Automatic Shaping and Decomposition of Reward Functions

Bhaskara Marthi

Abstract

This paper investigates the problem of automatically learning how to restructure the reward function of a Markov decision process so as to speed up reinforcement learning. We begin by describing a method that learns a shaped reward function given a set of state and temporal abstractions. Next, we consider decomposition of the per-timestep reward in multieffector problems, in which the overall agent can be decomposed into multiple units that are concurrently carrying out various tasks. We show by example that to find a good reward decomposition, it is often necessary to first shape the rewards appropriately. We then give a function approximation algorithm for solving both problems together. Standard reinforcement learning algorithms can be augmented with our methods, and we show experimentally that in each case, significantly faster learning results. This is an extended version of the paper that appeared at ICML 2007.

1 Introduction

Reinforcement learning is a popular approach to creating autonomous agents. In the RL framework, rather than being explicitly programmed, the agent is allowed to act in the environment, and receives numerical rewards at each step. RL algorithms attempt to learn a policy that maximizes the total expected reward (or some related criterion). Thus, the reward function implicitly describes optimal behaviour. Conversely, given any definition of optimality (or more precisely, a separable utility function on state–action trajectories), there are infinitely many reward functions that are consistent with it. As practitioners have long recognized, the choice of reward function can have a strong effect on how long it takes to learn an optimal policy (Mataric, 1994; Alstrom, 1998). Intuitively, a good reward function is one that gives the agent useful feedback about an action soon after it is performed. In many goal-based problems, however, the most obvious reward function is the one that gives a reward upon reaching the goal state, and either discounts the future or charges a cost for each nongoal state. Such a reward function gives very delayed feedback, leading to slow learning. This realization led to the idea of a *shaping reward* added on to the original one, which rewards intermediate progress towards the goal.

(Ng et al., 1999) proved the basic theoretical result that a shaping reward preserves the optimal policy if and only if it can be written as a difference of some potential function evaluated at the source and destination states.

Unfortunately, all the above approaches require as input either a shaping function or a potential function. These quantities, which will usually depend on the numerical magnitude of total expected rewards, are not always easy for the system designer to estimate. Furthermore, consider the transfer learning setting, where an agent makes use of experience in one problem to learn faster on a second. One might imagine using shaping rewards as a mechanism for transfer. But, even in MDPs that are "structurally" similar, the ideal potential function, which equals the true value function, might be quite different. In Section 3, we show a way around these problems. Our algorithm takes as input a *state abstraction* function. This could, for example, be represented by a list of the state variables that are considered most relevant to the task. It also accepts a set of *temporally abstract actions* (although this is optional). It then solves for the potential function of the resulting *abstract MDP*, and uses it to construct a shaping reward.

Next, we consider the class of *multieffector environments*, in which the overall agent is decomposed into units and an action is a vector containing a command to each unit. Recent theoretical and practical work (Bagnell & Ng, 2006; Russell & Zimdars, 2003; Schneider et al., 1999) suggests that learning in such domains can be sped up given an additive *reward decomposition* across units; intuitively, such a decomposition lets each unit know the portion of the observed rewards for which it was responsible.

Additive reward decompositions are often easy to specify if the activities of the different units are fairly independent. If, on the other hand, the units are working towards some joint goal, there may be no appropriate decomposition. In Section 4, we first argue that even if the original reward function cannot be usefully decomposed, a shaped version of it often can. Such decompositions are rarely exact, so we describe an algorithm that finds the best approximate decomposition in a linear family.

2 Markov Decision Processes

We use the standard Markov decision process (MDP) formalism for representing fully observable environments (Bertsekas & Tsitsiklis, 1996). Here are our notation and assumptions. We define an MDP to be a 5-tuple $\mathcal{M} = (S, A, P, R, d)$ where S is a set of states, A is a set of actions, $P(\cdot|s, a)$ is the transition probability distribution upon doing action a in state s, $R(s, a, s')$ is the resulting reward, and $d(\cdot)$ is a probability distribution over the initial state. A (stationary) *policy* is a function on the state space such that each $\pi(s)$ is an action, or more generally, a probability distribution over actions. We work with undiscounted value and Q-functions. [1] To ensure well-definedness, we make the usual assumption that the MDP has at least one policy that is proper, i.e., eventually reaches a

[1] The results can be extended to the discounted case.

terminal state with probability 1, and every nonproper policy has value $-\infty$ at some state.

