Sparse Sampling Methods for Planning and Learning in Large and Partially Observable Markov Decision Processes (DRAFT — DO NOT DISTRIBUTE)

Michael Kearns
AT&T Labs
mkearns@research.att.com

Yishay Mansour
AT&T Labs
and
Tel Aviv University
mansour@research.att.com

Andrew Y. Ng
UC Berkeley
ang@cs.berkeley.edu

November 25, 1998

Abstract

We apply methods of sparse sampling from a generative model and the tools of uniform convergence to develop new algorithms for planning and learning in Markov decision processes with very large state spaces and partial observability. Our results include an algorithm for computing a near-optimal policy in an arbitrarily large MDP whose running time has no dependence on the number of states; algorithms for generating experience in a POMDP in order to efficiently choose a near-best strategy from a given class of strategies; and a number of suggestions for implementing our algorithms in practice.

1 Introduction

In the past decade, Markov decision processes (MDP's) and reinforcement learning have become a standard framework for planning and learning under uncertainty within the artificial intelligence literature. The desire to attack problems of increasing complexity with this formalism has recently led researchers to focus particular attention on two important settings that arise in many natural applications: the case of (exponentially) large state spaces, and the case of partial observability. A variety of interesting algorithmic suggestions have been made for coping with these two settings. For instance, function approximation [SB98] is a well-studied proposal for learning parametric value functions in large state spaces, and approximations to belief-state planning have been formulated for partially observable MDP's (POMDP's). Despite this activity, a number of rather basic questions about planning and learning in large or partially observable environments have remain unanswered, and we address two such questions here:

- How does the complexity of planning in MDP's scale with the number of states?
- How should we efficiently generate experience in a POMDP in order to choose a good strategy from a limited class?

There are three common themes run throughout our results. The first arises from the fact that as an (PO)MDP becomes very large or complex, the classical (planning) assumption that the (PO)MDP is given explicitly by tables of transition probabilities and rewards becomes infeasible. Thus, we instead consider a setting in which we are given access to a *generative model*, or simulator, of the MDP. Since a generative model provides less information than explicit tables, but more information than a continuous trajectory of experience, results obtained via a generative model blur the distinction between what is typically called "planning" and "learning" in (PO)MDP's.

The second theme is the power of algorithms that do only *sparse sampling* from the generative model, where by sparse we mean a number of samples that is *independent of the size or complexity of the PO(MDP)*. In particular, our main results include:

- An algorithm for near-optimal planning in an arbitrary MDP that accesses a generative model, and takes per-state computation time that has *no dependence* on the MDP (and thus applies even to infinite state spaces);
- Algorithms for generating trajectories in a POMDP that permit a near-best strategy to be chosen from a restricted class of strategies, where the number of calls made to the generative model again has *no dependence* on the POMDP;
- A number of practical algorithms inspired by the general theory we develop here.

The third theme is the connection, both in methods and results, that arises between reinforcement learning and classical supervised learning from independent random examples. Our analyses rely heavily on a combination of Bellman equation calculations, which are standard in reinforcement learning, and uniform convergence arguments, which are standard in supervised learning. This combination of techniques was first applied in [KS99], and results in a similar (and perhaps surprising) conclusion for MDP's, and one that is already familiar in supervised learning. Namely, in arbitrarily complex environments (the (PO)MDP in reinforcement learning, the input distribution and target function in supervised learning), careful application of sparse sampling and restricted model classes can render the difficulty of planning and learning independent of the complexity of this environment.

We defer more detailed description of our results until after the proper definitions have been made. The organization of the paper is as follows: in Section 2, we make the necessary definitions. In Section 3, we motivate and describe our algorithm and results for planning in large or infinite MDP's. In Section 4, we give the necessary definitions for POMDP's, and we give our results for finding a near-best strategy in a POMDP from a restricted class of strategies.

2 Preliminaries

In this section we provide the most basic definitions we will need, ones that are used repeatedly throughout the paper. Other definitions will be given in the sections that require them.

We begin with the definition of a Markov decision process on a set of $N = |S|$ states, explicitly allowing the possibility of the number of states being (countably or uncountably) infinite.

Definition 1 *A* **Markov decision process** M *on a set of* **states** S *and with* **actions** $\{a_1, \ldots, a_k\}$ *consists of:*

- **Transition Probabilities***: For each state-action pair* (s, a)*, a next-state distribution* $P_{sa}(s') = P^M_{sa}(s')$ *that specifies the probability of transition to each state* s' *upon execution of action* a *from state* s*. Note that* $\sum_{s'} P_{sa}(s') = 1$ *(*$\int_S P_{sa}(s')ds' = 1$ *in the case of infinite* S*).*

- **Reward Distributions***: For each state-action pair* (s, a)*, a distribution* R_{sa} *on real-valued* **rewards** *for executing action* a *from state* s*. We assume rewards are bounded in absolute value by* $R_{\max}$*.*

For simplicity, we shall assume in this paper that all rewards are in fact deterministic — that is, the reward distributions have zero variance, and thus the reward received for executing a from s is always exactly R_{sa}. However, all of our results have easy generalizations for the case of stochastic rewards, with an appropriate and necessary dependence on the variance of the reward distributions.

Throughout the paper, we will primarily be interested in MDP's with a a very large (or even infinite) number of states, thus precluding approaches that compute directly on the full next-state distributions. Instead, we will assume that our planning algorithms are given M in the form of the ability to *sample* the behavior of M. Thus, the model given is simulative rather than explicit. We call this ability to sample the behavior of M a *generative model*.

Definition 2 *A* **generative model** *for a Markov decision process M is a randomized algorithm that, on input a state-action pair (s, a), outputs R_{sa} and a state s', where s' is randomly drawn according to the transition probabilities $P_{sa}(\cdot)$.*

In Section 4, we will have reason to consider a weaker type of generative model, but this will suffice for now. We think of a generative model as falling somewhere in between being given explicit next-state distributions, and being given only "irreversible" experience in the MDP (in which the agent follows a single, continuous trajectory, with no ability to reset to any desired state). On the one hand, a generative model may often be available when explicit next-state distributions are not; on the other, a generative model obviates the important issue of exploration that arises in a setting where we only have irreversible experience. In this sense, planning results using generative models blur the distinction between what is typically called "planning" and what is typically called "learning".

Following standard terminology in reinforcement learning, we define a (stochastic) **policy** to be any mapping $\pi : S \mapsto \{a_1, \ldots, a_k\}$. Thus $\pi(s)$ may be a random variable, but depends only on the current state s. We will be primarily concerned with discounted reinforcement learning [1], so we assume we are given a number $0 \leq \gamma < 1$ called the **discount factor**, with

[1]However, most of our results have straightforward generalizations to the undiscounted finite-horizon case for any fixed horizon H, and even to the undiscounted infinite-horizon case, provided proper account is taken of the mixing time of policies [KS98]. Details will be provided in a later version of this paper.

which we may then define the **value function** V^π for any policy π:

$$V^\pi(s) = \mathbf{E}\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_i \middle| s, \pi\right] \tag{1}$$

where r_i is the reward received on the ith step of executing the policy π from state s, and the expectation is over the transition probabilities and any randomization in π. Note that for any s and any π, $V^\pi(s) \leq V_{max}$, where we define $V_{max} = R_{max}/(1-\gamma)$.

