Apprenticeship Learning Using Linear Programming

Umar Syed USYED@CS.PRINCETON.EDU
Princeton University, Department of Computer Science, 35 Olden Street, Princeton, NJ 08540

Michael Bowling BOWLING@CS.UALBERTA.CA
University of Alberta, Department of Computing Science, Edmonton, Alberta, T6G 2E8 Canada

Robert E. Schapire SCHAPIRE@CS.PRINCETON.EDU
Princeton University, Department of Computer Science, 35 Olden Street, Princeton, NJ 08540

Abstract

In apprenticeship learning, the goal is to learn a policy in a Markov decision process that is at least as good as a policy demonstrated by an expert. The difficulty arises in that the MDP's true reward function is assumed to be unknown. We show how to frame apprenticeship learning as a linear programming problem, and show that using an off-the-shelf LP solver to solve this problem results in a substantial improvement in running time over existing methods — up to two orders of magnitude faster in our experiments. Additionally, our approach produces stationary policies, while all existing methods for apprenticeship learning output policies that are "mixed", i.e. randomized combinations of stationary policies. The technique used is general enough to convert any mixed policy to a stationary policy.

1. Introduction

In apprenticeship learning, as with policy learning for Markov decision processes (MDPs), the objective is to find a good policy for an autonomous agent, called the "apprentice", in a stochastic environment. While the setup of an apprenticeship learning problem is almost identical to that of policy learning in an MDP, there are a few key differences. In apprenticeship learning the true reward function is unknown to the apprentice, but is assumed to be a weighted combination of several known functions. The apprentice is also assumed to have access to demonstrations from another agent, called the "expert", executing a policy in the same environment. The goal of the apprentice is to find a policy that is at least as good as the expert's policy with respect to the true reward function. This is distinct from policy learning, where the goal is to find an optimal policy with respect to the true reward function (which cannot be done in this case because it is unknown).

The apprenticeship learning framework, introduced by Abbeel & Ng (2004), is motivated by a couple of observations about real applications. The first is that reward functions are often difficult to describe exactly, and yet at the same time it is usually easy to specify what the rewards must depend on. A typical example, investigated by Abbeel & Ng (2004), is driving a car. When a person drives a car, it is plausible that her behavior can be viewed as maximizing some reward function, and that this reward function depends on just a few key properties of each environment state: the speed of the car, the position of other cars, the terrain, etc. The second observation is that demonstrations of good policies by experts are often plentiful. This is certainly true in the car driving example, as it is in many other applications.

Abbeel & Ng (2004) assumed that the true reward function could be written as a linear combination of k known functions, and described an iterative algorithm that, given a small set of demonstrations of the expert policy, output an apprentice policy within $O(k \log k)$ iterations that was nearly as good as the expert's policy. Syed & Schapire (2008) gave an algorithm that achieved the same guarantee in $O(\log k)$ iterations. They also showed that by assuming that the linear combination is also a convex one, their algorithm can sometimes find an apprentice policy that is substantially better than the expert's policy. Essentially, the assumption of positive weights implies that the apprentice has some prior knowledge about which policies are better than others, and their algorithm leverages this knowledge.

Existing algorithms for apprenticeship learning share a couple of properties. One is that they each use an algorithm for finding an MDP's optimal policy (e.g. value iteration or policy iteration) as a subroutine. Another is that they output apprentice policies that are "mixtures", i.e. randomized combinations of stationary policies. A stationary policy is a function of just the current environment state.

Our first contribution in this paper is to show that, if one uses the linear programming approach for finding an

Appearing in *Proceedings of the 25^{th} International Conference on Machine Learning*, Helsinki, Finland, 2008.

MDP's optimal policy (Puterman, 1994) as a subroutine, then one can modify Syed & Schapire's (2008) algorithm so that it outputs a stationary policy instead of a mixed policy. Stationary policies are desirable for a number of reasons, e.g. they are simpler to describe, and are more natural and intuitive in terms of the behavior that they prescribe. Moreover, this technique can be straightforwardly applied to any mixed policy, such as the ones output by Abbeel & Ng's (2004) algorithms, to convert it to a stationary policy that earns the same expected cumulative reward.

Our technique leads naturally to the second contribution of this paper, which is the formulation of the apprenticeship learning problem as a linear program. We prove that the solution to this LP corresponds to an apprentice policy that has the same performance guarantees as those produced by existing algorithms, and that the efficiency of modern LP solvers results in a very substantial improvement in running time compared to Syed & Schapire's (2008) algorithm — up to two orders of magnitude in our experiments.

In work closely related to apprenticeship learning, Ratliff, Bagnell & Zinkevich (2006) described an algorithm for learning the true reward function by assuming that the expert's policy is not very different from the optimal policy. They took this approach because they wanted to learn policies that were similar to the expert's policy. In apprenticeship learning, by contrast, the learned apprentice policy can be very different from the expert's policy.