3 Learning shaping functions

3.1 Background

Reward shaping refers to the practice of replacing the original reward function of an MDP by a *shaping reward* $\tilde{R}(s, a, s')$ in the hope that this will make the problem easier to solve. Let $\tilde{\mathcal{M}}$ be the modified MDP, and $\tilde{V}$ refer to the value function in $\tilde{\mathcal{M}}$. One would like to know what types of shaping reward functions preserve the optimal policy of the original problem. (Ng et al., 1999) answered this question by showing that if there exists a *potential function* $\Phi(s)$ such that $\tilde{R}(s, a, s') = R(s, a, s') + \Phi(s') - \Phi(s)$, then, for any policy π, $\tilde{V}^{\pi}(s) = \pi(s) - \Phi(s)$. In particular, (near) optimal policies in $\tilde{\mathcal{M}}$ correspond to (near) optimal policies in $\mathcal{M}$. The condition is also necessary: given any shaped reward that does *not* correspond to a potential, there is some set of transition probabilities for which optimal policies in the shaped MDP are suboptimal in the original problem.

Having characterized when shaping is correct, the next question is when and how it speeds up learning. (Ng et al., 1999) argue that the ideal shaping function is based on the potential $\Phi = V$. In this case, the value function in the shaped MDP is identically 0. Shaping also reduces the amount of exploration required. (Laud & Dejong, 2003) analyse the benefits of shaping in terms of the *horizon*, which is a bound on how far one has to look ahead to act near-optimally. They provide an algorithm that finds an optimal policy while essentially only exploring the portion of the state space that is visible within the horizon of states occurring on an optimal trajectory. A shaping function is therefore capable of speeding up learning if the shaped MDP has a short horizon. Using the value function as a potential leads to a shaped reward satisfying $E[\tilde{R}(s, a, s')|s, a] = Q(s, a) - V(s)$. Thus, optimal actions can be taken by maximizing the one-step reward, and so the horizon equals 1 in this case.

3.2 Difference functions

The optimal shaping function above can be described in a slightly different way. First, define the state transition graph $\mathcal{G}(\mathcal{M})$ of an MDP $\mathcal{M}$. The nodes of this graphs are the states of $\mathcal{M}$ and there is an edge between s and s' whenever there exists an action a for which $P(s'|s, a) > 0$. Now define the *difference function* D of $\mathcal{M}$: For each directed edge (s, s') of $\mathcal{G}(\mathcal{M})$, $D(s, s') = V(s') - V(s)$. The optimal shaping function above can then be rewritten as $\tilde{R}(s, a, s') = R(s, a, s') + D(s, s')$. We will make use of this alternate representation in our algorithms.

3.3 Our approach

Existing approaches to shaping require the shaping function or potential function to be provided as input. These quantities are based on the numerical magnitude of total rewards, and may be difficult to estimate. The difference function defined above suffers from this problem as well. We would therefore like an algorithm that takes input of a more qualitative nature.

Our approach is to find an approximation to the ideal shaping function by solving a simpler abstract problem. Given an MDP $\mathcal{M}$ with the usual notation, let z be a function that maps each state s to an *abstract state*. We will identify an abstract state z with the set of states that map to it. Also, let O be a set of temporally abstract *options* (Sutton et al., 1999) where, for our purposes, an option o consists of a policy π_o and a termination set $G_o \subset S$. Given an option o, define the transition probability $P(s'|s,o)$ where s' is obtained by doing actions according to o starting at s until a termination state of o is reached. Similarly, define the reward $R(s,o)$ to be the expected total reward until s' is reached.

The set of options O thus defines a new MDP over the original state space, in which the action set is replaced by O. We would like to turn this into an MDP over the abstract state space. To do this requires finding a way of weighting the states that correspond to a given abstract state. Consider a policy π that always chooses an option uniformly at random. π results in a distribution $\mathcal{P}(\omega)$ over state trajectories $\omega = (s_0, s_1, \ldots, s_T)$ corresponding to sampling from the initial state distribution d, then following π until termination. Note that the trajectories only include the terminal states of the options, and not the intermediate states. Let the random variable $C_x(\omega)$ denote the number of times some state or abstract state x occurs along ω. We can now define the *weight* w_s of s in z to be $\frac{E_{\mathcal{P}}[C_s]}{E_{\mathcal{P}}[C_z]}$. In words, the weight of a state is the proportional to its expected frequency of occurrence.

