set of all optimal policies which are currently stored. Given a new weight vector $\mathbf{w}_{\text{new}}$, we might expect the policy $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w}_{\text{new}} \cdot \boldsymbol{\rho}^{\pi})$ to provide a good starting point for learning. Our transfer learning algorithm works by initializing the bias and gain vectors to those of π_{init} and then further optimizing them 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 $\mathbf{w}_{\text{new}}$ improves by more than a satisfaction threshold γ. 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 can be achieved (Natarajan and Tadepalli 2005). The algorithm for Variable-reward Reinforcement Learning is presented in Algorithm 1.

Note that the counters i and c updated and compared in the algorithm are primarily for the purposes of theoretical analysis. In practice, we keep looping through the algorithm and adding to the cache when necessary as long as new weight vectors are experienced.

We could use any vector-based average reward reinforcement learning algorithm in step 12 of the algorithm. In this work, we employ the vectorized version of H-learning algorithm. Recall that in H-learning, the action models are learned and used to update the h-value of a state. In the vectorized version, the $\mathbf{h}$-values, the reward models (the binary feature values $\mathbf{f}(s,a)$), and the gain $\boldsymbol{\rho}$ are all vectors. The greedy action a is now defined as

$$a \leftarrow \underset{b}{\text{argmax}}\left(\mathbf{w} \cdot \left(\mathbf{f}(s,b) - \boldsymbol{\rho} T(s,b) + \sum_{s'} \Pr(s'|s,b)\mathbf{h}(s')\right)\right).$$

[2]The reasoning here is exactly the same as in the POMDPs, where the value function is a convex piecewise linear function over the belief states (Kaelbling et al. 1998).

Algorithm 1 VRRL sketch

```
1: $i \leftarrow 1$
2: $c \leftarrow 0$
3: $\mathcal{C} \leftarrow \emptyset$
4: $\pi_{\text{init}} \leftarrow \emptyset$
5: **repeat**
6:    Obtain the current weight vector $\mathbf{w}$
7:    **if** $\mathcal{C} \neq \emptyset$ **then**
8:       Compute $\pi_{\text{init}} \leftarrow \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w} \cdot \boldsymbol{\rho}^{\pi})$
9:       Initialize the value function vectors of the states
10:      Initialize the gain to $\boldsymbol{\rho}^{\pi_{\text{init}}}$
11:   **end if**
12:   Learn the new policy $\pi'$ through vector-based RL
13:   **if** $(\mathcal{C} = \emptyset)$ **or** $(\mathbf{w} \cdot \boldsymbol{\rho}^{\pi'} - \mathbf{w} \cdot \boldsymbol{\rho}^{\pi_{\text{init}}} > \gamma)$ **then**
14:      $\mathcal{C} \leftarrow \mathcal{C} \cup \pi'$
15:      $c \leftarrow 0$
16:      $i \leftarrow i + 1$
17:   **else**
18:      $c \leftarrow c + 1$
19:   **end if**
20: **until** $c \geq \frac{1}{\epsilon} \ln \frac{(i+1)^2}{\delta}$
21: **return** $\mathcal{C}$
```

After the execution of a, the value of the state is then updated as

$$\mathbf{h}(s) \leftarrow \max_{b} \left(\mathbf{f}(s,b) - \boldsymbol{\rho}\, T(s,b) + \sum_{s'} \Pr(s'|s,b)\mathbf{h}(s') \right).$$

For the gain $\boldsymbol{\rho}$, the only change from the non-vectorized version is in the update for the average reward $\boldsymbol{\rho}_r^{\pi}$ which is done as

$$\boldsymbol{\rho}_r^{\pi} \leftarrow (1-\alpha)\boldsymbol{\rho}_r^{\pi} + \alpha(\mathbf{f}(s,a) - \mathbf{h}(s) + \mathbf{h}(s')).$$

We can also apply this vectorizing trick to R-learning, the model-free average-reward reinforcement learning algorithm (Schwartz 1993). Some experimental results based on this are presented in (Natarajan and Tadepalli 2005).

Though we expound on the application of our variable-reward concepts to the average-reward setting, we could easily extend the idea to using weighted total reward in a total-reward setting, or the weighted discounted reward in a discounted-reward setting. In both these settings, the value function of any fixed policy is a linear function of the reward weights. Hence we could store the value function in a vectorized form just as in the average-reward case. In the average-reward setting, the gain vector provides a convenient way to find the best policy in the cache for a given set of reward weights. In the total-reward and the discounted-reward settings, we need to keep track of the expected returns for the start state distribution to serve a similar purpose. This is going to be in a vectorized form whose dimension is the number of the reward components. Its inner product with the reward weights gives the expected returns of any policy, and can be used to determine the initial policy as in step 8.

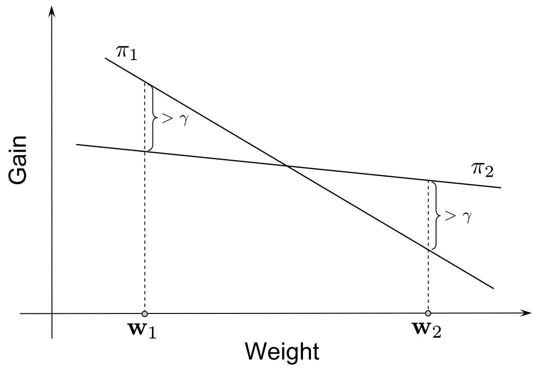

Fig. 2 Two policies π_1 and π_2 learned for weights $\mathbf{w}_1$ and $\mathbf{w}_2$ respectively

It should be noted that the algorithm presented here transfers over seamlessly to any representation of policies, including value-function approximation, as long as the representation is the same during retrieval (step 8) and caching (step 14). This is simply because the algorithm caches the parameters of the representation as a proxy for the policy itself.

2.3.2 Theoretical analysis

We now derive an upper bound on the number of policies stored by the VRRL algorithm and use it to derive the sample complexity.