We also define the Q-**function** for a given policy π as

$$Q^\pi(s,a) = R_{sa} + \gamma \mathbf{E}_{s' \sim P_{sa}(\cdot)}\left[V^\pi(s')\right] \tag{2}$$

(where the notation $s' \sim P_{sa}(\cdot)$ means that s' is drawn according to the distribution $P_{sa}(\cdot)$). We will later describe an algorithm $\mathcal{A}$ that takes as input any state s and (stochastically) outputs an action a, and which therefore implements a policy. When we have such as algorithm, we will also write $V^{\mathcal{A}}$ and $Q^{\mathcal{A}}$ to denote the value function and Q-function of the policy implemented by $\mathcal{A}$. Finally, we define the optimal value function and the optimal Q-function as $V^*(s) = \sup_\pi V^\pi(s)$ and $Q^*(s,a) = \sup_\pi Q^\pi(s,a)$, and the **optimal policy** π^*, $\pi^*(s) = \arg\max_a Q^*(s,a)$ for all $s \in S$.

3 Planning in Large or Infinite MDP's

Usually one considers the *planning* problem in MDPs to be that of computing a near-optimal policy, given as input the transition probabilities $P_{sa}(\cdot)$ and the rewards R_{sa} (for instance, by solving the MDP for the optimal policy). Thus, the input is a complete and exact model, and the output is a total mapping from states to actions. This is what we might call a "batch" approach to planning — we are given the fully specified environment all at once, and must produce a complete solution (policy) all at once.

Without additional assumptions about the structure of the MDP, such an approach is clearly infeasible in very large state spaces, where even reading all of the input takes N^2 time, and even specifying a general policy requires space on the order of N. In such MDP's, a more fruitful way of thinking about planning might be an *amortized* view, in which we examine the *per-state* complexity of planning. Thus, the input to a planning algorithm would be a single state, and the output would be which single action to take from that state. In this amortized view, a planning algorithm is simply itself a policy (but one that may need to perform some nontrivial computation at each state).

The focus of this first part of the paper is on a planning algorithm whose per-state running time has *no* dependence on the number of states. More precisely, the first of our main results is the description and analysis of an algorithm $\mathcal{A}$ that has access to a generative model for an arbitrary MDP M, takes any state of M as input, produces an action as output, and meets the following performance criteria:

- The policy implemented by $\mathcal{A}$ is near-optimal in M;
- The running time of $\mathcal{A}$ (that is, the time required to compute an action at any state) has *no dependence* on the number of states of M.

This result is obtained under the assumption that the input state to $\mathcal{A}$ requires only $O(1)$ space, a standard assumption that is known as the *uniform cost model* [AHU74], and is typically adopted to allow analysis of algorithms that operate on real numbers (such as we require to allow infinite state spaces). The uniform cost model essentially posits the availability of infinite-precision registers (and constant-size circuitry for performing the basic arithmetic operations on these registers). If one is unhappy with this model, then algorithm $\mathcal{A}$ will suffer a dependence on the number of states only equal to the space required to name the states (at worst $\log(N)$ for N states).

3.1 A Sparse Sampling Planning Algorithm

Here is the main result of this section:

Theorem 1 *There is a randomized algorithm $\mathcal{A}$ that, given access to a generative model for any MDP M, takes as input any state $s \in S$ and any value $\varepsilon > 0$, outputs an action, and satisfies the following two conditions:*

- *(Efficiency) The running time of $\mathcal{A}$ is $O((kC)^H)$, where*

$$H = \log_\gamma(\lambda/V_{\max}),$$

$$C = \frac{{V_{\max}}^2}{\lambda^2}\left(2H\log\frac{H{V_{\max}}^2}{\lambda^2} + \log\frac{1}{\delta}\right),$$

$$\lambda = (\epsilon(1-\gamma)^2)/4,$$

$$\delta = \lambda / R_{\max},$$

$$V_{\max} = R_{\max}/(1-\gamma).$$

In particular, the running time depends only on $R_{\max}$, γ, and ε, and does not *depend on $N = |S|$. If we view $R_{\max}$ as a constant, this bound can be written as*

$$\left(\frac{k}{\varepsilon(1-\gamma)}\right)^{O\left(\frac{1}{1-\gamma}\log\left(\frac{1}{\varepsilon(1-\gamma)}\right)\right)}. \tag{3}$$

- *(Near-Optimality) The value function of the stochastic policy implemented by $\mathcal{A}$ satisfies*

$$|V^{\mathcal{A}}(s) - V^*(s)| \leq \varepsilon \tag{4}$$

simultaneously for all states $s \in S$.

As we have already suggested, it will be helpful to think of algorithm $\mathcal{A}$ in two different ways. On the one hand, $\mathcal{A}$ is an algorithm that takes a state as input and has access to a generative model, and as such we shall be interested in its resource complexity — its running time, and the number of calls it needs to make to the generative model (both per state input). On the other hand, $\mathcal{A}$ produces an action as output in response to each state given as input, and thus implements a (possibly stochastic) mapping of states to actions.

While the formal proof of Theorem 1 is given in Appendix A, and detailed pseudo-code for the algorithm is provided in Figure 1, we now give some high-level intuition for the algorithm and its analysis.

For the sake of simplicity, let us consider here only the two-action case here, with actions a_1 and a_2. Then given as input a state s, the algorithm must use the generative model to find a near-optimal action to perform from state s. Recall that the optimal policy at s is given by $\pi^*(s) = \arg\max_a Q^*(s,a)$, and therefore is completely determined by, and easily calculated from, $Q^*(s,\cdot)$. Estimating the Q-values is a common way of planning in MDP's, and the basic idea of our algorithm is to find good estimates of $Q^*(s,a)$ for all actions a by looking only within a *small* neighborhood of s. In particular, for our algorithm to run in time that does not depend on $N = |S|$, it is critical that the size of this neighborhood does not depend on N, even though, for example, s may have very diffuse transition probabilities, so that it is possible to reach any other state in S from s.

Function: **EstimateQ**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A list $(\hat{Q}_h^*(s, a_1), \hat{Q}_h^*(s, a_2), \ldots, \hat{Q}_h^*(s, a_k))$, of estimates of the $Q^*(s, a_i)$.

1. If $n = 0$, return $(0, \ldots, 0)$.

2. For each $a \in A$, use G to generate C samples from the next-state distribution $P_{sa}(\cdot)$. Let S_a be a set containing these C next-states.

3. For each $a \in A$ and let our estimate of $Q^*(s, a)$ be

$$\hat{Q}_h^*(s, a) = R(s, a) + \gamma \frac{1}{C} \sum_{s' \in S_a} \textbf{EstimateV}(h - 1, C, \gamma, G, s'). \qquad (5)$$

4. Return $(\hat{Q}_h^*(s, a_1), \hat{Q}_h^*(s, a_2), \ldots, \hat{Q}_h^*(s, a_k))$.

Function: **EstimateV**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A number $\hat{V}_h^*(s)$ that is an estimate of $V_h^*(s)$.

1. Let $(\hat{Q}_h^*(s, a_1), \hat{Q}_h^*(s, a_2), \ldots, \hat{Q}_h^*(s, a_k)) :=$ **EstimateQ**(h, C, γ, G, s).

2. Return $\max_{a \in \{a_1, \ldots, a_k\}} \{\hat{Q}_h^*(s, a)\}$.

Function: **Algorithm** $\mathcal{A}(\epsilon, \gamma, R_{max}, G, s_0)$
Input: tolerance ϵ, discount γ, max reward R_{max}, generative model G, state s_0.
Output: An action a.

1. Let the required horizon H and width C parameters be calculated as given as functions of ϵ, γ and R_{max} in Theorem1.