Definition 1. The *abstract MDP* corresponding to an MDP (S, A, P, R, d), state abstraction z, and option set O is defined as $\bar{M} = (\bar{S}, \bar{A}, \bar{P}, \bar{R}, \bar{d})$ where:

- $\bar{S} = z(S)$
- $\bar{A} = O$
- $\bar{P}(z, o, z') = \sum_{s \in z} w_s \sum_{s' \in z'} P(s'|s,o)$
- $\bar{R}(z,o) = \sum_{s \in z} w_s R(s,o)$
- $\bar{d}(z) = \sum_{s \in z} d(s)$

Algorithm 1 estimates the abstract MDP from samples, then solves for the difference function. Section 3.4 will describe how this is done. We will show experimentally in Section 3.5 that even when the original MDP is very large, the abstract MDP can be made small enough to solve feasibly. The algorithm satisfies the following consistency property:

Theorem 1. *In finite MDPs, as the number of samples T tends to ∞, the difference function found by Algorithm 1 when run in an MDP converges almost surely to the difference function of the corresponding abstract MDP.*

Algorithm 1 Shaping function learner. z is a state abstraction function, O is a set of options, and T is a nonnegative integer. The update procedure on line 9 maintains a simple running average, and assumes that unseen state–action pairs lead to a dummy terminal state with a very negative reward.

1: **function** LEARN-SHAPING-FUNCTION(z, O, T)
2: Initialize transition, reward estimates $\hat{P}, \hat{R}$
3: **repeat**
4: $s \leftarrow$ current environment state
5: Sample o randomly from O
6: Follow option o until it terminates
7: $s' \leftarrow$ current environment state
8: $r \leftarrow$ be the total reward received while doing o
9: Update $\hat{P}, \hat{R}$ using sample $(z(s), o, r, z(s'))$
10: **until** T actions have been taken in the environment
11: $\hat{\mathcal{M}} \leftarrow (z(S), O, \hat{P}, \hat{R})$
12: Compute difference function D of $\hat{\mathcal{M}}$
13: **return** function $\tilde{R}(s, a, s') = R(s, a, s') + D(s, s')$
14: **end function**

One can also show that if the option set allows near-optimal behaviour, for example, if it includes the primitive actions, and if the state abstraction approximately respects the transition and reward functions of the original MDP, i.e., it is an approximate model irrelevance abstraction (Li et al., 2006), then the returned shaped reward function will be close to the optimal one. Regardless of the accuracy, though, the shaping rewards will preserve optimality because they are based on a potential function. A poor set of abstractions will only affect the sample complexity.

Several extensions to the basic algorithm are possible:

- The procedure could be run in parallel with a control learning algorithm, and the learnt optimal policy could be used as one of the options, so that the shaped rewards will approach optimality.
- If no options are available, it is always possible to use a trivial option that randomly chooses an action and terminates in one step.
- The method can be applied to *partially observable* MDPs, so long as the abstraction is a function only of observable quantities. In particular, given a filtering algorithm, any function of the belief state estimate, such as the most likely physical state, can be used.

3.4 Computing the Difference Function

There are several ways to compute the difference function. First, one can use any dynamic programming algorithm to compute V and then use the fact that $D(s, s') = V(s') - V(s)$. This can be done assuming the abstract MDP is small enough to solve exactly. For example, in the Othello problem used in Section 3.5,

the abstract MDP has 135 states and 2 actions, and so, e.g., modified policy iteration can be used.

When the abstract MDP itself is large, we may use, e.g., approximate value iteration using function approximation for the value function. A potential problem is that we are ultimately interested in D, not V. Consider, for example, an MDP where we are trying to navigate to a goal, and pay a cost of 1 per time step. The value function of a state depends on the total distance to the goal, whereas the difference between two adjacent states only depends on which one is closer. If the state space is large, the magnitude of the values would be much larger than differences between adjacent states. Thus, a function approximator such as linear regression with an L^2 penalty might do a poor job of fitting those aspects of V that are actually relevant for predicting D. It may therefore be preferable to use an algorithm that tries to approximate D directly.