Theorem 2 *Assuming that the vector-based RL finds optimal policies for each weight, the number of policies learned by VRRL is upper-bounded by $O((\frac{Wn}{\gamma})^n)$, where n is the number of components of the weight vector, W is the maximum range of the weight components, and γ is the satisfaction parameter.*

Proof Let π_1 and π_2 be any two policies learned by our algorithm. Recall that the average rewards of the policies are linear functions of the reward weights as shown in Fig. 2). Let the gain vectors of the two policies be $\boldsymbol{\rho}^{\pi_1}$ and $\boldsymbol{\rho}^{\pi_2}$ respectively. Let $\mathbf{w}_1$ and $\mathbf{w}_2$ be the weights at which the policies π_1 and π_2 were learned. Since π_1 and π_2 are optimal for $\mathbf{w}_1$ and $\mathbf{w}_2$, we know that

$$\mathbf{w}_1 \cdot \boldsymbol{\rho}^{\pi_1} - \mathbf{w}_1 \cdot \boldsymbol{\rho}^{\pi_2} \geq 0,$$
$$\mathbf{w}_2 \cdot \boldsymbol{\rho}^{\pi_2} - \mathbf{w}_2 \cdot \boldsymbol{\rho}^{\pi_1} \geq 0.$$

If π_1 was learned before π_2 then π_2 must have been judged better than π_1 at $\mathbf{w}_2$ by our algorithm or else it would not have been stored. Hence,

$$\mathbf{w}_2 \cdot \boldsymbol{\rho}^{\pi_2} - \mathbf{w}_2 \cdot \boldsymbol{\rho}^{\pi_1} > \gamma. \tag{5}$$

Similarly, if π_1 was learned after π_2, we will have

$$\mathbf{w}_1 \cdot \boldsymbol{\rho}^{\pi_1} - \mathbf{w}_1 \cdot \boldsymbol{\rho}^{\pi_2} > \gamma. \tag{6}$$

Since at least one of (5) and (6) are true, adding the two left-hand sides gives us

$$\begin{aligned}
&(\mathbf{w}_2 - \mathbf{w}_1) \cdot (\boldsymbol{\rho}^{\pi_2} - \boldsymbol{\rho}^{\pi_1}) > \gamma \\
&\quad\Longrightarrow \quad |\mathbf{w}_2 - \mathbf{w}_1||\boldsymbol{\rho}^{\pi_2} - \boldsymbol{\rho}^{\pi_1}| > \gamma \\
&\quad\Longrightarrow \quad |\mathbf{w}_2 - \mathbf{w}_1| > \frac{\gamma}{|\boldsymbol{\rho}^{\pi_2} - \boldsymbol{\rho}^{\pi_1}|} \\
&\quad\Longrightarrow \quad |\mathbf{w}_2 - \mathbf{w}_1| > \frac{\gamma}{\max_{i,j}(|\boldsymbol{\rho}^{\pi_j} - \boldsymbol{\rho}^{\pi_i}|)} > \frac{\gamma}{\sqrt{n}}.
\end{aligned}$$

The above equation implies that the weight vectors for which distinct policies are stored should be at least at a distance of $\frac{\gamma}{\sqrt{n}}$ from each other.[3] Let the maximum range of the weight space, i.e., the difference between the highest and the lowest weight for any reward component be W. Hence, the maximum number of stored policies N is bounded by the number of points that can be packed in a hypercube of side W, where the distance between any two points is $\geq \frac{\gamma}{\sqrt{n}}$.

We estimate an upper bound on this quantity as follows: Suppose there are N such points in the hypercube. We fill the volume of the hypercube by surrounding each point by a hypersphere of radius $\frac{\gamma}{2\sqrt{n}}$. No two such hyperspheres will overlap since the centers of the hyperspheres are at least at a distance of $\frac{\gamma}{\sqrt{n}}$ from each other. Hence, the volume of the hypercube divided by the volume of the hypersphere will upper-bound the number of hyperspheres. Since the volume of hypersphere of radius r is $\frac{r^n \pi^{\lfloor \frac{n}{2} \rfloor}}{\lfloor \frac{n}{2} \rfloor !}$ (Weeks 1985),

$$N \leq \frac{W^n \lfloor \frac{n}{2} \rfloor !}{(\frac{\gamma}{2\sqrt{n}})^n \pi^{\lfloor \frac{n}{2} \rfloor}} \leq O\left(\left(\frac{Wn}{\gamma}\right)^n\right).$$

□

Corollary 1 *The VRRL algorithm has a sample complexity bounded by* $O(\frac{N}{\epsilon} \ln \frac{N}{\delta})$ *where* N *is the upper-bound in the previous theorem.*

Proof Algorithm 1 is a rough sketch of the VRRL algorithm. Notice that for the learner to terminate in stage i, it should have passed through $m_i = \frac{1}{\epsilon}(2\ln(i+1) + \ln \frac{1}{\delta})$ randomly chosen test weight vectors without learning a new policy. We first bound the probability of the learner terminating with a non-ϵ, γ-cover in stage i. Note that a new policy is not learned only when the current policy cache produces a γ-optimal policy. A non-ϵ, γ-cover produces a γ-optimal policy on a random SMDP with probability at most $1-\epsilon$. The probability that all m_i weight vectors lead to γ-optimal policies is at most $(1-\epsilon)^{m_i} < e^{-\epsilon m_i} = e^{\ln \frac{\delta}{(i+1)^2}} = \frac{\delta}{(i+1)^2}$.

The probability that the learner terminates with some non-ϵ, γ-cover in some stage is therefore bounded by $\sum_{i=1}^{\infty} \frac{\delta}{(i+1)^2} < \int_{i=0}^{\infty} \frac{\delta}{(i+1)^2} di < \delta$.

From Theorem 2, we know that the learner learns at most N policies. The value of i is upper-bounded by N. Each time i is incremented, $m_i = O(\frac{1}{\epsilon} \ln \frac{(N+1)^2}{\delta})$ SMDPs are tested. Thus, we have a sample complexity of $O(\frac{N}{\epsilon} \ln \frac{N}{\delta})$. □

[3] Every component (dimension) of $\boldsymbol{\rho}$ is between 0 and 1 because it represents a feature expectation. The maximum distance between two such n-dimensional points is $\sqrt{n}$.

Note that VRRL is not sample-efficient since N is exponential in n, the natural parameter of the SMDP family. However, it is easy to see that VRRL runs in time linear in the sample size and polynomial in n (not counting the cost of solving the SMDPs, and assuming that the policy cache is indexed efficiently for a constant-time retrieval). Thus, according to Definition 4, VRRL is a time-efficient transfer learner.

This worst-case sample complexity bound suggests that the algorithm scales poorly in the number of reward components. While we may be able to improve our bounds with a tighter analysis, we argue that in many practical domains the number of reward components is small. VRRL is highly effective in such domains in transferring knowledge across different SMDPs. We provide empirical evidence for these claims in the later sections.