2. Let $(\hat{Q}_H^*(s, a_1), \hat{Q}_H^*(s, a_2), \ldots, \hat{Q}_H^*(s, a_k)) :=$ **EstimateQ**(H, C, γ, G, s_0).

3. Return $\arg\max_{a \in \{a_1, \ldots, a_k\}} \{\hat{Q}_H^*(s, a)\}$.

Figure 1: Algorithm $\mathcal{A}$ planning in large or infinite state spaces. **EstimateV** finds the $\hat{V}_h^*$ described in the text, and **EstimateQ** finds analogously defined $\hat{Q}_h^*$. Algorithm $\mathcal{A}$ implements the policy.

Since $Q^*(s,a) = R_{sa} + \mathbf{E}_{s' \sim P_{sa}(\cdot)}[V^*(s')]$, if we take a sufficiently large (but independent of N) random sample of next states s' from $P_{sa}(\cdot)$, and somehow manage to determine very good approximations to the $V^*(s')$, we can obtain a good approximation to $Q^*(s,a)$. In other words, the standard duality between Q-functions and value functions allows us to reduce the problem of sparsely sampling the generative model to estimate $Q^*(s,a)$ to the problem of sparsely sampling the generative model to estimate $V^*(s)$. (This point is made formal by the proof in Appendix A.) For expository reasons, we will actually describe how we estimate $V^*(s)$.

There are two parts to the approximation we will use. First, rather than estimating V^*, we will actually estimate, for a value of H to be specified later, the H-step expected discounted reward

$$V_H^*(s) = \mathbf{E}\left[\sum_{i=1}^{H} \gamma^{i-1} r_i \,\middle|\, s, \pi^* \right] \tag{6}$$

where r_i is the reward received on the ith time step upon executing the optimal policy π^* from s. Note the "0-step" expected discounted reward is easy to estimate: Since $V_0^*(s) = 0$, we may simply pick our 0-step estimates to be $\hat{V}_0^*(s) = 0$. Moreover, we see that the $V_h^*(s)$, for $h \geq 1$, are recursively given by

$$\begin{aligned} V_h^*(s) &= R_{sa^*} + \gamma \mathbf{E}_{s' \sim P_{sa^*}(\cdot)}[V_{h-1}^*(s')] & (7) \\ &\approx \max_a \{ R_{sa} + \gamma \mathbf{E}_{s' \sim P_{sa}(\cdot)}[V_{h-1}^*(s')] \} & (8) \end{aligned}$$

where a^* is the action taken by the optimal policy from state s. The quality of the approximation in Equation (8) becomes better for larger values of h, and is controllably tight for the largest value $h = H$ we eventually choose. One of the main efforts in the formal proof in Appendix A is establishing that the error incurred by the recursive application of this approximation can be made controllably small by choosing H sufficiently large.

Thus, if we are able to obtain an estimate $\hat{V}_{h-1}^*(s)$ of $V_{h-1}^*(s)$ for any s, we can inductively define an algorithm for finding an estimate $\hat{V}_h^*(s)$ of $V_h^*(s)$ by making use of Equation (8). The algorithm will approximate the next-state distribution by a sample of C random next states from the generative model, where C is a parameter to be determined (and which, for reasons that will become clear later, we call the "width"). Recursively, given a way of finding the estimator $\hat{V}_{h-1}^*(s)$ for any s, we may then find our estimate $\hat{V}_h^*(s)$ of $V_h^*(s)$ as follows:

1. For each action a, use the generative model to get R_{sa} and to sample a set S_a of C independently sampled states from $P_{sa}(\cdot)$.

2. Use our procedure for finding $\hat{V}^*_{h-1}$ to estimate $\hat{V}^*_{h-1}(s')$ for each state s' in any of the sets S_a.

3. Following Equation (8), our estimate of $V^*_h(s)$ is given by

$$\hat{V}^*_h(s) = \max_a \left\{ R_{sa} + \gamma \frac{1}{C} \sum_{s' \in S_a} \hat{V}^*_{h-1}(s) \right\} \tag{9}$$

We have described our algorithm "bottom up," and it is also informative to view it "top down." Figure 2 shows a conceptual picture of a run of the algorithm from an input state s_0, for $C = 3$. (C will typically be much larger.) From the root s_0, we try action a_1 three times and action a_2 three times. From each of the resulting states, we also try each action C times, and so on down to a depth H in the tree. Then, zero values that we assign to the leaves correspond to our estimates of $\hat{V}^*_0$, which are then "backed-up" to find estimates of $\hat{V}^*_2$ for their parents, which are in turn backed-up to their parents, and so on, up to the root to find an estimate of $\hat{V}^*_H(s_0)$. (Though we describe the algorithm here as building the entire tree in memory, a better implementation, such as that given in Figure 1, will have much more modest memory requirements.)

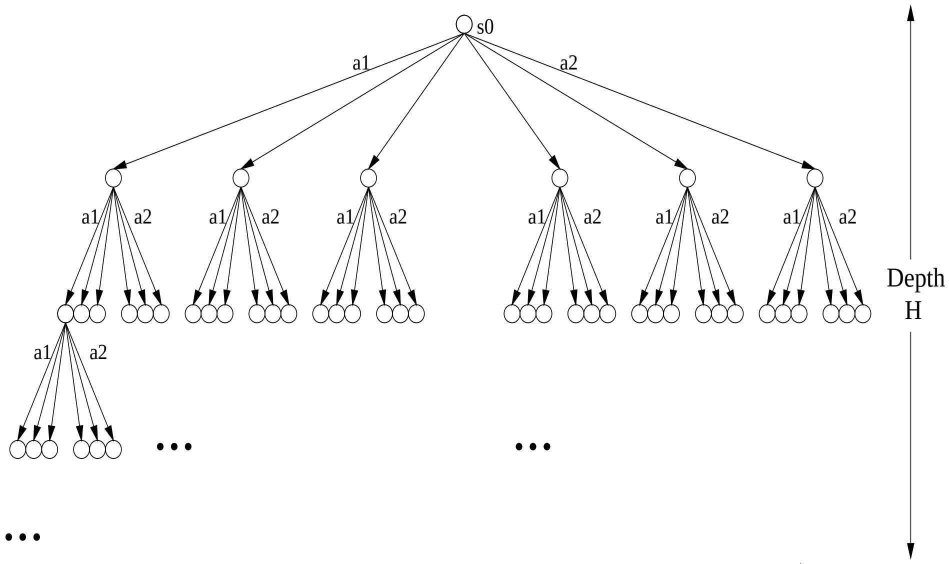

Figure 2: States looked at by unconstrained planning algorithm. (Shown with $C = 3$, actions a_1, a_2.)

To complete the description of the algorithm, all that remains is to choose H, the depth, and C, which controls the width of the tree. Bounding the required depth H is the easy and standard part. It is not hard to see that if we choose depth $H = \log_\gamma \epsilon(1-\gamma)/R_{max}$ (the so-called *ϵ-horizon time*), then the discounted sum of the rewards that is obtained by considering rewards beyond this horizon is bounded by ϵ. In other words, by taking the "infinite tree" and truncating it at depth H, the H-step expected discounted rewards are still good enough estimates of the value function for us to compute a near-optimal action from s_0.