The difference function satisfies two sets of equations. First, for any edge (s, s') in the transition graph,

$$\begin{aligned} D(s,s') &= V(s') - V(s) \\ &= V(s') - \max_a \left(E_{P(s''|s,a)}(R(s,a,s'') + V(s''))\right) \\ &= -\max_a \left(E_{P(s''|s,a)}(R(s,a,s'') + V(s'') - V(s'))\right) \\ &= -\max_a \left(E_{P(s''|s,a)}(R(s,a,s'') + V(s'') - V(s) - (V(s') - V(s)))\right) \\ &= D(s,s') - \max_a \left(E_{P(s''|s,a)}(R(s,a,s'') + D(s,s''))\right) \end{aligned}$$

Second, for any cycle $s_0, s_1, \ldots, s_n = s_0$ in the transition graph,

$$D(s_0, s_1) + \ldots + D(s_{n-1}, s_n) = 0$$

These equations suggest an approximate dynamic programming procedure for computing D, shown in Algorithm 2. The algorithm alternates between "Bellman" type updates of the difference function, and normalizations to ensure that the difference along any cycle sums to 0. The convergence properties of this algorithm are still open. It has been experimentally verified to converge to the true difference function in small examples, for a variety of settings of p. [2]

3.5 Experiments

Our goal in the experiments is to determine whether the samples spent on learning a shaping reward could have been more usefully spent on standard reinforcement learning. As a baseline, we use Q-learning with function approximation, which may seem superficially similar to Algorithm 1 since both are learning an abstracted value function. The test domain is the game of Othello (VanEck & VanWezel, 2005). To turn Othello into a Markovian environment, we assume a fixed, materialistic opponent who always makes the move that captures the

[2] See paper website at `http://people.csail.mit.edu/bhaskara/autoshape` for code and experiments

Algorithm 2 Difference updating algorithm. The notation $x \leftarrow_\eta y$ means that the parameters of x are adjusted, by an amount proportional to η, to make x closer to y.

function DIFFERENCE-UPDATING($\mathcal{M}$, p, η)
  $\forall s, s'\ D(s, s') \leftarrow 0$
  for t from 0 to T **do**
    MoveType $\leftarrow$ SAMPLEBERNOULLI(p)
    if MoveType = 0 **then**
      $(s, s') \leftarrow$ SAMPLEUNIFORMLY(EDGES($\mathcal{G}(\mathcal{M})$))
      $D(s, s') \leftarrow_\eta D(s, s') - \max_a \left(E_{P(s''|s,a)}(R(s, a, s'') + D(s, s''))\right)$
    else
      $(s_0, \ldots, s_n) \leftarrow$ SAMPLEUNIFORMLY(CYCLES($\mathcal{G}(\mathcal{M})$))
      $\sum_{i=1}^{n} D(s_{i-1}, s_i) \leftarrow_\eta 0$
    end if
  end for
  return D
end function

largest number of pieces. Reward is only received when the game ends: 1 for winning, 0 for tying, and -1 for losing.

A qualitative piece of prior knowledge about this game is that the squares on the edges of the board are most valuable because they are difficult to capture; in particular, the corner squares can never be recaptured once a player has occupied them. Define the advantage of a player on a set of squares to be the number of pieces of that player on that set minus the number of opponent pieces on that set. We divide the game into three equal-length phases, and the squares on the board into four sets: corner squares, edge squares, "precorner squares" (diagonally adjacent to the corner), and internal squares. For each phase, we have a feature that equals the advantage on each of the four sets, as well as a constant feature. The features all depend on the board position immediately after the move being considered. Second, we tried augmenting this algorithm by first learning a shaping reward based on a state abstraction that grouped together states having like values of 1) the advantage on corner squares 2) the phase of the game, and 3) the advantage on non-corner squares (binned into five equal intervals). We used two options, both of which terminate after one step. The first picks a random move, while the second makes a greedy move; typically, neither of these will be optimal. Though the original game has about 10^{28} states, the abstract MDP has only 135 states, and so can be estimated and solved reasonably well after 10000 moves, or about 300 games. If a state is encountered during Q-learning that is not present in the abstract MDP, the shaping reward is just set to 0.