In light of this worst-case result, it is interesting to consider whether the algorithm can achieve better sample complexity for problems $(\mathcal{F}, D)$ where there exists a small set of policies that are sufficient to cover the problem. It turns out that even if a problem $(\mathcal{F}, D)$ has a small ϵ, γ-cover, the algorithm may require an arbitrarily large number of samples to find such a cover. To see this, consider a distribution D that places a probability mass of $1 - \epsilon - \epsilon'$ on weight vector $\mathbf{w}_1$, a probability mass of ϵ' on $\mathbf{w}_2$, and the remaining probability mass of ϵ/K on each of K weight vectors $\mathbf{w}'_1, \ldots, \mathbf{w}'_K$. It is possible to construct a variable-reward SMDP family $\mathcal{F}$ such that, given any pair of weight vectors generated according to D, the optimal policy for one is not γ-optimal for the other. Hence, the ϵ, γ-cover of such a problem $(\mathcal{F}, D)$ consists of the optimal policies for $\mathbf{w}_1$ and $\mathbf{w}_2$ (all the remaining weight vectors together have a probability mass of ϵ), and is of size 2. However, making ϵ' arbitrarily small and K arbitrarily large will cause the algorithm to require arbitrarily many samples to generate an ϵ, γ-cover for this problem.

The above example shows that a small cover does not necessarily guarantee a small sample complexity. This result does not seem to be specific to our algorithm but is a consequence of the fact that certain problem distributions are not exploitable in a transfer context. In the above example, there are many inherently different MDPs, each generated with a very small probability—this is a pathological scenario for any transfer learner.

If we place additional constraints on a policy cover, it is possible to show that our algorithm is efficient when there exists a small cover.

Definition 5 An ϵ, γ-cover for $(\mathcal{F}, D)$ is p-constrained if for each policy in the cover, with at least probability p, it is γ-optimal for a randomly drawn weight vector from D.

With this definition in hand, we can show the following result.

Theorem 3 *Let $(\mathcal{F}, D)$ be a variable-reward transfer learning problem. If there exists a p-constrained $\epsilon, \gamma/2$-cover for $(\mathcal{F}, D)$ with M policies, then with probability at least $1 - \delta$, the above algorithm will store an ϵ, γ-cover after at most $\frac{1}{p} \ln \frac{M}{\delta}$ samples.*

Proof Let $\mathcal{C}$ be the $\epsilon, \gamma/2$-cover assumed in the theorem. Consider a policy π in the cover and a weight vector w for which π is $\gamma/2$-optimal. If π' is the optimal policy for w, it follows that π' is γ-optimal for any weight vector for which π is $\gamma/2$-optimal. Thus, if we can guarantee that for each π in the cover, we sample a weight vector for which it is $\gamma/2$-optimal then the set of policies stored for those weights will effectively cover all of the weight vectors that $\mathcal{C}$ covers. Hence the set of learned policies will be an ϵ, γ cover for $(\mathcal{F}, D)$.

It remains to bound the number of samples required to guarantee that, with high probability, we get a set of weights such that each policy in $\mathcal{C}$ is $\gamma/2$-optimal for at least one weight. Given m samples, the probability that a single policy is not $\gamma/2$-optimal for any of the m samples is no more than $(1-p)^m$. Using the union bound, the probability that at least one policy is not $\gamma/2$-optimal for any sample is bounded by $M(1-p)^m$. We want this probability to be less than δ, i.e. $M(1-p)^m \leq \delta$. Solving for m gives the bound of the theorem. □

Note that although this bound may look like it is only logarithmic in M, it will in fact at least scale as $M \ln M$. To see this, note that we must have $pM < 1$ which means that $1/p$ is at least as large as M. This result shows that if there is a small cover such that the probability of each policy being optimal for a problem is not too small then we can get an efficient sample complexity. This agrees with the intuition that transfer learning is most useful in situations with a small number of inherently distinct SMDP types, each of which is not too unlikely to be experienced.

2.4 Experimental results

We present results of the performance of the VRRL system within the simplified RTS domain shown in Fig. 3. It is a map with a fixed number of peasants, the peasants' home base, multiple resource sites where the peasants can harvest resources, and an enemy base which can be attacked when it appears. The state variables in this domain are the locations of the peasants, what they are currently carrying (gold, wood, ..., nothing), the availability of resources at each of the resource locations, and the enemy's status (present or absent). The peasants can move one cell to the north, south, east, and west, pick a resource from a resource site, put a resource at the home base, attack the enemy base, and idle (no-op); the probability of failure for an action is 0.05. Resources are generated stochastically at the sites with a probability of 0.5. The probability of the appearance of an enemy base is 10^{-4}, and it persists until it is attacked. Due to the lack of scalability of non-hierarchical learning, these performance curves are based on a 25×25 RTS map with 1 peasant, 5 fixed resource sites (2 resources), a home base, and an enemy base. The rate of ϵ-greedy exploration is 0.1.

The reward feature vector has components associated with dropping off each of the resources, enemy elimination, and a time-step penalty that provides shaping to the flat learner. Theoretically, each of the weight components $w_i \in (-\infty, \infty)$. Empirically, we defined a set of seed values that we shuffle to generate the training and testing weights. These seed values are chosen to make one component dominate, and this is consequently reflected in

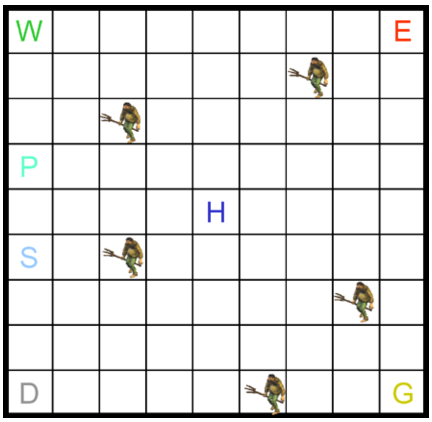

Fig. 3 The RTS domain with 5 peasants, multiple resources (W, G, ...), a home base H, and an enemy base E