However, such a tree may still be as large as M itself. For instance, if the next-state distribution from s is uniform or nearly uniform over all the states in M, then it would seem that, in order to approximate the next-state distribution well, we would need to take at least order $N = |S|$ samples, if only to make sure we see most of possible next-states at least once in our sample. Such an analysis would give $C = N$ in the worst case, for an overall bound of $N^H \gg N$ — in which case we would be better off simply running on full probability tables for M, a task we are implicitly assuming is infeasible due to large or infinite N.

The central claim we establish about C is that it can be chosen *independent* of the number of states in M, yet still result in choosing near-optimal actions at the root. The key to the argument is that even though the size

C sample may be a very poor approximation to the next-state distribution from s, it will, nevertheless, give a good estimate of the *expectation* term in Equation (8), and that is really all we need. For this we apply a careful combination of uniform convergence methods and inductive arguments on the tree depth. Again, the technical details of the proof of Theorem 1 are provided in Appendix A.

3.2 Some Practical Issues and Lower bounds

Even though the running time of algorithm $\mathcal{A}$ does not depend on the size of the MDP, it still runs in time exponential in the ϵ-horizon time H, and therefore exponential in $1/(1-\gamma)$. It would seem that the algorithm would be practical only if γ is not too close to 1. We will shortly describe lower bounds showing it is not possible to do much better without further assumptions about the MDP. Nevertheless, there are a couple of simple tricks that may help to reduce the running time in certain cases.

The first observation is that whenever there are two nodes at the same level of the tree in Figure 2 that are labeled by the same state, then they may be collapsed into just one node (keeping just one of their subtrees). As a concrete example, suppose that, during a run of the algorithm, we tried action a_1 from the root s_0 and got back some state s' from the generative model, and later tried a_2 also from s_0 and got back the same s'. Our current description of the algorithm would call **EstimateV** (described in Figure 1) twice, and find two independent estimates of $V^*_{H-1}(s')$ — one for the a_1 subtree, and one for the a_2 subtree. A more efficient algorithm that still gives the same theoretical guarantees would notice that these two nodes at the same level correspond to the same state s', and call **EstimateV** only once to find just a single estimate $\hat{V}^*_{H-1}(s')$, and then use the same value in both places. This example describes "collapsing" two children of the same root node, but it is valid to apply it to any two nodes at the same level of the tree labeled by the same state, and can easily be implemented by having **EstimateV** memoize its resulting $\hat{V}^*_h(s)$ as functions of h and s (though memory requirements may then have to be further addressed). Indeed, this optimization is particularly nice when the domain is actually deterministic: if each action deterministically causes a transition to a fixed next-state, then the tree would grow only as 2^H for 2 actions (or k^H for k actions). A similar approach to this particular case of "search" in deterministic domains with infinite state-spaces was explored experimentally in Davies, Ng and Moore [DNM98], and encouraging results reported. If the domain is "nearly deterministic," then we have behavior somewhere in between. Similarly, if there are only some $N_0 \ll |S|$ states reachable from s_0, then the tree would also never grow wider than N_0, giving running time $O(N_0^H)$.

In implementing the algorithm, one may also not wish to specify ϵ in advance, but rather just try to do as well as is possible with the computational resources available, in which case an "iterative-deepening" approach may be taken. In our case, this would entail simultaneously increasing C and H by decreasing the target ϵ. Also, as studied in Davies et. al. [DNM98], if we have access to an initial estimate of the value function, we can replace our estimates $\hat{V}_0^*(s) = 0$ at the leaves with the estimated value function at those states. Though we shall not do so here, it is again easy to make formal performance guarantees depending on C, H and the supremum error of the value function estimate we are using.

Unfortunately, despite these tricks, it is not difficult to prove a lower bound that shows that any planning algorithm with access only to a generative model, and which implements a policy that is ϵ-close to optimal in a general MDP, must have running time at least exponential in the ϵ-horizon time.

4 Choosing a Constrained Strategy in a POMDP

In the second part of the paper, we address the following question: given a restricted class Π of policies, how many calls to a generative model are needed before we have enough data to choose a $\pi \in \Pi$ that is close in performance to the best strategy in Π? Our results on this question will share the theme of sparse sampling with the results of the last section.

First we must motivate the consideration of restricted policy classes. After all, we have just established the fact that near-optimal planning from a generative model can be accomplished in an arbitrarily large Markov decision process by an algorithm (stochastic policy) whose computational complexity is independent of the number of states; what more could we ask for? There are at least two settings in which the approach of Section 3 is inapplicable, and we are instead led to restrict our ambitions to choosing a good policy from a limited class:

- **Partial Observability.** The results of Section 3 rely heavily on full observability — that is, we assume our algorithm (stochastic policy) will receive as input the fully specified state of the underlying MDP at each step. A more realistic setting is that of *partial observability*.

- **Faster Amortized Policies.** The algorithm/policy of Section 3 demands that significant computation be performed at each new state reached (even if this computation does not depend on the size of the state space). We would prefer, of course, to find a near-optimal policy whose per-state computation time was constant. While we cannot store policies as state-action look-up tables in very large state spaces, we might hope to choose a good, fast policy from a restricted class.

Actually, since we are now starting to address partially observable environments, it will be more accurate to refer to classes of *strategies*. A strategy is any mapping from entire histories of experience — transcripts of observations, rewards, and actions — to the next action.

The idea of imposing limitations on our hypotheses in order to handle complex environments is ubiquitous in machine learning and statistics — for instance, neural network and decision tree representations are often employed with no presumption that they can perfectly model the data source. The notion of resorting to constrained classes of strategies has been implicit in the reinforcement literature as well — for instance, in the use of function approximation, and the use of memoryless strategies in partially observable settings (where the full "belief state" approach is optimal but intractable).

The question of how many calls to a generative model are required to choose a near-best strategy in a given class is analogous to the classical question of *sample complexity* in supervised learning — but harder. The added difficulty lies in the *reuse* of data. In the supervised learning setting, *every* random example $\langle x, f(x)\rangle$ provides feedback about *every* hypothesis function $h(x)$. If $h(x)$ is restricted to lie in some "hypothesis class" $\mathcal{H}$, this fact permits bounds on the number of random examples required that are far smaller than the number of functions in $\mathcal{H}$. For instance, if $\mathcal{H}$ contains only a finite number L of functions, $O(\log(L))$ bounds (ignoring parameters of the problem other than L for now) are obtained on the sample size required to choose a near-best approximation to $f(x)$ lying in $\mathcal{H}$.

In the MDP or POMDP setting, however, we must decide *how* to use the generative model — that is, which states and actions to feed to the generative model — in order to recover this same desirable reuse of experience across multiple strategies in our class. To see the issue more clearly, consider the "straw man" algorithm that, for every strategy $\pi \in \Pi$, uses the generative model to generate many Monte Carlo trials of π starting at s_0, and thus form an accurate empirical estimate of $V^{\pi}(s_0)$. It is not clear that these trajectories under π are of much use in evaluating a different $\pi' \in \Pi$, as

π and π' may quickly disagree on which actions to take. Thus, the naive method of generating Monte Carlo trials would result in $O(L)$ bounds on the number of calls to the generative model, rather than $O(\log(L))$, for the simple case of $|\Pi| = L$.

In contrast, in the ensuing sections, we shall present two different ways of generating "reusable" trajectories. In both cases, we generate a (relatively) small number of trajectories — a number that is independent of the state-space size of the POMDP, and depends only linearly on a general measure of the complexity of the strategy class that we shall define. We prove that these generated trajectories are enough to give us accurate estimates of the expected return of any strategy *in our class*. In both cases, the trajectories may be correlated, and we apply uniform convergence techniques to limit the adverse effects of these correlations. Our measure of strategy class complexity is inspired by and generalizes the notion of VC dimension in supervised learning, and we will eventually give bounds that recover for our setting the most powerful analogous results in supervised learning — bounds for arbitrary, infinite strategy classes that depend only on the dimension of the class.