2. Preliminaries

Formally, an *apprenticeship learning problem* $\left(\mathcal{S}, \mathcal{A}, \theta, \alpha, \gamma, R^1 \ldots R^k, \mathcal{D}\right)$ closely resembles a Markov decision process. At each time step t, an autonomous agent occupies a state s_t from a finite set $\mathcal{S}$, and can take an action a_t from a finite set $\mathcal{A}$. When the agent is in state s, taking action a leads to state s' with transition probability $\theta_{sas'} \triangleq \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$. Initial state probabilities are given by $\alpha_s \triangleq \Pr(s_0 = s)$. The agent decides which actions to take based on its policy π, where $\pi_{sa} \triangleq \Pr(a_t = a \mid s_t = s)$. The value of a policy π is given by

$$V(\pi) \triangleq E\left[\sum_{t=0}^{\infty} \gamma^t R_{s_t a_t} \;\middle|\; \alpha, \pi, \theta\right]$$

where $R_{sa} \in [-1, 1]$ is the reward associated with the state-action pair (s, a), and $\gamma \in [0, 1)$ is a discount factor. An optimal policy π^* is one that satisfies $\pi^* = \arg\max_\pi V(\pi)$. We say a policy π is *ϵ-optimal* if $V(\pi^*) - V(\pi) \leq \epsilon$.

A policy π has *occupancy measure* x^π if

$$x^\pi_{sa} = E\left[\sum_{t=0}^{\infty} \gamma^t \mathbf{1}_{(s_t = s \wedge a_t = a)} \;\middle|\; \alpha, \pi, \theta\right] \qquad (1)$$

for all s, a. In other words, x^π_{sa} is the expected (discounted) number of visits to state-action pair (s, a) when following policy π.

Unlike an MDP, in apprenticeship learning the true reward function R is unknown. Instead, we are given *basis reward functions*[1] $R^1 \ldots R^k$, where R^i_{sa} is the reward of state-action pair (s, a) with respect to the ith basis reward function. We assume that the true reward function R is an unknown convex combination w^* of the basis reward functions, i.e., for all s, a

$$R_{sa} = \sum_i w^*_i R^i_{sa}$$

where the unknown weights satisfy $w^*_i \geq 0$ and $\sum_i w^*_i = 1$. Each basis reward function R^i has a corresponding *basis value function* $V^i(\pi)$ given by

$$V^i(\pi) \triangleq E\left[\sum_{t=0}^{\infty} \gamma^t R^i_{s_t a_t} \;\middle|\; \alpha, \pi, \theta\right].$$

Given the assumption of positive weights, the value of k can be viewed as a measure of how much the apprentice knows about the true reward function. If $k = 1$, the (only) basis value of a policy is equal to its true value, and the situation reduces to a traditional MDP. At the other extreme, if the ith basis reward function is just an indicator function for the ith state-action pair, then $k = |\mathcal{SA}|$, and the basis values of a policy are equal to its occupancy measure. In this situation, the apprentice knows essentially nothing about which policies are better than others.

The positive weight assumption also implies that if for state-action pairs (s, a) and (s', a') we have $R^i_{sa} \geq R^i_{s'a'}$ for all i, then $R_{sa} \geq R_{s'a'}$. So the basis rewards themselves can encode prior knowledge about the true rewards. If we wish not to assert any such prior knowledge, we can simply add the negative of each basis reward function to the original set, thereby at most doubling the number of basis reward functions.

We also assume that we are given a data set $\mathcal{D}$ of M i.i.d. sample trajectories from an *expert policy* π^E executing in the environment, where the mth trajectory is a sequence of state-action pairs visited by the expert, i.e., $(s^m_0, a^m_0, s^m_1, a^m_1, \ldots, s^m_H, a^m_H)$. For simplicity, we assume that all sample trajectories are truncated to the same length H.

The goal of apprenticeship learning (Abbeel & Ng, 2004) is to find an *apprentice policy* π^A such that

$$V(\pi^A) \geq V(\pi^E) \qquad (2)$$

even though the true value function $V(\pi)$ is unknown (since the true reward function is unknown).

2.1. A More Refined Goal

By our assumptions about the reward functions (and the linearity of expectation), we have

$$V(\pi) = \sum_i w^*_i V^i(\pi).$$

[1] In (Abbeel & Ng, 2004) and (Syed & Schapire, 2008) these functions were called *features*, but we believe that the present terminology is better suited for conveying these functions' role.

Consequently, for any policy π, the smallest possible difference between $V(\pi)$ and $V(\pi^E)$ is $\min_i V^i(\pi) - V^i(\pi^E)$, because in the worst-case, $w_i^* = 1$ for the minimizing i. Based on this observation, Syed & Schapire (2008) proposed finding an apprentice policy π^A that solves the maximin objective

$$v^* = \max_{\pi} \min_i V^i(\pi) - V^i(\pi^E). \tag{3}$$

Note that if π^A is a solution to (3), then $V(\pi^A) \geq V(\pi^E) + v^*$ (because $v^* = \min_i V^i(\pi^A) - V^i(\pi^E) \leq V(\pi^A) - V(\pi^E)$). We also have $v^* \geq 0$ (because $\pi = \pi^E$ is available in (3)). Therefore π^A satisfies the goal of apprenticeship learning given in (2).

Syed & Schapire (2008) showed that in some cases where $V(\pi^E)$ is small, v^* is large, and so adding v^* to the lower bound in (2) serves as a kind of insurance against bad experts. Our algorithms also produce apprentice policies that achieve this more refined goal.