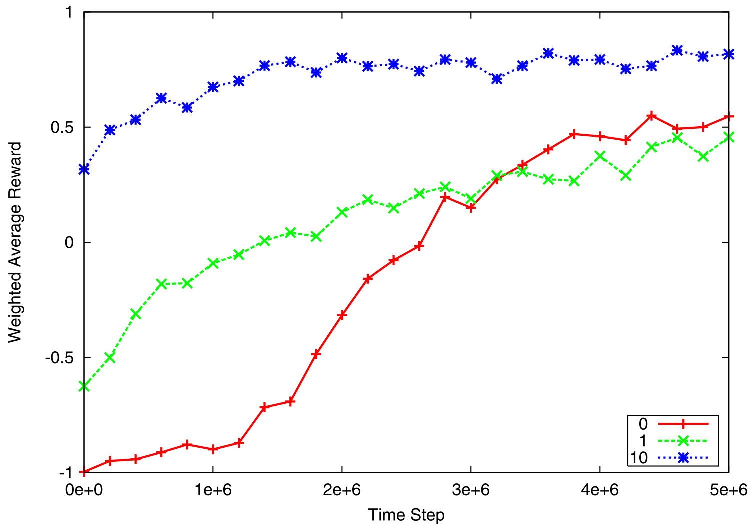

Fig. 4 Performance of VRRL on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials)

the policies learned. For a concrete illustration of how the reward feature and weight vectors generate the scalar reward, consider that the peasant is at its home base carrying gold (state s_1), it is executing put, and the immediate reward feature vector $\mathbf{f}(s_1, \text{put}) = (0, 1, 0, 1)$. If the current weight vector $\mathbf{w} = (10, 50, 30, -10)$, then the immediate scalar reward $R(s_1, \text{put}) = \mathbf{w} \cdot \mathbf{f}(s_1, \text{put}) = 40$.

Figure 4 shows the learning curves for the VRRL learner. All curves are averaged across 10 trials. The experiments are designed to show the performance of the learner on a particular test weight after having seen 0 through 10 training weights. Curve i represents the performance on the test weight having already seen i training weights; for the sake of clarity, the plots only show the learning curves for $i = 0, 1, 10$. Since the variable-reward framework is designed principally to deal with dynamically changing weights, we evaluate the performance on the test weight vector by incrementally introducing training weight vectors. Curve 0 measures the performance of the algorithm on the test weight, given no prior experience. Next, one training weight is introduced, and the optimal policy for this weight is learned and cached. Curve 1 now measures the performance on the test weight vector given the single training weight. When the second training weight is introduced, a cached policy is retrieved for initialization, and a new policy is learned; this new policy is cached if it is substantially better than the initializing policy. It is important to note that none of the learning done during performance evaluation (on the test weight vector) spills over into the training phases; the policy learned for the test weight is *never* cached.

From the results, we can observe that the learning curve for the test weight given no prior training weights (i.e., an empty policy cache) is the slowest to converge in both figures. However, after accumulating the 10 training weight vectors, the VRRL agent demonstrates a high jump-start[4] and quicker convergence for the test weight. With only one training weight vector, VRRL exhibits some negative transfer, that is, the initialization to the policy learned

[4]Jump-start is defined as the immediate benefit via transfer without any learning in the new setting.

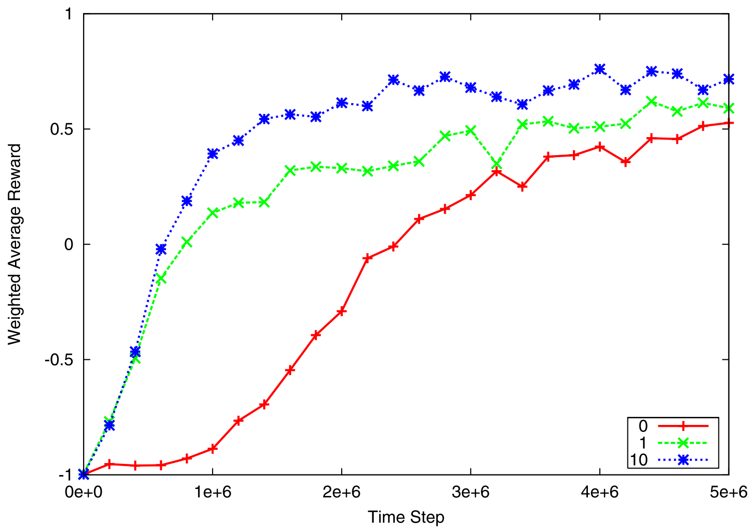

Fig. 5 Performance of the flat learner without the VRRL policy-caching mechanism on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials)

for the first training weight hurts the convergence for the test weight. This is unsurprising given that currently we always attempt to transfer past experience even when the experience is limited.

Since the MDPs have the same dynamics, the flat learner does not need to relearn the transition model from scratch when the reward weights change. Figure 5 shows the results of repeating the VRRL experimental setup with one crucial difference—no policy is ever cached. Instead, the only transfer here is due to the transfer of transition models. In this case, we do observe a slight speed-up in convergence but no jump-start.

3 Hierarchical reinforcement learning

The MAXQ value function decomposition is an elegant approach to exploit the hierarchical task structure typically present in large MDPs (Dietterich 2000). It facilitates learning separate value functions for subtasks which are composed to compute the value function for the supertask. The MAXQ approach has been simplified and adapted to a hierarchical average-reward reinforcement learning (HARL) setting (Seri and Tadepalli 2000).

In HARL, the original SMDP $\mathcal{M}$ 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 SMDP are called primitive; every other subtask is called composite. Solving the composite root task $\mathcal{M}_0$ solves the entire SMDP $\mathcal{M}$. The task hierarchy is represented as a directed acyclic graph known as the *task graph* that represents the subtask relationships.

We will explain the HRL concepts in the context of the RTS domain shown in Fig. 3. In the case of multiple peasants, only one peasant can occupy a cell. Although there are multiple peasants in the world, there is only a single *learning* peasant; all other peasants execute fixed policies. In what follows, "the peasant" refers to this learning peasant.

The task hierarchy for this domain is shown in Fig. 6. The leaf nodes correspond to the primitive actions. Formally, each composite subtask $\mathcal{M}_i$ is defined by the tuple $(\mathrm{B}_i, \mathcal{A}_i, \mathrm{G}_i)$:

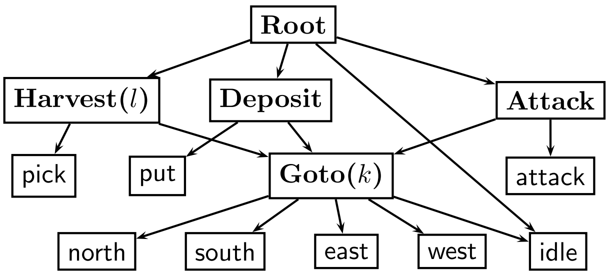

Fig. 6 The task hierarchy for the RTS domain

- *State abstraction B_i*: A projection function that maps a world state to an abstract state defined by a subset of the state variables. If the abstraction function is safe, it will only coalesce the world states that have the same value function and optimal policy into an abstracted state. For example, Goto(l)'s abstracted state space ignores every state variable but the location of the peasant.
- *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 parent task. The probability of the eventual termination of a subtask (other than Root) is 1. For example, Deposit is terminated when the peasant is not carrying anything.