Although our development concentrates on restricted classes of strategies, this can be translated to many other types of restriction. For example, if we restrict our value function estimates to be of a certain parametric form, this implies a restriction on policies, namely to those policies that can be achieved by behaving greedily with respect to the parametric value functions. Similarly, if we estimate models of a POMDP that lie in some restricted class (for instance, POMDP's with factored next-state distributions), this implies a restriction to strategies that are optimal with respect to such POMDP's. In this way, our results for restricted strategy classes can be applied to any other hypothesized constraint on a POMDP planning problem.

4.1 Preliminaries for POMDP's and Strategy Classes

Definition 3 *A* **partially observable** *Markov decision process (POMDP) consists of an* **underlying** *Markov decision process (transition probabilities $P_{sa}(\cdot)$ and rewards R_{sa}), along with* **observation distributions** $Q_s(o)$ *for each Here o is a random variable called the* **observation** *made at state s.*

When considering POMDP's, the word "state" is always exclusively reserved for a state of the underlying MDP, and is never to be confused with the random observation made at that state.

An agent wandering in a POMDP takes actions and receives rewards, as in an MDP, but the agent never directly sees the identity of the current state. Rather, the agent has access only to the current observation. This step towards realism greatly complicates the problems of both planning and learning. Intuitively, the agent may never know the true state, but must attempt to track the current "belief state" — that is, the likelihood that it is in each of the possible states of the underlying MDP (of which there may be a large or even infinite number). In general, this belief state may be arbitrarily complex, depending strongly on both the initial state of the underlying MDP (or an initial distribution of states), as well as on the entire history of actions and observations. This is a sharp contrast to the fully observable case, where the optimal policy depends only on the current state.

Definition 4 *A* **generative model** *for a partially observable Markov decision process M is randomized algorithm that, given as input a state-action pair (s, a), outputs a state s' that is distributed according to the next-state distribution $P_{sa}(\cdot)$, an observation o that is distributed according to the distribution $Q_s(\cdot)$, and the reward R_{sa}.*

Thus, a generative model for a POMDP simply consists of a generative model for the underlying MDP, along with a random observation at each state. At first blush, this definition may seem unreasonably generous — we are essentially assuming that we are provided with a *fully observable* simulation of a *partially observable* process. However, the key point is that planning algorithms provided with this generative model must still find a strategy that performs well in the partially observable setting. For instance, although we could simply use the generative model to find a near-optimal policy (state-to-action mapping) for the underlying MDP, this policy will be useless in the POMDP, where the state is unknown. We can really only use the generative model to find a strategy that maps from (histories of) observables to actions. As a concrete example, in designing an elevator control system [CB96], we may have access to a simulator that generates random rider arrival times, and keeps track of the waiting time of each rider, the number of riders waiting at every floor at every time of day, and so on. However helpful this information might be in *designing* the controller, this controller must only *use* information about which floors currently have had their call button pushed (the observables). In any case, readers uncomfortable with the power provided by our POMDP generative models are referred to the results of Section 4.3, where they are replaced by an extremely weak form of simulation — namely, a subroutine for generating only the observable history along truly random trajectories.

We now move on to define strategy classes in a POMDP. In general, an agent will, at time H, have seen some sequence of observables $o_0, \ldots, o_H$, and will have chosen actions and seen reinforcements for each of the H time steps prior to the current one. Thus, we may write its *observable history* as a list of tuples of observations, actions and rewards

$$\langle (o_0, a_0, r_0), \ldots, (o_{H-1}, a_{H-1}, r_{H-1}), (o_H, \_, \_) \rangle \tag{10}$$

where the last entry indicates that observable histories always conclude with the observation made at the final state reached. Such observable histories are inputs to strategies.

Definition 5 *A* **strategy** π *in a POMDP is any (stochastic) mapping from finite* **observable histories** $\langle (o_0, a_0, r_0), \ldots, (o_{H-1}, a_{H-1}, r_{H-1}), (o_H, \_, \_) \rangle$ *(of observations, actions, and rewards) to actions. A* **strategy class** Π *is any class of strategies.*

Of course, this definition includes the special case in which the MDP is fully observable (the observations are the states), and Π is simply a class of policies. As we have already remarked, we can think of Π as representing a *constraint* or *bias* on strategies adopted with the hope of avoiding the intractability of full belief-state planning, while still permitting good, if suboptimal, performance.

An important point to make is that once we limit the class of strategies we will entertain, there may not be a single "best" strategy in the class unless we explicitly induce a metric of some kind — one strategy might be better from certain states, and another strategy better from other states, and so pairs of strategy may now be incomparable. To get around this, we adopt the common assumption of a fixed start state s_0 in the underlying MDP. (This is also essentially equivalent to assuming a fixed distribution D over start states, since s_0 can be a "dummy" state whose next-state distribution under any action is D.) This permits the following definition.

Definition 6 *Let M be a POMDP with start state s_0, and let Π be a class of strategies. Then*

$$opt(M, \Pi) = \sup_{\pi \in \Pi} V^{\pi}(s_0) \tag{11}$$

where $V^{\pi}(s_0)$ is the expected discounted return of π from s_0.

Dependence of $opt(M, \Pi)$ on s_0 is suppressed for notational brevity. Later and purely for pedagogical convenience, we also assume a fixed deterministic observation o_0, not necessarily unique to s_0, is observed whenever we are in the start state s_0. (Without this assumption, we would need to modify the generative model slightly to be able to sample from $Q_{s_0}(\cdot)$; then all the ensuing algorithms continue to work with only trivial modification.)

4.2 The Trajectory Tree Method

As we have already mentioned, we will present two methods for generating trajectories from a generative model for a POMDP — in fact, the second of these two methods will not even require the full power of a generative model. We shall refer to the two methods as the *trajectory tree* method and the *random trajectory* method. For both methods, we will eventually prove that a number of calls to the generative model depending only on the class Π suffices to give uniformly good estimates of $V^\pi(s_0)$ for every $\pi \in \Pi$. While the bounds we can prove for the two methods are similar, they have some conceptual and algorithmic differences that make it worthwhile to consider both. We begin with the trajectory tree method.

We are given a generative model for a POMDP M with distinguished start state s_0. We now describe the generation of a single *trajectory tree*. For ease of exposition, we assume there are only two actions in M, action a_1 and action a_2, but the result easily generalizes to any finite number of actions.

A trajectory tree is simply a binary tree in which each node is labeled by both a state in M and an observation, and has a single child for action a_1 and a single child for action a_2. Additionally, each link from a parent to a child will have a reward labeling that link. The depth of the tree will always be H, the ϵ-horizon time, so the total size (number of nodes) in each trajectory tree will be 2^H. Thus, a trajectory tree is quite similar to the tree built by the planning algorithm $\mathcal{A}$ of Section 3, except that now we are in a POMDP and the nodes contain both observations and states, and we are fixing the number of children at exactly the number of actions.