2.2. Estimating the Expert Policy's Values

Our algorithms require knowledge of the basis values of the expert's policy. From the expert's sample trajectories $\mathcal{D}$, we can form an estimate $\widehat{V}^{i,E}$ of $V^i(\pi^E)$ as follows:

$$V^i(\pi^E) \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t=0}^{H} \gamma^t R^i_{s_t^m a_t^m} \triangleq \widehat{V}^{i,E}.$$

Clearly, as the number of sample trajectories M and the truncation length H increase, the error of this estimate will decrease. Thus the issue of accurately estimating $V^i(\pi^E)$ is related to *sample* complexity, while in this work we are primarily concerned with *computational* complexity. To make our presentation cleaner, we will assume that $\mathcal{D}$ is large enough to yield an estimate $\widehat{V}^{i,E}$ of $V^i(\pi^E)$ such that $|\widehat{V}^{i,E} - V^i(\pi^E)| \leq \epsilon$, for all i. We call such an estimate ϵ-*good*. The sample complexity of apprenticeship learning is treated in (Syed & Schapire, 2008).

2.3. Policy Types

Unless otherwise noted, a policy π is presumed to be stationary, i.e., π_{sa} is the probability of taking action a in state s. One exception is a *mixed policy*. A mixed policy $\tilde{\pi}$ is defined by a set of ordered pairs $\{(\pi^j, \lambda^j)\}_{j=1}^N$. The policy $\tilde{\pi}$ is followed by choosing at time 0 one of the stationary policies π^j, each with probability λ^j, and then following that policy exclusively thereafter. The value of a mixed policy is the expected value of the stationary policies it comprises, i.e.,

$$V(\tilde{\pi}) = E\left[V(\pi^j)\right] = \sum_{j=1}^{N} \lambda^j V(\pi^j), \text{ and}$$

$$V^i(\tilde{\pi}) = E\left[V^i(\pi^j)\right] = \sum_{j=1}^{N} \lambda^j V^i(\pi^j).$$

3. Multiplicative Weights Algorithm for Apprenticeship Learning

Syed & Schapire (2008) observed that solving the objective in (3) is equivalent to finding an optimal strategy in a certain two-player zero-sum game. Because the size of this game's matrix is exponential in the number of states $|\mathcal{S}|$, they adapted a multiplicative weights method for solving extremely large games. The resulting MWAL (Multiplicative Weights Apprenticeship Learning) algorithm is described in Algorithm 1 below.

Algorithm 1 MWAL algorithm

1: **Given:** $\mathcal{S}, \mathcal{A}, \theta, \alpha, \gamma, R^1 \ldots R^k, \mathcal{D}$.
2: Using the expert's sample trajectories $\mathcal{D}$, compute an ϵ-good estimate $\widehat{V}^{i,E}$ of $V^i(\pi^E)$, for all i.
3: Let $\beta = \left(1 + \sqrt{\frac{2 \log k}{T}}\right)^{-1} \in (0, 1]$.
4: Initialize $w_i^1 = \frac{1}{k}$, for $i = 1 \ldots k$.
5: **for** $t = 1 \ldots T$ **do**
6: Compute ϵ-optimal policy π^t for reward function $R_{sa} = \sum_i w_i^t R^i_{sa}$.
7: Compute ϵ-good estimate $\widehat{V}^{i,t}$ of $V^i(\pi^t)$, for $i = 1 \ldots k$.
8: Let $w_i^{t+1} = w_i^t \beta^{\widehat{V}^{i,t} - \widehat{V}^{i,E}}$, for $i = 1 \ldots k$.
9: Renormalize w.
10: **end for**
11: **Return:** Let apprentice policy π^A be the mixed policy defined by $\{(\pi^t, \frac{1}{T})\}_{t=1}^T$.

In each iteration of the MWAL algorithm, an optimal policy π^t is computed with respect to the current weight vector w^t. Then the weights are updated so that w_i is increased/decreased if π^t is a bad/good policy (relative to π^E) with respect to the ith basis reward function.

The next theorem bounds the number of iterations T required for the MWAL algorithm to produce a good apprentice policy. The computational complexity of each iteration is discussed in Section 3.1.

Theorem 1 (Syed & Schapire (2008)). *Let π^A be the mixed policy returned by the MWAL algorithm. If*

$$T \geq O\left(\frac{\log k}{(\epsilon(1-\gamma))^2}\right)$$

then

$$V(\pi^A) \geq V(\pi^E) + v^* - O(\epsilon)$$

where $v^* = \max_\pi \min_i V^i(\pi) - V^i(\pi^E)$.

3.1. MWAL-VI and MWAL-PI

The specification of the MWAL algorithm is somewhat open-ended. Step 6 requires finding an ϵ-optimal policy in an MDP, and Step 7 requires computing ϵ-good estimates of the basis values of that policy. There are several procedures available for accomplishing each of these steps, with each option leading to a different variant of the basic algorithm.

We briefly describe some natural options, and remark on their implications for the overall computational complexity of the MWAL algorithm.

In Step 6, we can find the optimal policy using value iteration (Puterman, 1994), which has a worst-case running time of $O\left(\log_\gamma\left(1/\epsilon(1-\gamma)\right)|\mathcal{S}|^2|\mathcal{A}|\right)$. We can also use value iteration to compute the k basis values in Step 7 (this is sometimes called "policy evaluation"), which implies a worst-case running time of $O\left(k\log_\gamma\left(1/\epsilon(1-\gamma)\right)|\mathcal{S}|^2\right)$. We call this variant the MWAL-VI algorithm.