The mechanism of state abstraction is critical to HRL. The individual tasks only see abstracted states (states comprised of a subset of the world state variables), and this helps compact the value functions significantly. Since a task only performs value function updates in states for which its child subtasks terminate, this *funneling* by the child tasks leads to a much smaller number of values being stored in the parent task. Since the Root task solves the entire SMDP, it sees the entire world state. However, it also benefits most from funneling. The Harvest tasks see an abstract state space based on the location of the peasant, what it is carrying, and the resource present at the resource locations. The Deposit task only sees the location of the peasant, and what it is carrying. The Attack task keeps track of the location of the peasant, and the status of the enemy. The Goto tasks only see the location of the peasant. The primitive movement actions, pick, and idle only store one value each. put needs to keep track of the location and the resource being toted. Finally, attack looks at the location of the peasant.

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-SMDP $\mathcal{M}_i$. A *hierarchically optimal policy* is a hierarchical policy that has the best possible gain, i.e., average reward per time step for the original SMDP. 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. For example, the best way to exit the building might depend on where one wants to go. To enable subtask optimization without regard to the supertasks, Dietterich introduced the notion of *recursive optimality* (Dietterich 2000).

In the context of HARL, recursive optimality is best defined as maximizing the average-adjusted total reward during each task assuming that all its subtasks are in turn recursively optimized with respect to the same global gain. Under this definition, recursive optimality coincides with hierarchical optimality when the task hierarchy satisfies a condition called *result distribution invariance* (RDI) i.e. the terminal state distribution of a task does not depend on the policy employed to accomplish it (Seri and Tadepalli 2000). (A sufficient condition for RDI is when every subtask has a unique terminal state.) Often, the tasks in the hierarchy can be designed in such a way that RDI is satisfied.[5] In the RTS domain, we have designed the hierarchy to include a parameterized Harvest subtask. Since the set of terminal states of this subtask depends solely on the binding value of the parameter, RDI holds, and this subtask can be called optimally based on the external context. On the other hand, the terminal states of an unparameterized Harvest subtask depends on the local policy, and RDI no longer holds.

3.1 Average-reward HRL

In this work, we adapt the model-based HARL algorithm as follows. At each non-root task node i, the algorithm maintains two functions. The first is a value function $V_i(s)$ that represents the total expected reward during task i starting from state s. The second function $T_i(s)$ represents the expected duration of task i starting from s. Given these two functions and an estimate of the global gain ρ, we can easily compute the bias $h_i(s)$ for task i and state s as follows:

$$h_i(s) = V_i(s) - \rho T_i(s). \tag{7}$$

The recursively optimal value function V_i for task i satisfies the following Bellman equations:[6]

$$\begin{aligned} V_i(s) &= R(s,i) \quad \text{if } i \text{ is a primitive subtask} && (8)\\ &= 0 \quad \text{if } s \text{ is a terminal/goal state for } i && (9)\\ &= V_j(s) + \sum_{s'\in\mathcal{S}} \Pr(s'|s,j)\, V_i(s') \quad \text{otherwise,} && (10)\end{aligned}$$

$$\text{where } j = \text{argmax}_a\left(h_a(s) + \sum_{s'\in S} P(s'|s,a)h_i(s')\right). \tag{11}$$

The value functions at the primitive subtasks just keep track of the reward received (see (8)). For the composite tasks, the value of every terminal state is 0 (see (9)). The value of a non-terminal state in a composite task is the sum of the total reward achieved by the child task from that state followed by the total reward till the completion of the parent task. In our version of the HARL algorithm, $V_i(s)$ is updated by the right hand side of (10) after selecting a subtask. Unless the action is exploratory, the child task is chosen to maximize the average-adjusted total reward during the task, where the adjustment is with respect to the current global gain ρ (see (11)) (Seri and Tadepalli 2000).

[5] We are currently exploring the issue of whether RDI can always be enforced through appropriate hierarchy design.

[6] The value functions and models are in fact based on the abstracted states, i.e., to $B_i(s)$ rather than s. We ignore this extra notation to improve readability.

In order to compute the bias values, $h_a(s)$ and $h_i(s)$, we need the durations $T_a(s)$ and $T_i(s)$. These are computed by updates based on the following Bellman equations, which are similar to (8)–(10)

$$T_i(s) = T(s,i) \quad \text{if } i \text{ is a primitive subtask} \tag{12}$$

$$= 0 \quad \text{if } s \text{ is a terminal/goal state for } i \tag{13}$$

$$= T_j(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,j) T_i(s') \quad \text{otherwise,} \tag{14}$$

where j is chosen according to (11). Since the root node represents a never-ending task, it does not have a finite V_i or T_i. Hence we store its bias value h_{root} directly, which is calculated using the following Bellman equation:

$$h_{\text{root}}(s) = \max_j \left(h_j(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,j) h_{\text{root}}(s') \right). \tag{15}$$

When the root task selects its child a greedily, the average reward ρ_r^π and average time ρ_t^π are updated (analogous to (2) and (3)) as follows:

$$\rho_r^\pi \leftarrow (1-\alpha)\rho_r^\pi + \alpha(V_a(s) + h_{\text{root}}(s') - h_{\text{root}}(s)), \tag{16}$$

$$\rho_t^\pi \leftarrow (1-\alpha)\rho_t^\pi + \alpha T_a(s). \tag{17}$$

The ratio of ρ_r^π to ρ_t^π gives the gain ρ, which is used in (7).

4 Variable-reward hierarchical reinforcement learning

VRRL exploits the decomposition of reward into reward components, whereas HRL is based on the idea of decomposing the tasks into subtasks. In this section, we explicate the variable-reward hierarchical reinforcement learning (VRHRL) algorithm that synergistically combines the two ideas. We seek to incorporate the variable-reward transfer mechanism into a hierarchical framework to benefit from value-function decomposition.

In non-hierarchical learning, the optimal policy is represented by a monolithic value function. This means that changes in the reward function will often result in non-local changes to this value function. Storing a new value function for every new weight would consequently result in a larger policy cache. Moreover, besides taking a longer time to converge, a monolithic value function is also more prone to lead to negative transfer especially when only a small number of policies are stored. Instead, a hierarchical value function is less prone to negative transfer especially when the hierarchical decomposition is closely aligned with the reward decomposition. Every subtask has a local value function, and local changes in rewards can be better managed. For instance, if none of the reward variations affect navigation, the Goto subtask only needs to learn its local value function once; perfect transfer is achievable for this subtask across the family of MDPs (and consequently every task below it in the task hierarchy). In the case of the RTS domain with multiple colliding peasants, the Goto subtask is only affected by the collision component of the reward, and transfers over perfectly to all MDPs for which the corresponding reward weight is identical. Thus, negative transfer can be significantly reduced with good hierarchy design.