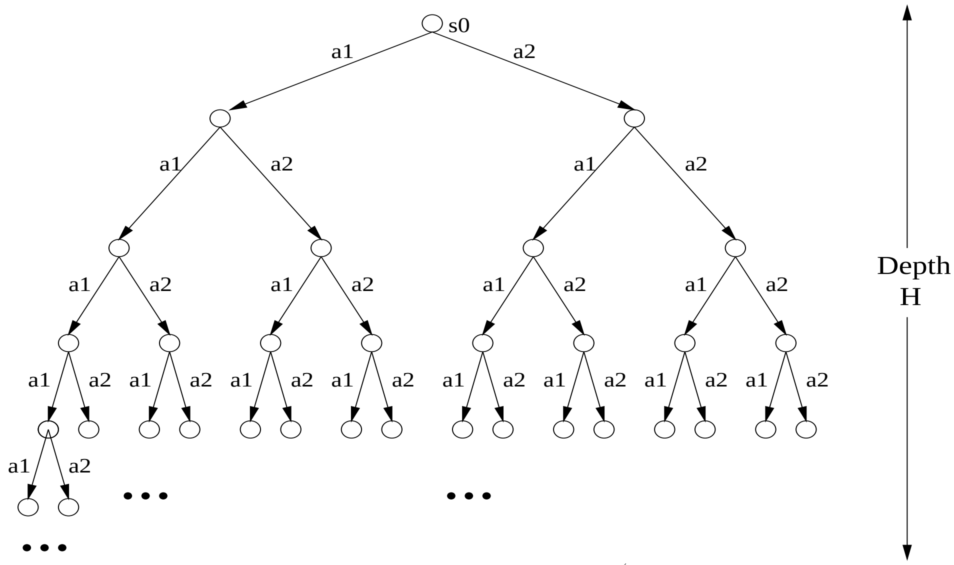

Figure 3: The structure of a typical trajectory tree. (Shown here with actions a_1 and a_2, and with observation, state, and reinforcement labels omitted.)

Also as in Section 3, a trajectory tree is built in a straightforward manner from the generative model. The root will always be labeled by the start state s_0 and the observation o_0. To generate the two children of the root, we call the generative model on (s_0, a_1) and (s_0, a_2), and the generative model returns the two next states reached (say s'_1 and s'_2, respectively), the two observations made (say o'_1 and o'_2, respectively), and the two rewards received ($r'_1 = R_{s_0 a_1}$ and $r'_2 = R_{s_0 a_2}$). Then (s'_1, o'_1) and (s'_2, o'_2) will label the a_1-child and a_2-child of the root, and the links from the root to these children will have rewards r'_1 and r'_2. Recursively, for any node s of depth less than H, we generate two children and rewards in the same way with the generative model.

Now for any *deterministic* strategy π, and for any trajectory tree T, π accumulates a well-defined return on T. Strategy π defines a path through the tree T — we start π at the root, where it sees whatever observation is stored there (that is, the observable history so far consists only of this observation). Strategy π then decides to either take action a_1 or action a_2, which selects a child of the root. Inductively, if π has reached some internal node in T, we can feed to π the entire observable history generated along the path to this node, and discover which child of the current node π selects. In this way, we "run" π on T to reach some leaf node of T, and we define the return $R(\pi, T)$ to be the discounted sum of rewards along the path taken

Function: **MakeTrajectoryTree**(h, G, s, o)
Input: depth h, width C, generative model G, state s, observation o.
Output: A trajectory tree T rooted at state s (with assumed observation o as s).

1. Set $T :=$ MAKE-NODE(). T.STATE $:= s$, T.OBSERV $:= o$.

2. If $h = 0$, do T.LEAF? := YES, return T else T.LEAF? := NO, end.

3. For $i = 1$ to k,
   3.1 Use G on (s, a_i) to get next-state s'_i, next-observ o'_i and reward r_i.
   3.2 T.REINF$(i) := r_i$.
   3.3 T.CHILD$(i) :=$ **MakeTrajectoryTree**$(h - 1, G, s'_i, o'_i)$

4. Return T.

Function: **EvaluateStrategy**(π, T)
Input: Policy π, trajectory tree T.
Output: $R(\pi, T)$

1. Set $n := T$, $j := 0$, $r := 0$, $t :=< n.\text{OBSERV}, \_, \_ >$.

2. While (n.LEAF? = NO)
   2.1 Set $a := \pi(t)$. Let i be such that $a = a_i$ (the i-the action).
   2.2 Set $r := r + \gamma^j n.\text{REINF}(i)$, $j := j + 1$
   2.3 Change the last element of t to $< n.\text{OBSERV}, a, n.\text{REINF}(i) >$, and then append $< n.\text{CHILD}(i).\text{OBSERV}, \_, \_ >$ to the end of t.
   2.4 Set $n = n$.CHILD(i).

3. Return r.

Function: **PickStrategy**(m, Π, s_0, o_0, G)
Input: Number m of trees, strategy class Π, start state s_0, observation o_0 for s_0, generative model G
Output: A strategy $\pi \in \Pi$.
Remark: The arg max below can be implemented as an enumeration only if $|\Pi|$ is finite (and preferably "small"); otherwise, some approximation to it via search would be needed.

1. Generate m independent trajectory trees $T_1, \ldots, T_m$ with m calls to **MakeTrajectoryTree**(h, G, s_0, o_0)

2. Return $\arg\max_{\pi \in \Pi} \frac{1}{m} \sum_{i=1}^{m}$ **EvaluateStrategy**(π, T_i)

Figure 4: Algorithm for choosing a constrained strategy in a POMDP. **MakeTrajectoryTree** makes trajectory trees, **Evaluate Strategy** calculates $R(\pi, T)$, and **PickStrategy** picks the strategy (this version works correctly only when strategies are deterministic or $|\Pi|$ is finite; see text).

by π. In the general case that π is stochastic, π defines a *distribution* on paths in T, and $R(\pi, T)$ becomes the expected discounted sum of rewards according to this distribution. If we have created trajectory trees $T_1, \ldots, T_m$, a natural estimate for $V^\pi(s_0)$ is then

$$\hat{V}^\pi(s_0) = (1/m) \sum_{i=1}^{m} R(\pi, T_i). \tag{12}$$

The main goal of this section is to establish a nontrivial relationship between the quality of this estimate and the "sample size" m.

4.2.1 Trajectory Tree Analysis: Finite Strategy Classes

We are now in position to state our first result of the variety we seek in this section: a bound on the number of trajectory trees required to choose a near-best strategy in Π. As is typical of analogous results in supervised learning, we will actually prove an even stronger *uniform convergence* result. We begin with the case where Π is a finite class of L strategies.

Theorem 2 *Let Π be any class of L (stochastic) strategies in an arbitrary POMDP M, let $T_1, \ldots, T_m$ be m trajectory trees created using a generative model for M as described above, and let $\hat{V}^\pi(s_0)$ be the estimate for $V^\pi(s_0)$ derived from these trajectory trees. If*

$$m = O((V_{\max}/\epsilon)^2 \log(L/\delta)) \tag{13}$$

then with probability at least $1 - \delta$,

$$\left| V^\pi(s_0) - \hat{V}^\pi(s_0) \right| \leq \epsilon \tag{14}$$

holds simultaneously for all $\pi \in \Pi$. The total number of calls made to the generative model will thus be at most $2^H m = O(2^H (V_{\max}/\epsilon)^2 \log(L/\delta))$.

Proof:(Sketch) Let us fix a (stochastic) strategy $\pi \in \Pi$. Then each trajectory tree is used to generate a run (or distribution on runs) of the strategy π, and our estimate $\hat{V}^\pi(s_0)$ for the return of strategy π is the average return of its m runs (distributions on runs). The crucial observation is that for this fixed π, the values $R(\pi, T_i)$ that are generated by the different trajectory trees T_i are independent. This is easily seen if we imagine that each trajectory tree is constructed by first constructing the path (or distribution on paths) determined by π, and then afterwards constructing the rest of the tree. The resulting distribution on trees is identical to the distribution generated by our original description.