Another choice for Step 6 is to find the optimal policy using policy iteration (Puterman, 1994). No polynomial time bound for policy iteration is known; however, in practice it has often been observed to be faster than value iteration. We call this variant the MWAL-PI algorithm. In Section 8, we present experiments comparing these algorithms to the ones described later in the paper.

4. Dual Methods for MDPs

As we previously observed, the MWAL algorithm must repeatedly find the optimal policy in an MDP, and this task is usually accomplished via classic iterative techniques such as value iteration and policy iteration. However, there are other techniques available for solving MDPs, and in this work we show that they can lead to better algorithms for apprenticeship learning. Consider the following linear program:

$$\max_x \sum_{s,a} R_{sa}x_{sa} \tag{4}$$

such that

$$\sum_a x_{sa} = \alpha_s + \gamma\sum_{s',a} x_{s'a}\theta_{s'as} \tag{5}$$

$$x_{sa} \geq 0 \tag{6}$$

It is well-known (Puterman, 1994) that if x^* is a solution to (4) - (6), then $\pi^*_{sa} = \frac{x^*_{sa}}{\sum_a x^*_{sa}}$ is an optimal policy, and x^* is the occupancy measure of π^*. Often (5) - (6) are called the *Bellman flow constraints*.

The linear program in (4) - (6) is actually the dual of the linear program that is typically used to find an optimal policy in an MDP. Accordingly, solving (4) - (6) is often called the *Dual LP* method of solving MDPs .

Having found an optimal policy by the Dual LP method, computing its values is straightforward. The next lemma follows immediately from the definitions of the occupancy measure and value of a policy.

Lemma 1. *If policy π has occupancy measure x^π, then $V(\pi) = \sum_{s,a} R_{sa}x^\pi_{sa}$ and $V^i(\pi) = \sum_{s,a} R^i_{sa}x^\pi_{sa}$.*

5. Main Theoretical Tools

Recall that the MWAL algorithm produces mixed policies. In Sections 6 and 7, we will present algorithms that achieve the same theoretical guarantees as the MWAL algorithm, but produce stationary policies (and are also faster). To prove the correctness of these algorithms, we need to show that every mixed policy has an equivalent stationary policy.

In Section 4, we said that the Dual LP method of solving an MDP outputs the occupancy measure of an optimal policy. In fact, *all* x that satisfy the Bellman flow constraints (5) - (6) are the occupancy measure of some stationary policy, as the next theorem shows.

Theorem 2. *Let x satisfy the Bellman flow constraints* (5) - (6)*, and let $\pi_{sa} = \frac{x_{sa}}{\sum_a x_{sa}}$ be a stationary policy. Then x is the occupancy measure for π. Conversely, if π is a stationary policy such that x is its occupancy measure, then $\pi_{sa} = \frac{x_{sa}}{\sum_a x_{sa}}$ and x satisfies the Bellman flow constraints.*

An equivalent result as Theorem 2 is given in (Feinberg & Schwartz, 2002), p. 178. For completeness, a simple and direct proof is contained in the Appendix.

The Bellman flow constraints make it very easy to show that, for every mixed policy, there is a stationary policy that has the same value.

Theorem 3. *Let $\tilde{\pi}$ be a mixed policy defined by $\{(\pi^j, \lambda^j)\}_{j=1}^N$, and let x^j be the occupancy measure of π^j, for all j. Let $\hat{\pi}$ be a stationary policy where*

$$\hat{\pi}_{sa} = \frac{\sum_j \lambda^j x^j_{sa}}{\sum_a \sum_j \lambda^j x^j_{sa}}.$$

Then $V(\hat{\pi}) = V(\tilde{\pi})$.

Proof. By Theorem 2, x^j satisfies the Bellman flow constraints (5) - (6) for all j. Let $\hat{x}_{sa} = \sum_j \lambda^j x^j_{sa}$. By linearity, $\hat{x}$ also satisfies the Bellman flow constraints. Hence, by Theorem 2, the stationary policy $\hat{\pi}$ defined by $\hat{\pi}_{sa} = \frac{\hat{x}_{sa}}{\sum_a \hat{x}_{sa}}$ has occupancy measure $\hat{x}$. Therefore,

$$V(\hat{\pi}) = \sum_{s,a} R_{sa}\hat{x}_{sa} = \sum_j \lambda^j \sum_{s,a} R_{sa}x^j_{sa} = \sum_j \lambda^j V(\pi^j)$$
$$= V(\tilde{\pi}).$$

where these equalities use, in order: Lemma 1; the definition of $\hat{x}$; Lemma 1; the definition of a mixed policy. □

6. MWAL-Dual Algorithm

In this section, we will make a minor modification to the MWAL algorithm so that it outputs a stationary policy instead of a mixed policy.

Recall that the MWAL algorithm requires, in Steps 6 and 7, a way to compute an optimal policy and its basis values, but that no particular methods are prescribed. Our proposal is to use the Dual LP method in Step 6 to find the occupancy measure x_t of a policy π_t that is ϵ-optimal for reward function $R_{sa} = \sum_i w^t_i R^i_{sa}$. Then in Step 7 we let

$\widehat{V}^{i,t} = \sum_{s,a} R^i_{sa} x^t_{sa}$, for $i = 1 \dots k$. Note that Lemma 1 implies $\widehat{V}^{i,t} = V^i(\pi^t)$.