4.1 Bellman equations for VRHRL

Variable-Reward HRL (VRHRL) extends our hierarchical RL approach by turning the value-functions over states into vectors of the same dimension as the reward components. Every subtask i (but the root) stores the total expected reward vector $\mathbf{V}_i$ during that subtask, and the expected duration T_i of the subtask for every state.

More formally, the value function vector $\mathbf{V}_i(s)$ for a non-root subtask i represents the vector of total expected reward components during task i starting from state s following a recursively optimal policy π. Hence, the value function decomposition for a non-root subtask satisfies the following equations, analogous to the scalar equations (8)–(11)

$$\mathbf{V}_i(s) = \mathbf{f}(s, i) \quad \text{if } i \text{ is a primitive subtask} \tag{18}$$

$$= 0 \quad \text{if } s \text{ is a terminal/goal state for } i \tag{19}$$

$$= \mathbf{V}_j(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s, j) \cdot \mathbf{V}_i(s') \quad \text{otherwise,} \tag{20}$$

where

$$j = \operatorname*{argmax}_a \left(\mathbf{w} \cdot \left(\mathbf{h}_a(s) + \sum_{s' \in S} \Pr(s'|s, a) \mathbf{h}_i(s') \right) \right). \tag{21}$$

Storing the bias vector indirectly in a form that is independent of the gain gives a limited form of reusability in that the value function for a subtree of the task hierarchy may be transferred across MDPs with different global gain vectors as long as the optimal policy for the subtree remains the same. 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.

In addition to the vectorization, one important difference from the previous set of equations is that the recursively optimal child task selection maximizes the weighted bias (see (21)). This is because the objective of action selection is to maximize the weighted gain (the dot product of the weight vector with the gain vector). In analogy to the scalar case, we compute the bias vector as

$$\mathbf{h}_i(s) = \mathbf{V}_i(s) - \boldsymbol{\rho} T_i(s).$$

The Bellman equations for the task durations T_i remain the same as before (see (12)–(14)). As in the scalar case, we keep track of the bias values of the root task directly since the root represents a recurrent task.

$$\mathbf{h}_{\text{root}}(s) = \max_j \left(\mathbf{h}_j(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s, j) \cdot \mathbf{h}_{\text{root}}(s') \right)$$

4.2 Transfer mechanism in VRHRL

The VRHRL agent has three components: the task hierarchy with the current subtask value functions and the associated 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 $\mathbf{w}_1$. When the agent receives a new weight $\mathbf{w}_2$, it first caches π_1 (the subtask functions and the global gain achieved for $\mathbf{w}_1$) in the policy cache $\mathcal{C}$. Next, it determines $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w}_2 \cdot \boldsymbol{\rho}^\pi)$, which in this case is π_1, and initializes its subtask 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, the algorithm either chooses an exploratory subtask based on ϵ-greedy exploration, or a greedy one according to (21). It learns the transition model $\Pr(s'|s,a)$ for each subtask by counting the number of resulting states for each state-subtask pair to estimate the transition probabilities. In doing so, the states are abstracted using the abstraction function defined at that subtask so that the transition probabilities can be represented compactly. Every time a subtask terminates in state s', (18)–(20) are used to update the total expected reward vector $\mathbf{V}_i(s)$ of task i, and (12)–(14) are used to update the scalar duration function $T_i(s)$.

For the global gain, the vector equivalent of (16) is

$$\boldsymbol{\rho}_r^\pi \leftarrow (1-\alpha)\boldsymbol{\rho}_r^\pi + \alpha(\mathbf{V}_a(s) + \mathbf{h}_{\text{root}}(s') - \mathbf{h}_{\text{root}}(s)) \tag{22}$$

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; (17) can be used unchanged. The updates in (22) and (17) are performed only when a is selected greedily. Finally, $\boldsymbol{\rho}^\pi$ is set to $\boldsymbol{\rho}_r^\pi / \rho_t^\pi$.

On sensing a new weight $\mathbf{w}_3$, the agent only caches the learned hierarchical policy π_2 for $\mathbf{w}_2$ if $\mathbf{w}_2 \cdot (\boldsymbol{\rho}^{\pi_2} - \boldsymbol{\rho}^{\pi_{\text{init}}}) > \gamma$. If this condition is not satisfied, then the newly learned policy is not sufficiently better than the cached version.

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 are 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 a similarity constant σ.

Once caching is complete, the policy that maximizes the weighted gain w.r.t. $\mathbf{w}_3$ 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 value function initialization for the agent.

Just as in the VRRL framework, VRHRL is also applicable when using value-function approximation. In the experiments described in the next section, we employ a linear value function over a set of predefined features for every task in the hierarchy. Instead of updating the value function for each possible value of the abstracted feature vector, the weights of the linear value function are updated in proportion to the gradient of the temporal difference error.

4.3 Experimental results

The performance results for the hierarchical agent are based on a 25×25 RTS map (discussed in Sect. 2.4) with 5 peasants, 5 fixed resource sites (5 resource types), a home base, and an enemy base. Since this is a huge state space, in addition to the hierarchical decomposition, we also employ linear value-function approximation. For a vectored value function,

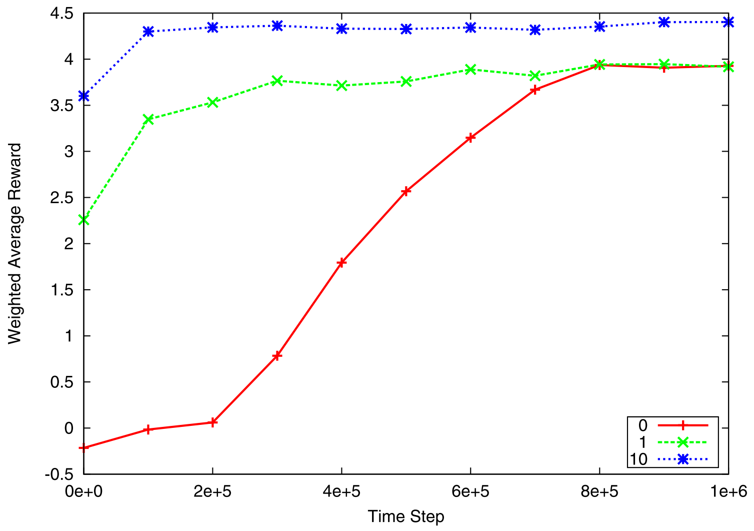

Fig. 7 Performance of VRHRL on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials)

this entails maintaining a set of parameters for each vector component; for a hierarchical value function, we maintain such a set of parameters for each subtask. The state features used are distances to sites, indicators for what the peasant is carrying, indicators for what resource is present at the sites, and an indicator for the appearance of the enemy base, etc. The learning rate for updating the weights of the linear function approximation is 10^{-4}. The rate of ϵ-greedy exploration is set to 0.1. The reward feature vector has 7 components that are associated with 5 resources, peasant collisions, and enemy elimination.