The results are shown in Figure 1. As soon as the shaping reward learning phase is complete, the shaped algorithm jumps ahead of the unshaped one. Furthermore, the dynamic range of the Q-values learnt by the unshaped algorithm is about two orders of magnitude lower than it should be, whereas the shaped

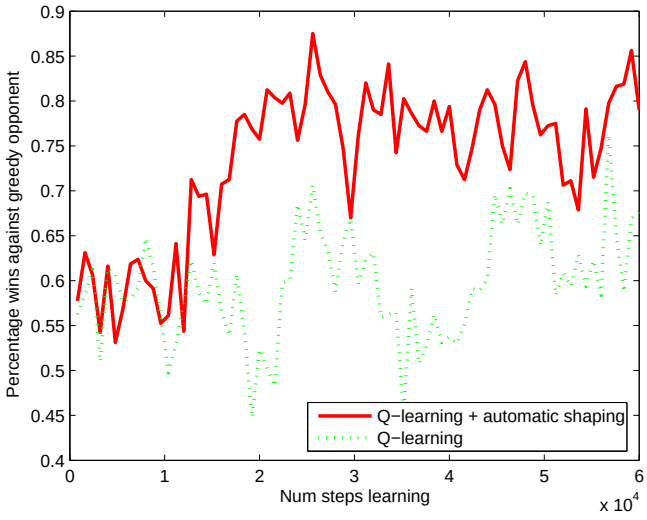

Figure 1: Learning curves for Othello, averaged over four runs. Each learnt policy was evaluated by playing 40 games against a greedy opponent.

version has learnt Q-values that appear to be plausible estimates of future reward (see website). In Othello, the corner squares often become useful several steps after they are first occupied. The shaped algorithm is able to realize the value of the corner squares quickly, as soon as it builds a reasonable model of the abstract MDP.

4 Learning reward decompositions

4.1 Background

Many real-world MDPs have what might be termed multieffector structure. Formally, a *multieffector MDP* (sometimes also known as a cooperative multiagent MDP) consists of:

- An MDP $\mathcal{M}$;
- A set $\mathcal{E}$ of *effectors*;
- A function c on $\mathcal{E}$, where $c(e)$ denotes the set of *commands* that may be sent to effector e;
- A function E from the state space of $\mathcal{M}$ to $2^{\mathcal{E}}$, where $E(s)$ denotes the set of effectors present in state s. We further require that the set of actions available at s equals $\prod_{e \in E(s)} c(e)$, i.e., an action at a state corresponds to giving a command to each unit present in that state.

The terminology of effectors is borrowed from robotics, but we use it more generally. For example, in a network routing problem (Littman & Boyan, 1993),

each node would be considered an effector. $E(s)$ would be the set of nodes active in state s, and the set of commands for a node would be the set of neighbours, so on each step, each active node is commanded to pass its current packet to one of its neighbours.

Several practical applications have used fully decentralized learning algorithms for the case when the reward function decomposes additively across effectors (Schneider et al., 1999; Littman & Boyan, 1993). (Russell & Zimdars, 2003) described a partially decentralized SARSA algorithm. Given a reward decomposition $R = \sum_e R_e$, the algorithm estimates Q-components $Q_e^\pi(s,a) = E[\sum_t r_{e,t}]$ where the expectation is over trajectories that begin by doing a in s then following π. The algorithm is shown empirically to work well when the individual Q-components can each be approximated in terms of a small number of state and action variables. (Bagnell & Ng, 2006) described a centralized model-based algorithm whose sample complexity is logarithmic in the number of effectors. The bound applies to settings where the states and actions decompose across effectors, each effector's reward depends only on its local state, and the local transition models are not too tightly coupled. Overall, there is plenty of evidence in the literature that reward decompositions are capable of improving sample complexity, but only when each reward component is local, in some sense, to a particular piece of the problem.

4.2 Our approach

We use a very simple navigation problem to build intuition. The problem involves N robots, each on a separate undirected graph. The MDP state factorizes as $s = (s_1, \ldots, s_n)$, where each s_e is a node of the corresponding robot's graph. Each graph has a terminal node σ_e, and the terminal state of the MDP is $(\sigma_1, \ldots, \sigma_n)$. Each robot is considered as an effector, and the available commands for an effector e in a state s are to move to any of the neighbours of s_e. A global cost of -1 is charged per step.

Consider the case where there are two robots, each robot graph is just the chain $0, 1, \ldots, 10$, 0 is the terminal node, and the actions at nonterminal nodes are L(eft) which moves towards 0, and R(ight) which moves away from it. First, suppose we just use Algorithm 1 where the state abstractor is the identity function. The learnt potential is then $\Phi(s) = -\max_e(s_i)$. Suppose action (R, L) is done in state $(5, 10)$. Since robot 2, which was further from the goal, moved in the right direction, the potential value increases by 1, and so a shaping reward of $1 - 1 = 0$ is given. But this ignores the fact that robot 1's piece of the action was suboptimal. Of course, in this particular state, it doesn't matter what robot 1 does, but the point is that in a related state, such as $(5, 2)$, robot 1's action does matter, and we have lost a chance to give useful feedback.