Now we can apply Theorem 3 to combine all the policies computed during the MWAL algorithm into a single stationary apprentice policy. This amounts to changing Step 11 to the following:

> **Return:** Let apprentice policy π^A be the stationary policy defined by
>
> $$\pi^A_{sa} = \frac{\frac{1}{T}\sum_t x^t_{sa}}{\sum_a \frac{1}{T}\sum_t x^t_{sa}}.$$

We call this modified algorithm the MWAL-Dual algorithm, after the method it uses to compute optimal policies.

It is straightforward to show that these changes to the MWAL algorithm do not affect its performance guarantee.

Theorem 4. *Let π^A be the stationary policy returned by the MWAL-Dual algorithm. If*

$$T \geq O\left(\frac{\log k}{(\epsilon(1-\gamma))^2}\right)$$

then

$$V(\pi^A) \geq V(\pi^E) + v^* - O(\epsilon)$$

where $v^* = \max_\pi \min_i V^i(\pi) - V^i(\pi^E)$.

Proof. By Theorem 3, the stationary policy returned by the MWAL-Dual algorithm has the same value as the mixed policy returned by the original MWAL algorithm. Hence the guarantee in Theorem 1 applies to the MWAL-Dual algorithm as well. □

Of course, the trick used here to convert a mixed policy to a stationary one is completely general, provided that the occupancy measures of the component policies can be computed. For example, this technique could be applied to the mixed policy output by the algorithms due to Abbeel & Ng (2004).

Let $T(n)$ be the worst-case running time of an LP solver on a problem with at most n constraints and variables.[2] For a typical LP solver, $T(n) = O(n^{3.5})$ (Shu-Cherng & Puthenpura, 1993), although they tend to be much faster in practice. Using this notation, we can bound the running time of Steps 6 and 7 in the MWAL-Dual algorithm. Finding an optimal policy using the Dual LP method takes $T(|\mathcal{S}||\mathcal{A}|)$ time. And by Lemma 1, given the occupancy measure of a policy, we can compute its basis values in $O(k|\mathcal{S}||\mathcal{A}|)$ time.

7. LPAL Algorithm

We now describe a way to use the Bellman flow constraints to find a good apprentice policy in a much more direct fashion than the MWAL algorithm. Recall the objective function proposed in (Syed & Schapire, 2008) for solving apprenticeship learning:

$$v^* = \max_\pi \min_i V^i(\pi) - V^i(\pi^E) \tag{7}$$

We observed earlier that, if π^A is a solution to (7), then $V(\pi^A) \geq V(\pi^E) + v^*$, and that $v^* \geq 0$. In this section, we describe a linear program that solves (7). In Section 8, we describe experiments that show that this approach is much faster than the MWAL algorithm, although it does have some disadvantages, which we also illustrate in Section 8.

Our LPAL (Linear Programming Apprenticeship Learning) algorithm is given in Algorithm 2. The basic idea is to use the Bellman flow constraints (5) - (6) and Lemma 1 to define a feasible set containing all (occupancy measures of) stationary policies whose basis values are above a certain lower bound, and then maximize this bound.

Algorithm 2 LPAL algorithm

1: **Given:** $\mathcal{S}, \mathcal{A}, \theta, \alpha, \gamma, R^1 \dots R^k, \mathcal{D}$.
2: Using the expert's sample trajectories $\mathcal{D}$, compute an ϵ-good estimate $\widehat{V}^{i,E}$ of $V^i(\pi^E)$, for all i.
3: Find a solution (B^*, x^*) to this linear program:

$$\max_{B,x} \quad B \tag{8}$$

such that

$$B \leq \sum_{s,a} R^i_{sa} x_{sa} - \widehat{V}^{i,E} \tag{9}$$

$$\sum_a x_{sa} = \alpha_s + \gamma \sum_{s',a} x_{s'a}\theta_{s'as} \tag{10}$$

$$x_{sa} \geq 0 \tag{11}$$

4: **Return:** Let apprentice policy π^A be the stationary policy defined by

$$\pi^A_{sa} = \frac{x^*_{sa}}{\sum_a x^*_{sa}}.$$

Theorem 5. *Let π^A be the stationary policy returned by the LPAL algorithm. Then*

$$V(\pi^A) \geq V(\pi^E) + v^* - O(\epsilon)$$

where $v^* = \max_\pi \min_i V^i(\pi) - V^i(\pi^E)$.

Proof. By Theorem 2, the Bellman flow constraints (10) - (11) imply that all feasible x correspond to the occupancy measure of some stationary policy π. Using this fact and Lemma 1, we conclude that solving the linear program is equivalent to finding (B^*, π^A) such that

$$B^* = \min_i V^i(\pi^A) - \widehat{V}^{i,E}$$

and B^* is as large as possible. Since $|\widehat{V}^{i,E} - V^i(\pi^E)| \leq \epsilon$ for all i, we know that $B^* \geq v^* - \epsilon$. Together with (9) and

[2] Technically, the time complexity of a typical LP solver also depends on the number of bits in the problem representation.