The experiments are designed exactly as in Sect. 2.4. Two important parameters that govern the learning behavior of the VRHRL algorithm are the satisfaction constant $\gamma = 0.01$, and the similarity constant $\sigma = 0.01$; these parameters are fixed for all experiments. Just as the learning rate and the rate of exploration are key parameters in regular RL, the satisfaction and similarity constants trade-off speedup in learning against the size of the policy cache. For instance, the smaller the satisfaction, the more the number of policies stored in the cache. When a new weight is detected, the algorithm is more likely to find a policy that is very close to optimal from this heavily-filled cache. The smaller the similarity, the more new subtask data is stored by the algorithm instead of maintaining references to previously stored subtask information.

Figure 7 shows the learning curves for VRHRL. The learning curve for the test weight given no prior training weights (i.e., an empty policy cache) is the slowest to converge. However, the jump-start and speed of convergence for the test weight improves as more training weight vectors are experienced.

We have previously noted that the flat learner does not need to relearn the transition model from scratch when the reward weights change. In the hierarchical learner, the transition models at the composite tasks (and the models at the primitive subtasks) do not need to be relearned when RDI holds because then the state transitions at every subtask are invariant to the changes in the policies of the child tasks. Figure 8 shows the results of repeating the VRHRL experiment without the policy-caching mechanism. Here, reusing the learned models is only slightly beneficial to the hierarchical learner, leading to a small initial speed-up in

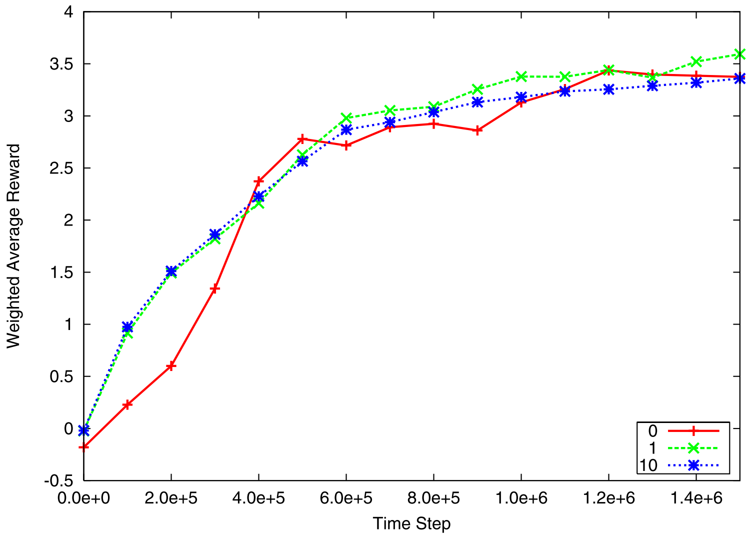

Fig. 8 Performance of the hierarchical learner without the VRHRL policy-caching mechanism (averaged across 10 trials)

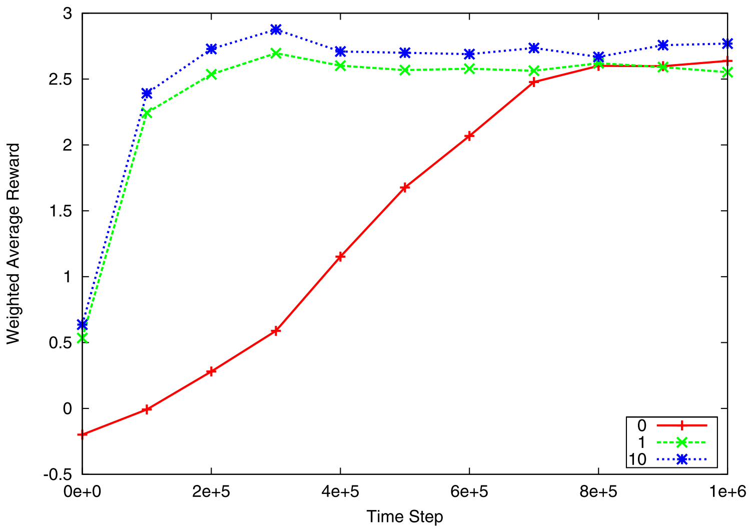

Fig. 9 Performance of the hierarchical learner when caching only Goto tasks (averaged across 10 trials)

learning but without any jump-start or faster convergence. A complete trial of the VRHRL experiment (10 training weights, 1 test weight) takes 7266.1 secs on average, while that of the experiment without caching takes 7204.3 secs on average—the caching mechanism adds an overhead of only 0.85% to the overall running time of the experiment.

We have mentioned that VRHRL benefits from having certain subtasks like Goto transfer over significantly because they are affected by only a small number of reward components.

Figure 9 demonstrates that with caching only the Goto tasks (and reinitializing all other tasks to zero), the learner sees a jump-start that is smaller than that for complete policy caching, but the performance is still better than when not caching at all.

We have also conducted experiments in which the weight vector **w** is induced from the scalar reward signal R using perceptron-style incremental learning:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R - \mathbf{w} \cdot \mathbf{f})\mathbf{f}$$

where **f** is the reward feature vector. The agent is allowed to take random actions for a certain number of steps in order to sense the scalar reward and the corresponding reward features. The number of steps can be tailored to allow the agent to get a good estimate of the true weight vector—the less likely it is to see a certain reward component, the more time the agent needs to estimate the weight for that component. In our experiments, we allow the agent to learn an extremely good estimate of the weight vector, and the resulting learning curves look exactly like in the plots shown for VRRL and VRHRL except that they are shifted to the right by the number of sensing steps.