On the other hand, suppose we try to decompose the reward. An obvious choice of reward decomposition is to have each $R_e = -1/n$ until the terminal state is reached, but such a decomposition is unlikely to be useful. For example, if we are using the decomposed SARSA algorithm, each Q-component $Q_e(s,a)$ would equal the distance to the goal of the furthest robot, and so each component

would depend on all the state and action variables. We could instead only share the -1 reward among robots that haven't reached their terminal node yet, but each Q-component will still depend on the entire state. We could also just give each robot a constant negative reward till it reaches its goal. In this case, the Q-components will be local, but the reward structure of the problem has been changed significantly; for example, it will now be preferred to have two robots finish in 10 steps and the third in 200 steps, rather than having all of them finish in 100 steps.

The solution is to decompose the shaped reward function instead. The reward components have to add up to 0 if the joint action is optimal, which requires each effector action to be optimal, and -2 otherwise. Also, we would like to be able to write, for each effector e, $R_e(s, a, s') = R_e(s_e, a_e, s'_e)$. This leads to a linear system with more equations than variables, and so it can only be solved in a least squares sense. In the solution, R_e is about $-.1$ whenever effector e's part of the move is optimal, and -1.6 otherwise. Note that the reward decomposition is not exact.

Algorithm 3 Reward decomposition learner. z is a state abstraction function, O is a set of options, T, is a nonnegative integer, and each g_e is a function from triples (s, a, s') to a feature vector.

1: **function** LEARN-REWARD-DECOMPOSITION(z, O, T, $\{g_e\}$)
2: Learn a shaped reward function $\tilde{R}$ as in Algorithm 1 using z, O, T.
3: Use the samples from step 2 to get a least squares estimate $\tilde{R}(s, a, s') = \sum_e \beta_e \cdot g_e(s, a, s')$
4: **Return** weights β corresponding to reward components $R_e(s, a, s') = \beta_e \cdot g_e(s, a, s')$
5: **end function**

Algorithm 3 is based on the above idea. It applies to general multieffector MDPs—unlike in the example, the effectors need not be completely decoupled from each other. It requires a set of features to be provided for each effector, and uses standard linear regression to compute corresponding weights.

Algorithm 3 will, if implemented naively (as we did in our experiments), require space and time exponential in the number of effectors N. We can get around this by using the algorithm of (Guestrin et al., 2003), which takes in a DBN representation of an MDP, and finds a linear approximation to its value function in time polynomial in the DBN size, given bounds on the treewidth. The abstract MDP learner would have to be modified to take in the DBN structure and learn the parameters.

The results of (Bagnell & Ng, 2006) imply that the sample complexity of learning a good reward decomposition is at least linear in N in the worst case. In many problems of interest, such as search and rescue, real-time strategy games, and Robocup, there is a reasonable upper bound on N. In large MDPs of this sort, there have been practical demonstrations (Marthi et al., 2005) that state-of-the-art RL algorithms can perform adequately without a reward

decomposition in situations with on the order of a few dozen effectors; the bottleneck tends to be the length of the planning horizon. On the other hand, for MDPs where N is very large, such as sensor networks or control of traffic signals, it would not be practical to learn a reward decomposition using Algorithm 3. Further prior knowledge would be needed, e.g., information that allows inter-object generalization.

4.3 Experiments

For our experiments we used a navigation problem in which four robots are navigating to a goal in a two-dimensional grid. There is a constant cost of -1 per timestep. In addition, there is a collision cost whenever two robots are in the same location. We compared flat Q-learning, automatic shaping, and automatic decomposition. For automatic shaping, we used the state abstraction that mapped a state into the shortest path distance of each robot from its destination. For the automatic decomposition, each robot had indicator features for its distance to the goal and for collisions. Thus, at the abstract level, the problem is similar to the earlier example, but in the actual problem, there are interactions that must be taken into account. We used the decomposed SARSA algorithm (Russell & Zimdars, 2003) to learn a Q-component for each robot. Each robot's Q-component depended only on its position and action, and whether another robot was planning to move to the same square. Figure 2 shows the learning curves. Both automatic shaping and automatic decomposition eventually learn an optimal policy, while Q-learning never does (the asymptote for automatic decomposition is slightly lower because it failed to find an optimal policy on one of the 20 trials, and episodes were cut off after 60 steps). The decomposed method learns much faster, though—after the initial potential learning phase, its learning curve increases almost vertically.