This independence implies that we can apply the Chernoff bound for the deviation of the estimate from the value of $V^{\pi}(s_0)$. Since the maximum return is bounded by V_{max}, we have that the probability that the deviation is more than $\epsilon/2$ is bounded by $e^{-\epsilon^2 m/(4{V_{max}}^2)}$.

So far we have restricted our attention to a fixed policy π. By appealing to the so-called *union bound*, we have that the probability that *some* $\pi \in \Pi$ deviates by more than $\epsilon/2$ is bounded by $ne^{-\epsilon^2 m/(4{V_{max}}^2)} = \delta$. Note that if all of the estimates are within $\epsilon/2$ from the true value, then the policy with the highest estimate can be at most ϵ from the optimal value. $\square$

The crucial point to note about this result is the dependence on L: it is only *logarithmic* in L, as opposed to the linear bound expected for the straw-man Monte Carlo approach described earlier. Thus, the trajectory tree approach is achieving considerable *reuse* of the generated experience: with only on the order of $\log(L)$ data, we can get an excellent estimate of the value of all L strategies. Moreover, we have the following corollary:

Corollary 3 *Under the conditions of Theorem 2, let*

$$\hat{\pi} = \arg\max_{\pi\in\Pi}\{\hat{V}^{\pi}(s_0)\} \tag{15}$$

the policy with the highest empirical return on the trajectory trees. Then with probability at least $1-\delta$, $\hat{\pi}$ *is near-optimal within* Π*:*

$$V^{\hat{\pi}}(s_0) \geq opt(M,\Pi) - 2\epsilon. \tag{16}$$

Proof: By the previous theorem, with probability $1-\delta$, $|V^{\pi}(s_0)-\hat{V}^{\pi}(s_0)| \leq \epsilon$ for all $\pi \in \Pi$, in which case

$$V^{\hat{\pi}}(s_0) \geq \hat{V}^{\pi}(s_0) - \epsilon \geq \hat{V}^{\pi^*}(s_0) - \epsilon \geq opt(M,\Pi) - 2\epsilon. \tag{17}$$

$\square$

4.2.2 Trajectory Tree Analysis: Infinite Strategy Classes

For the case of infinite strategy classes, we must define a notion $dim(\Pi)$ of the *dimension* of the strategy class Π. For readers familiar with the classical literature on uniform convergence in supervised learning, this definition involves giving a notion of Π *shattering* a set of trajectory trees. Informally, for now it suffices to say that $dim(\Pi)$ is always $O(\log(|\Pi|))$, but in the more interesting case of an infinite parametric class, $dim(\Pi)$ will typically be polynomially related to the number of parameters. Once this machinery is put in place, we can prove a result generalizing Theorem 2 for infinite classes, with $dim(\Pi)$ playing the role of $\log(|\Pi|)$: namely, for any strategy class Π, if we take $m = O(dim(\Pi))$ (ignoring dependences on all other parameters) trajectory trees, then for *every* $\pi \in \Pi$ these trajectory trees provide an ϵ-good estimate of $V^{\pi}(s_0)$ (with high probability).

However, in Section 4.3 we will introduce our second method of generating reusable trajectories, and similar machinery must be developed to handle the case of infinite strategy classes for this method. Thus, in the interests of brevity, we will not give the formal treatment of the infinite case for the trajectory tree method, but only state without proof a simple version of the general theorem, for infinite, two-action, deterministic strategy classes, which can be done making use only of the familiar VC dimension of binary functions.

Suppose Π is an infinite class of deterministic strategies in a two-action POMDP. Then each strategy $\pi \in \Pi$ is simply a deterministic function mapping from the set of all observable histories to the set $\{a_1, a_2\}$, and thus is a binary function. We can thus write $dim(\Pi)$ to denote the Vapnik-Chervonenkis dimension of the set of binary functions Π. We then have the following uniform convergence theorem and corollary.

Theorem 4 *Let Π be any class of deterministic strategies for an arbitrary two-action POMDP, and let $dim(\Pi)$ be the VC dimension of Π. Let $T_1, \ldots, T_m$ be m trajectory trees created using a generative model for M. If*

$$m = O\left((R_{\max}/(1-\gamma)\epsilon)^2 (H\, dim(\Pi) + \log(1/\delta))\right) \tag{18}$$

then with probability at least $1 - \delta$,

$$\left| V^{\pi}(s_0) - \hat{V}^{\pi}(s_0) \right| \leq \epsilon \tag{19}$$

simultaneously for all $\pi \in \Pi$.

Corollary 5 *Under the conditions of Theorem 4, it holds that with probability at least $1 - \delta$ that the strategy $\hat{\pi} \in \Pi$ that maximizes the empirical trajectory tree return $\hat{V}^{\pi}(s_0)$ will be near-optimal within Π:*

$$V^{\hat{\pi}}(s_0) \geq opt(M, \Pi) - 2\epsilon. \tag{20}$$

4.2.3 Trajectory Tree Algorithmic Issues

Corollaries 3 and 5 establish that the empirical-return maximizing strategy is, with high probability, a near-optimal strategy within Π. Algorithmically, this opens up a number of directions for learning with deterministic or stochastic strategy classes in POMDPs. This section outlines a number of such algorithmic ideas; a practical algorithm for learning stochastic strategies in also given in detail in Appendix B.

In analogy to supervised learning algorithms that pick a hypothesis from a restricted class so as to minimize empirical training error, our uniform convergence results suggest an algorithm that picks a strategy so as to maximize the empirical return on our "training set" consisting of m trajectory trees. This process is outlined in Figure 4 (with the function for calculating $R(\pi, T)$ being the version that gives theoretical guarantees only for deterministic strategies or when $|\Pi|$ is finite). If we think of $\Pi = \{\pi_\theta | \theta \in \mathbb{R}^d\}$ as being smoothly parameterized in some set of parameters θ, then a useful picture is to think of ourselves as searching over the surface $\hat{V}^{\pi_\theta}(s_0) = (1/m) \sum R(\pi_\theta, T_i)$ parameterized by θ, which by our theoretical result is uniformly close to the true expected return $V^{\pi_\theta}(s_0)$.

For deterministic strategy classes, simple algorithms to do this naturally come to mind. For example, implementing the $\arg\max$ operation (over Π, to pick the empirical-return maximizing strategy) is easy if $|\Pi|$ is small, in which case a simple enumeration will suffice. If, on the other hand, we have a class Π that is too large to enumerate (which will be the more typical case), or perhaps even infinite, but $dim(\Pi)$ is bounded, then a greedy local search algorithm can be used as an approximation to an explicit maximization. Here the search is over strategies to (locally) maximize $\hat{V}^{\pi}(s_0) = (1/m) \sum R(\pi, T_i)$.

Regarding the 2^H cost per trajectory tree, it may also be possible in practice to significantly reduce this cost, by use of a simple observation. In particular, suppose we perform a greedy local search over a class Π of deterministic strategies. Then at any time in the search, to evaluate the policy we are currently considering, we really need to look at only one length-H path through each tree, corresponding to the path taken by the strategy our search is currently considering. Thus, it seems a sensible strategy would be to build the trajectory trees *lazily* — that is, to incrementally build each node of each tree only as it is needed to evaluate $R(\pi, T_i)$ for the current strategy π. If there are parts of a tree that are reached only by poor policies, then a good search algorithm may never even build these parts of the tree.