Lemma 1 this implies

$$V^i(\pi^A) = \sum_{s,a} R^i_{sa} x^*_{sa} \geq \widehat{V}^{i,E} + B^* \geq V^i(\pi^E) + v^* - 2\epsilon.$$

□

Note that the *overall* worst-case running time of the LPAL algorithm is $T(|\mathcal{S}||\mathcal{A}| + k)$, where $T(n)$ is the complexity of an LP solver.

8. Experiments

8.1. Gridworld

We tested each algorithm in gridworld environments that closely resemble those in the experiments of Abbeel & Ng (2004). Each gridworld is an $N \times N$ square of states. Movement is possible in the four compass directions, and each action has a 30% chance of causing a transition to a random state. Each gridworld is partitioned into several square regions, each of size $M \times M$. We always choose M so that it evenly divides N, so that each gridworld has $k = (\frac{N}{M})^2$ regions. Each gridworld also has k basis reward functions, where the ith basis reward function R^i is a 0-1 indicator function for the ith region.

For each gridworld, in each trial, we randomly chose a sparse weight vector w^*. Recall that the true reward function has the form $R(s) = \sum_i w^*_i R^i(s)$, so in these experiments the true reward function just encodes that some regions are more desirable than others. In each trial, we let the expert policy π^E be the optimal policy with respect to R, and then supplied the basis values $V^i(\pi^E)$, for all i, to the MWAL-VI, MWAL-PI, MWAL-Dual and LPAL algorithms.[3]

Our experiments were run on an ordinary desktop computer. We used the Matlab-based `CVX` package (Grant & Boyd, 2008) for our LP solver. Each of the values in the tables below is the time, in seconds, that the algorithm took to find an apprentice policy π^A such that $V(\pi^A) \geq 0.95V(\pi^E)$. Each running time is the average of 10 trials.

Table 1. Time (sec) to find π^A s. t. $V(\pi^A) \geq 0.95V(\pi^E)$

Gridworld Size	MWAL-VI (sec)	MWAL-PI (sec)	MWAL-Dual (sec)	LPAL (sec)
16×16	6.43	5.78	46.99	1.46
24×24	14.45	10.27	90.16	1.55
32×32	27.23	15.04	247.38	2.76
48×48	61.37	35.33	791.61	8.62
64×64	114.12	85.26	3651.70	30.52
128×128	406.24	307.58	4952.74	80.21
256×256	1873.93	1469.56	29988.85	588.60

[3]Typically in practice, π_E will be unknown, and so the basis values would need to be estimated from the data set of expert sample trajectories $\mathcal{D}$. However, since we are primarily concerned with computational complexity in this work, and not sample complexity, we sidestep this issue and just compute each $V^i(\pi^E)$ directly.

Table 2. Time (sec) to find π^A s. t. $V(\pi^A) \geq 0.95V(\pi^E)$

Gridworld Size	Number of Regions	MWAL-VI (sec)	MWAL-PI (sec)	MWAL-Dual (sec)	LPAL (sec)
24×24	64	14.45	10.27	90.16	1.55
	144	32.33	20.06	97.58	2.64
	576	129.87	75.81	120.82	1.86
32×32	64	27.23	15.04	247.38	2.76
	256	107.11	60.24	270.71	8.43
	1024	440.64	267.12	361.36	4.75
48×48	64	61.37	35.33	791.61	8.62
	144	135.83	79.88	800.23	11.42
	256	244.46	150.08	815.66	16.89
	576	575.34	352.15	847.38	16.33
	2304	2320.71	1402.10	1128.32	11.14

In the first set of experiments (Table 1), we tested the algorithms in gridworlds of varying sizes, while keeping the number of regions in each gridworld fixed (64 regions). Recall that the number of regions is equal to the number of basis reward functions. In the next set of experiments (Table 2), we varied the number of regions while keeping the size of the gridworld fixed.

Several remarks about these results are in order. For every gridworld size and every number of regions, the LPAL algorithm is substantially faster than the other algorithms — in some cases two orders of magnitude faster. As we previously noted, LP solvers are often much more efficient than their theoretical guarantees. Interestingly, in Table 2, the running time for LPAL eventually decreases as the number of regions increases. This may be because the number of constraints in the linear program increases with the number of regions, and more constraints often make a linear program problem easier to solve.

Also, the MWAL-Dual algorithm is much slower than the other algorithms. We suspect this is only because the MWAL-Dual algorithm calls the LP solver in each iteration (unlike the LPAL algorithm, which calls it just once), and there is substantial overhead to doing this. Modifying MWAL-Dual so that it uses the LP solver as less of a black-box may be a way to alleviate this problem.

8.2. Car driving

In light of the results from the previous section, one might reasonably wonder whether there is any argument for using an algorithm other than LPAL. Recall that, in those experiments, the expert's policy was an optimal policy for the unknown reward function. In this section we explore the behavior of each algorithm when this is not the case, and find that MWAL produces better apprentice policies than LPAL. Our experiments were run in a car driving simulator modeled after the environments in (Abbeel & Ng, 2004) and (Syed & Schapire, 2008).