5 Related work

Aside from the references in Section 3, there have been several recent papers on reward shaping. (Wiewiora, 2003) showed that in the case of tabular temporal-difference using an advantage-based exploration policy, shaping using potential function Φ is equivalent to initializing the Q-function as $Q(s, a) = \Phi(s)$. "Multigrid" DP algorithms use solutions to a coarse-grained approximation of the problem to initialize the more fine-grained one(Chow & Tsitsiklis, 1991). It is not yet known, however, to what extent the equivalence between shaping and initialization extends to function approximation, multieffector learning algorithms, and learning algorithms that don't use a value or Q-function.

(Konidaris & Barto, 2006) considered the problem of transfer learning. In their approach, the agent has an internal representation called an agent space that is shared across environments. After solving a source environment, they use supervised learning to project the value function onto the agent space, so it can be used as a shaping reward in future environments. The agent space

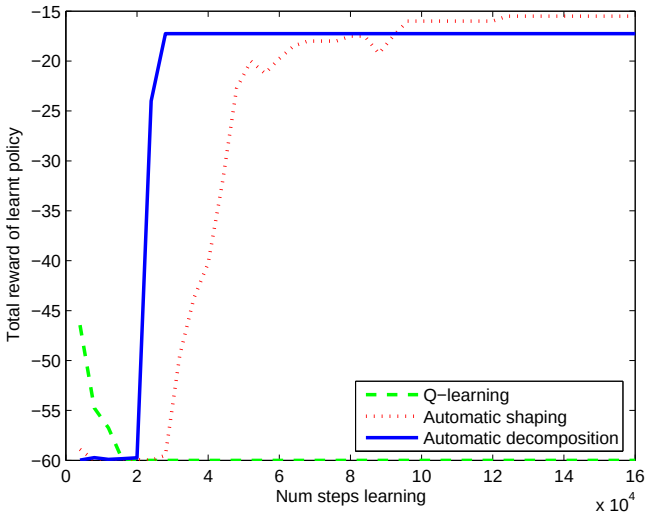

Figure 2: Learning curves for joint navigation problem averaged over 20 trials. Each learnt policy's reward was averaged over 10 runs.

is analogous to an abstract MDP, but their approach differs from ours in that the numerical values are learnt in the source rather than the target problems, which uses fewer samples but requires that the environments are closely related. (Laud & DeJong, 2002) solved a robotic walking problem using a dynamic shaping procedure, in which the parameters of the shaping function were directly adjusted, in contrast to our method, which adjusts the potential function. Their method requires an approximate quality function, which serves as a kind of higher level shaping reward, to be provided as input.

Various types of abstract MDPs have been studied (Hauskrecht et al., 1998; Steinkraus & Kaelbling, 2004). These methods have typically been based on directly solving the abstract MDP and bounding the resulting loss in policy quality in terms of the accuracy of the abstraction. In contrast, our method does not sacrifice optimality; the abstraction accuracy will just affect the speed of convergence. As a result, we can be more aggressive about the abstractions, like in the Othello example.

There is a large literature on distributed and cooperative multiagent reinforcement learning. Most of this work, e.g., (Littman & Boyan, 1993; Stone & Sutton, 2001) assumes that the reward decomposition is provided as input. (Chang et al., 2004) considers a decentralized algorithm for partially observable multieffector problems, in which effector views the global reward as a sum of its local reward plus an underlying Markovian noise process, and estimates the local reward using a Kalman filter. The QUICR algorithm (Agogino & Tumer, 2006) uses a decentralized Q-learning algorithm, where each unit's reward in the Q-learning backup is defined as the amount by which the reward would have

decreased if the unit had moved into an absorbing state instead of doing its part of the action. A merit of QUICR is that it specifies a particular definition of what it means for a unit to be responsible for a reward (with respect to a model that allows counterfactual reasoning). But, unlike our approach, their definition does not minimize the magnitude of the change in the problem's reward structure. For example, if several units must cooperate to achieve a certain subgoal, each one would receive the entire resulting reward. As a result, the subgoal will seem more valuable than it actually is in the context of the overall problem.