5 Related work

Taylor et al. (2005) propose a value-function mapping approach for transfer between source and target reinforcement learning tasks. In their approach, a hand-coded similarity function is used for transferring the Q-value functions. In particular, hand-coded functions are used to specify similar state spaces, action spaces, and representation mappings between the source and target tasks. This work has been extended to automate the determination of the similarity mappings (Liu and Stone 2006). More specifically, qualitative dynamic Bayesian nets (QDBNs) are used to represent the structural information in the source and target tasks; these QDBNs are assumed to be designed by the domain expert. An algorithm based on structure mapping is designed to discover the similarities between the QDBNs of the source and target tasks to facilitate the transfer of value functions. Since our work assumes that the dynamics are exactly the same in the source and target tasks, it would be interesting to combine the two methods to handle both varying dynamics and reward functions.

Torrey et al. (2007) use Inductive Logic Programming (ILP) techniques to learn macros that are used for transferring Q-functions across different problems. Their method, applied to the problem of m-on-n breakaway in Robo-cup soccer, proceeds as follows: the learner collects the training examples while playing a few source games. ILP is then used to learn relational macros that describe a successful strategy (policy) in the source task. The macros are then used as default policies in the target task for a fixed number of iterations. Q-learning is then used to obtain a better policy. In effect, the relational macros are used as an exploration method in the target domain. While their approach generalizes over several source games to learn the ILP rules, we consider each source task for caching incrementally. This greedy approach could result in a larger policy cache, and increase negative transfer in the pathological case where successive weights are radically different from one another.

Guestrin et al. (2003) use linear programming techniques to solve relational MDPs to compute a set of value functions from one MDP that can be directly used in another. An alternative approach of converting the relational MDPs into several propositional MDPs and solving the propositional MDPs has also been studied (Mausam 2003). Here, the value functions are represented using first-order regression trees and are then used to determine a policy for a new MDP. In both these methods, the relational structure is exploited for learning the value functions for similar classes of MDPs. Researchers have also looked at the problem of transferring knowledge from one learner to another as imitation (Price and Boutilier 2003). The observer extracts information about its capabilities by observing the state transitions of the agent. Our goal is chiefly to facilitate transfer in the presence of *time-varying* rewards.

Our approach to variable-reward transfer learning borrows ideas from multi-criteria reinforcement learning (Gabor et al. 1998). These ideas are related to earlier work on solving multi-criteria MDPs where weights are used to indicate the importance of different reward components. For example, in the work by White (1982), vector-based generalizations of successive approximation techniques are used to solved the MDP. Feinberg and Schwartz (1995) formulate the problem as optimizing a weighted sum of the total discounted rewards for the different components of the reward function. Russell and Zimdars (2003) consider the additive decomposition of rewards to solve the MDP. Guestrin et al. (2001) used reward decomposition to make multi-agent coordination tractable. The work by Parr (1998) decomposes the problem of solving a big MDP into one of solving smaller weakly-coupled sub-MDPs. The fact that the value function of a fixed policy over the sub-MDP is linear in the values of its exit states is effectively exploited to speed up the solution of the overall MDP. In contrast to all of the work mentioned in this paragraph, our approach to reward decomposition is motivated by transfer across MDPs that share the same dynamics with different reward structure.

6 Conclusions and future work

In this paper, we showed that vector-based value function learning and caching of policies can lead to effective variable-reward transfer learning. We also showed that hierarchical reinforcement learning can accelerate transfer across variable-reward MDPs more so than in the non-hierarchical setting. Our results are in the model-based setting and have the added advantage that the models need not be relearned from scratch when the rewards change. We have also shown that it is easy to learn them from non-decomposed scalar rewards since the scalar reward is linear in the reward weights.

Possible future directions include extensions to shared subtasks in the multi-agent setting (Mehta and Tadepalli 2005), and to MDP families that share only part of the dynamics. In RTS games, we could consider MDPs that contain different objects such as peasants, footmen, and archers in different proportions and locations. Although the dynamics of local interaction for each peasant may be the same, the changes in the peasants' locations and their numbers mean that, technically, the different MDPs have different dynamics. Nevertheless, people seem to be able to effectively transfer their strategies from one such RTS game to another. Duplicating this ability in machines would be a big advance for the field of machine learning.

Acknowledgements We thank the reviewers for their detailed comments and helpful suggestions. 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

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

Andre, D., & Russell, S. (2002). State abstraction for programmable reinforcement learning agents. In *Eighteenth national conference on artificial intelligence* (pp. 119–125).

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *9*, 227–303.

Feinberg, E., & Schwartz, A. (1995). Constrained Markov decision models with weighted discounted rewards. *Mathematics of Operations Research*, *20*(2), 302–320.

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

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

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. In *International joint conference on artificial intelligence*.

Kaelbling, L., Littman, M., & Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *AI Journal*.

Liu, Y., & Stone, P. (2006). Value-function-based transfer for reinforcement learning using structure mapping. In *Proceedings of the twenty-first national conference on artificial intelligence*.

Mausam, D. (2003). Solving relational MDPs with first-order machine learning. In *Proceedings of the ICAPS workshop on planning under uncertainty and incomplete information*.

Mehta, N., & Tadepalli, P. (2005). Multi-agent shared hierarchy reinforcement learning. In *ICML workshop on rich representations in reinforcement learning*.

Natarajan, S., & Tadepalli, P. (2005). Dynamic preferences in multi-criteria reinforcement learning. In *Proceedings of the ICML*.

Parr, R. (1998). Flexible decomposition algorithms for weakly coupled Markov decision problems. In *UAI*.

Price, B., & Boutilier, C. (2003). Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, 569–629.

Puterman, M. L. (1994). *Markov decision processes*. New York: Wiley.

Russell, S., & Zimdars, A. (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 the 10th international conference on machine learning*. San Mateo: Morgan Kaufmann.

Seri, S., & Tadepalli, P. (2002). Model-based hierarchical average reward reinforcement learning. In *Proceedings of the ICML* (pp. 562–569).

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1–2), 181–211.

Tadepalli, P., & Ok, D. (1998). Model-based average reward reinforcement learning. *Artificial Intelligence*, *100*, 177–224.

Taylor, M., Stone, P., & Liu, Y. (2005). Value functions for RL-based behavior transfer: a comparative study. In *Proceedings of the twentieth national conference on artificial intelligence*.

Torrey, L., Shavlik, J., Walker, T., & Maclin, R. (2007). Relational macros for transfer in reinforcement learning. In *Proceedings of the 17th conference on inductive logic programming*.

Weeks, J. (1985). *The shape of space: how to visualize surfaces and three-dimensional manifolds*.

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