The task in our driving simulator is to navigate a car on a busy three-lane highway. The available actions are to move left, move right, drive faster, or drive slower. There are three basis reward functions that map each environment state to a numerical reward: collision (0 if contact with an-

other car, and 1/2 otherwise), off-road (0 if on the grass, and 1/2 otherwise), and speed (1/2, 3/4 and 1 for each of the three possible speeds, with higher values corresponding to higher speeds). The true reward function is assumed to be some unknown weighted combination w^* of the basis reward functions. Since the weights are assumed to be positive, by examining the basis reward functions we see that the true reward function assigns higher reward to states that are intuitively "better".

We designed three experts for these experiments, described in Table 3. Each expert is optimal for one of the basis reward functions, and mediocre for the other two. Therefore each expert policy π^E is an optimal policy if $w^* = w^E$, where w^E is the weight vector that places all weight on the basis reward function for which π^E is optimal. At the same time, each π^E is very likely to be suboptimal for a randomly chosen w^*.

We used the MWAL and LPAL algorithms to learn apprentice policies from each of these experts.[4] The results are presented in Table 4. We let $\gamma = 0.9$, so the maximum value of the basis value function corresponding to speed was 10, and for the others it was 5. Each of the reported policy values for randomly chosen w^* was averaged over 10,000 uniformly sampled w^*'s. Notice that for each expert, when w^* is chosen randomly, MWAL outputs better apprentice policies than LPAL.

Table 3. Expert types

	Speed	Collisions (per sec)	Off-roads (per sec)
"Fast" expert	Fast	1.1	10
"Avoid" expert	Slow	0	10
"Road" expert	Slow	1.1	0

Table 4. Driving simulator experiments.

Expert type	Algorithm used	$w^* = w^E$		w^* chosen randomly	
		$V(\pi^A)$	$V(\pi^E)$	$V(\pi^A)$	$V(\pi^E)$
"Fast"	MWAL	10	10	9.83	8.25
	LPAL	10	10	8.84	8.25
"Avoid"	MWAL	5	5	8.76	6.32
	LPAL	5	5	7.26	6.32
"Road"	MWAL	5	5	9.74	7.49
	LPAL	5	5	8.12	7.49

9. Conclusion and Future Work

Each of the algorithms for apprenticeship learning presented here have advantages and disadvantages that make them each better suited to different situations. As our experiments showed, the LPAL algorithm is much faster than any of the MWAL variants, and so is most appropriate for problems with large state spaces or many basis reward functions. And unlike the original MWAL algorithm, it produces a stationary policy, which make it a good choice whenever a simple and easily interpretable apprentice policy is desired. On the other hand, we also presented evidence that LPAL performs poorly when the expert policy is far from an optimal policy for the true reward function. If one suspects in advance that this may be the case, then one of the MWAL variants would be a better choice for a learning algorithm. Among these variants, only MWAL-Dual produces a stationary policy, although it has the drawback of being the slowest algorithm that we tested.

Although the theoretical performance guarantees of both the MWAL and LPAL algorithm are identical, the results in Table 4 suggest that the two algorithms are not equally effective. It seems possible that the current theoretical guarantees for the MWAL algorithm are not as strong as they could be. Investigation of this possibility is ongoing work.

One way to describe the poor performance of the LPAL algorithm versus MWAL is to say that, when there are several policies that are better than the expert's policy, the LPAL algorithm fails to optimally break these "ties". This characterization suggests that recent techniques for computing robust strategies in games (Johanson et al., 2008) may be an avenue for improving the LPAL algorithm.

It would also be interesting to examine practically and theoretically how apprenticeship learning can be combined with MDP approximation techniques. In particular, the dual linear programming approach in this work might combine nicely with recent work on stable MDP approximation techniques based on the dual form (Wang et al., 2008).

[4] Each of the MWAL variants behaved in exactly the same way in this experiment. The results presented are for the MWAL-PI variant.

Acknowledgements

We would like to thank Michael Littman, Warren Powell, Michele Sebag and the anonymous reviewers for their helpful comments. This work was supported by the NSF under grant IIS-0325500.

References

Abbeel, P., & Ng, A. (2004). Apprenticeship learning via inverse reinforcement learning. *Proceedings of the International Conference on Machine Learning*.

Feinberg, E. A., & Schwartz, A. (2002). *Handbook of Markov Decision Processes: Methods and Applications*. Springer.

Grant, M., & Boyd, S. (2008). CVX: Matlab software for disciplined convex programming (web page and software). http://stanford.edu/∼boyd/cvx.

Horn, R. A., & Johnson, C. R. (1985). *Matrix Analysis*. Cambridge University Press.

Johanson, M., Zinkevich, M., & Bowling, M. (2008). Computing robust counter-strategies. *Advances in Neural Information Processing Systems*.

Puterman, M. L. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley and Sons.

Ratliff, N. D., Bagnell, J. A., & Zinkevich, M. A. (2006). Maximum margin planning. *Proceedings of the International Conference on Machine Learning*.

Shu-Cherng, & Puthenpura, S. (1993). *Linear Optimization and Extensions: Theory and Algorithms*. Prentice Hall.

Syed, U., & Schapire, R. E. (2008). A game-theoretic approach to apprenticeship learning. *Advances in Neural Information Processing Systems*.

Wang, T., Lizotte, D., Bowling, M., & Schuurmans, D. (2008). Stable dual dynamic programming. *Advances in Neural Information Processing Systems*.