6 Discussion and Conclusions

Several interesting directions remain to be pursued. There are two inputs to the potential function learning algorithm: the option set, and the state abstraction function. The learnt shaping function will work best when the options allow near-optimal behaviour (though even if the option set is very suboptimal, the shaping rewards will often "point in the right direction"), and the state abstractions capture the main distinctions made by the true value function. In a transfer-learning setting, an abstraction function and set of compactly described options could be induced from a solution to a source MDP using methods like those in (Yoon et al., 2002). It may also be possible to use the method of (Konidaris & Barto, 2006) to guide exploration when constructing the abstract MDP.

There are also strong connections to hierarchical reinforcement learning (Dietterich, 2000; Andre & Russell, 2002). In our examples, the abstractions are often based on state variables that correspond to higher level tasks, while leaving out the low-level details. Automatic reward decomposition should also be useful when the hierarchy allows concurrent tasks (Marthi et al., 2005).

In this paper, we have presented two algorithms for restructuring reward functions to make a reinforcement learning algorithm's job simpler. The first learns a shaping function, so that rewards occur closer in time to the actions that cause them. The second learns a reward decomposition, so that rewards are assigned to the effectors responsible for them. These quantities are learnt based on input knowledge of a qualitative nature. We believe, therefore, that the algorithms represent a step in the direction of completely autonomous reinforcement learning systems.

Acknowledgments

Thanks to Leslie Kaelbling and Tomas Lozano-Perez for their support during this research.

References

Agogino, A., & Tumer, K. (2006). QUICR-learning for multi-agent coordination. *AAAI 2006*.

Alstrom, J. R. . P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. *ICML 1998*.

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. *Proceedings of the 17th National Conference on Artificial Intelligence* (pp. 119–125).

Bagnell, J., & Ng, A. (2006). On local rewards and scaling distributed reinforcement learning. *Neural Information Processing Systems*. MIT Press.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Athena Scientific.

Chang, Y.-H., Ho, T., & Kaelbling, L. P. (2004). All learning is local: Multi-agent learning in global reward games. In S. Thrun, L. Saul and B. Schölkopf (Eds.), *Advances in neural information processing systems 16*. Cambridge, MA: MIT Press.

Chow, C., & Tsitsiklis, J. (1991). An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE transactions on automatic control*, *36*, 898–914.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, *13*.

Guestrin, C., Koller, D., Parr, R., & Venkataraman, S. (2003). Efficient solution algorithms for factored MDPs. *JAIR*, *19*.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. *Proceedings of the fourteenth conference on Uncertainty in Artificial Intelligence* (pp. 220–229).

Konidaris, G., & Barto, A. (2006). Autonomous shaping: knowledge transfer in reinforcement learning. *Proceedings of the 23rd international conference on Machine learning*.

Laud, A., & DeJong, G. (2002). Reinforcement learning and shaping: Encouraging intended behaviors. *ICML* (pp. 355–362).

Laud, A., & Dejong, G. (2003). The influence of reward on the speed of reinforcement learning: An analysis of shaping. *ICML 2003*.

Li, L., Walsh, T., & Littman, M. (2006). Towards a unified theory of state abstraction for MDPs. *Proceedings of the ninth international symposium on AI and mathematics*.

Littman, M., & Boyan, J. (1993). *A distributed reinforcement learning scheme for network routing* (Technical Report). Carnegie Mellon University, Pittsburgh, PA, USA.

Marthi, B., Russell, S., Latham, D., & Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. *IJCAI 2005*.

Mataric, M. J. (1994). Reward functions for accelerated learning. *ICML 1994*.

Ng, A., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. *ICML 1999*.

Russell, S., & Zimdars, A. (2003). Q-decomposition for reinforcement learning agents. *ICML 2003*.

Schneider, J., Wong, W., Moore, A., & Riedmiller, M. (1999). Distributed value functions. *ICML 1999* (pp. 371–378).

Steinkraus, K., & Kaelbling, L. (2004). *Combining dynamic abstractions in large MDPs* (Technical Report). MIT.

Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. *ICML 2001*.

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

VanEck, N., & VanWezel, M. (2005). *Reinforcement learning and its application to Othello* (Technical Report). Erasmus University.

Wiewiora, E. (2003). Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, *19*, 205–208.

Yoon, S. W., Fern, A., & Givan, R. (2002). Inductive policy selection for first-order mdps. *UAI 2002*.