10. Appendix

This is a proof of Theorem 2. Before proceeding, we introduce another linear system. For any stationary policy π, the *π-specific Bellman flow constraints* are given by the following linear system in which the x_{sa} variables are unknown:

$$x_{sa} = \pi_{sa}\alpha_s + \pi_{sa}\gamma \sum_{s',a'} x_{s'a'}\theta_{s'a's} \tag{12}$$

$$x_{sa} \geq 0 \tag{13}$$

The next lemma shows that π-specific Bellman flow constraints have a solution.

Lemma 2. *For any stationary policy π, the occupancy measure x^π of π satisfies the π-specific Bellman flow constraints* (12) - (13).

Proof. Clearly, x^π_{sa} is non-negative for all s, a, and so (13) is satisfied. As for (12), we simply plug in the definition of x^π_{sa} from (1). (In the following derivation, all the expectations and probabilities are conditioned on α, θ, and π. They have been omitted from the notation for brevity.)

$$\begin{aligned}
x^\pi_{sa} &= E\left[\sum_{t=0}^{\infty} \gamma^t \mathbf{1}_{(s_t=s \wedge a_t=a)}\right] \\
&= \sum_{t=0}^{\infty} \gamma^t \Pr(s_t = s, a_t = a) \\
&= \pi_{sa}\alpha_s + \sum_{t=0}^{\infty} \gamma^{t+1} \Pr(s_{t+1} = s, a_{t+1} = a) \\
&= \pi_{sa}\alpha_s \\
&\quad + \sum_{t=0}^{\infty} \gamma^{t+1} \sum_{s',a'} \Pr(s_t = s', a_t = a', s_{t+1} = s, a_{t+1} = a) \\
&= \pi_{sa}\alpha_s + \sum_{t=0}^{\infty} \gamma^{t+1} \sum_{s',a'} \Pr(s_t = s', a_t = a') \cdot \theta_{s'a's}\pi_{sa} \\
&= \pi_{sa}\alpha_s + \pi_{sa}\gamma \sum_{s',a'} E\left[\sum_{t=0}^{\infty} \gamma^t \mathbf{1}_{(s_t=s' \wedge a_t=a')}\right] \theta_{s'a's} \\
&= \pi_{sa}\alpha_s + \pi_{sa}\gamma \sum_{s',a'} x^\pi_{s'a'}\theta_{s'a's}
\end{aligned}$$

□

Now we show that the solution to the π-specific Bellman flow constraints given by Lemma 2 is unique.

Lemma 3. *For any stationary policy π, the π-specific Bellman flow constraints* (12) - (13) *have at most one solution.*

Proof. Define the matrix

$$A_{(sa,s'a')} \triangleq \begin{cases} 1 - \gamma\theta_{s'a's}\pi_{sa} & \text{if } (s,a) = (s',a') \\ -\gamma\theta_{s'a's}\pi_{sa} & \text{otherwise.} \end{cases}$$

and the vector $b_{sa} \triangleq \pi_{sa}\alpha_s$. (Note that A and b are indexed by state-action pairs.) We can re-write (12) - (13) equivalently as

$$Ax = b \tag{14}$$

$$x \geq 0 \tag{15}$$

The matrix A is column-wise strictly diagonally dominant. This is because $\sum_{s'} \theta_{sas'} = 1$, $\sum_a \pi_{sa} = 1$ and $\gamma < 1$, so for all s', a'

$$\begin{aligned}
&\sum_{s,a} \gamma\theta_{s'a's}\pi_{sa} = \gamma < 1 \\
\Rightarrow \quad & 1 - \gamma\theta_{s'a's'}\pi_{s'a'} > \sum_{(s,a)\neq(s',a')} \gamma\theta_{s'a's}\pi_{sa} \\
\Rightarrow \quad & |A_{(s'a',s'a')}| > \sum_{(s,a)\neq(s',a')} |A_{(sa,s'a')}|.
\end{aligned}$$

where the last line is the definition of column-wise strict diagonal dominance. This implies that A is non-singular (Horn & Johnson, 1985), so (14) - (15) has at most one solution. □

We are now ready to prove Theorem 2.

Proof of Theorem 2. For the first direction of the theorem, we assume that x satisfies the Bellman flow constraints (5) - (6), and that $\pi_{sa} = \dfrac{x_{sa}}{\sum_a x_{sa}}$. Therefore,

$$\pi_{sa} = \frac{x_{sa}}{\alpha_s + \gamma \sum_{s',a'} x_{s'a'}\theta_{s'a's}}. \tag{16}$$

Clearly x is a solution to the π-specific Bellman flow constraints (12) - (13), and Lemmas 2 and 3 imply that x is the occupancy measure of π.

For the other direction of the theorem, we assume that x is the occupancy measure of π. Lemmas 2 and 3 imply that x is the unique solution to the π-specific Bellman flow constraints (12) - (13). Therefore, π is given by (16). And since $\sum_a \pi_{sa} = 1$, we have

$$\frac{\sum_a x_{sa}}{\alpha_s + \gamma \sum_{s',a'} x_{s'a'}\theta_{s'a's}} = 1$$

which can be rearranged to show that x satisfies the Bellman flow constraints, and also combined with (16) to show that $\pi_{sa} = \dfrac{x_{sa}}{\sum_a x_{sa}}$. □