Avoiding the exponential dependence on H via lazy trajectory tree construction would appear to apply only to the case of deterministic strategies, since stochastic strategies define a distribution over all the paths in a trajectory tree, and so in general, evaluation requires examing the entire tree. However, if our strategy class Π is a smoothly parameterized set, then there turns out also to be a very practical implementation of stochastic gradient ascent that again has only linear dependence on H. This algorithm described in detail in Appendix B, and we sketch it only briefly here.

Suppose $\Pi = \{\pi_\theta : \theta \in \mathbb{R}^d\}$ is smoothly parameterized family of *stochastic* policies in parameters θ. Then as we suggested earlier, $\hat{V}^{\pi_\theta}(s_0) = (1/m)\sum R(\pi_\theta, T_i)$ is itself a smooth function of θ, and we may think about using gradient ascent to maximize it as a function of θ. The straightforward way of finding the gradient of $\hat{V}^{\pi_\theta}(s_0)$ would still seem to require examining the entire tree; but it turns out that, by further *subsampling* the trajectory trees, we can efficiently (in $O(H)$ time) find an unbiased estimate of the gradient $\frac{d}{d\theta}\hat{V}^{\pi_\theta}(s_0)$. This is all we need in order to do stochastic gradient ascent efficiently.

Having come this far, there is also a simple further modification to the subsampling algorithm that leads us close to to a line of research pursued by Kimura, Yamamura and Kobayashi [KYK95]. Rather than doing stochastic gradient ascent on $\hat{V}^{\pi_\theta}(s_0)$, it is possible to perform stochastic gradient ascent directly on $V^{\pi_\theta}(s_0)$. We can give an algorithm that, given a current setting of the parameters θ_0 as input, enjoys the following properties:

- (Efficiency) Has expected running time $O(1/(1-\gamma))$;
- (Unbiasedness) Outputs an unbiased estimate of

$$\left.\frac{d}{d\theta}\right|_{\theta=\theta_0} V^{\pi_\theta}(s_0); \tag{21}$$

- (Bounded variance) The estimate has bounded variance (for fixed, k, γ, R_{max} and given a bound on the gradient of the parameterized family itself, $|\frac{d}{d\theta}\mathbf{Pr}[\pi_\theta(s) = a]|$).

It should be clear that the three conditions above are exactly what we need to do stochastic gradient ascent directly on $V^{\pi_\theta}(s_0)$. Unfortunately, this version of the algorithm requires that we run the generative model on each gradient ascent step (that is, it is not sufficient to sample just a fixed number of trajectory trees, and then make no further use of the generative model).

Moreover, as gradient ascent is a local algorithm, it also seems difficult to prove bounds on performance other than convergence to a local optimum in θ. We elaborate considerably on these algorithmic ideas and issues, which incidentally include applications to learning deterministic strategies, in Appendix B.

4.3 The Random Trajectory Method

We now move on to present the second of our two methods for generating only a small number of trajectories sufficient to evaluate all the strategies in a large class. We call this second approach the *random trajectory* method, and one advantage it enjoys over the trajectory tree method is that it does not need the full power of a generative model for the POMDP. In fact, the random trajectory method requires only the observable histories generated by truly random trajectories from the start state. Resets to states other than the start state are unnecessary, as is the ability to see the underlying states along the trajectory. We begin with a definition capturing this weaker form of simulation.

Definition 7 *A* **random trajectory generator** *for a partially observable Markov decision process M, with designated start state s_0, generates an observable history of a given length H starting from s_0 by following the truly random policy (at each state each action is equally likely). Thus, the generator outputs only the observable history*

$$\langle (o_0, a_0, r_0), \ldots, (o_{H-1}, a_{H-1}, r_{H-1}), (o_H, \_, \_) \rangle \tag{22}$$

generated from s_0 by choosing each a_i uniformly from the set of actions.

As was the case for the depth of our trajectory trees in Section 4.2, we will choose the length H of our random trajectories to be the ϵ-horizon time.

In the method of trajectory trees, we used a (stronger) generative model to create a finite set of trajectory trees, and proved that this set gave uniformly good estimates of expected return within Π. Here the proposal is even simpler, but its analysis will be more challenging: we will simply take m truly random histories, derive from these histories an estimate of $V^\pi(s_0)$ for every $\pi \in \Pi$, and show that a relatively small value for m again yields uniformly good estimates.

For ease of exposition only, we will only consider the case of classes of *deterministic* strategies in the remainder of this section (except for some brief occasional remarks) on the random trajectory method. However, we emphasize that uniform convergence results analogous to those we will prove below for this method hold for the general case of classes of stochastic strategies. The uniform convergence tools required for the analysis of this general case are more difficult and beyond the scope of this paper. For the reader knowledgeable on the topic of uniform convergence, the generalization requires graduating from the standard notion of VC dimension to the more complex combinatorial dimension or pseudo-dimension, entirely analogous to the difference between classification and regression in supervised learning. Details will be provided in a later version of the paper.

Recall that each trajectory tree T allowed us to get an evaluation of *any* deterministic strategy π, since any π always defines some path in T. But if t is just a single random history, how can we evaluate an arbitrary π on t, given that π may diverge from t? The answer is that we cannot. Instead let us define the variable $\mathrm{acc}_\pi(t)$ to be 1 if π "accepts" the history t and 0 otherwise. More precisely, if

$$t = \langle (o_0, a_0, r_0), \ldots, (o_{H-1}, a_{H-1}, r_{H-1}), (o_H, \_, \_) \rangle \tag{23}$$

denote the prefixes of t by

$$t[j] = \langle (o_0, a_0, r_0), \ldots, (o_j, \_, \_) \rangle. \tag{24}$$

For each j, let $q_{\pi,j}$ be 1 if π, given as input the history $t[j]$, generates the action a_j, and 0 otherwise. Then $\mathrm{acc}_\pi(t) = \prod_j q_{\pi,j}$.

We can now define the estimate of $V^\pi(s_0)$ we derive from a set of random trajectories. Let $\mathcal{T} = \{t_1, \ldots, t_m\}$ be a set of histories from the random trajectory generator. For each strategy $\pi \in \Pi$, define $\hat{V}^\pi(s_0)$ as follows. Let $S_\pi(\mathcal{T}) \subseteq \mathcal{T}$ include all the histories for which $\mathrm{acc}_\pi(t) = 1$; thus,

$$S_\pi(\mathcal{T}) = \{t | t \in \mathcal{T} \text{ and } \mathrm{acc}_\pi(t) = 1\}. \tag{25}$$

Then $\hat{V}^\pi(s_0)$ is the average return of the histories in $S_\pi(\mathcal{T})$:

$$\hat{V}^\pi(s_0) = 1/|S_\pi(\mathcal{T})| \sum_{t \in S_\pi(\mathcal{T})} r(t) \tag{26}$$

where $r(t)$ is the discounted sum of returns along t.

Thus, in analogy with Section 4.2, we now have a method of generating a set of m random observable histories $\mathcal{T} = \{t_1, \ldots, t_m\}$ (as opposed to m trajectory trees), and for any strategy π, there is a well-defined estimate $\hat{V}^\pi(s_0)$ based on the set $\mathcal{T}$. As in Section 4.2, we wish to establish a nontrivial relationship between the "sample size" m and the deviations $|V^\pi(s_0) - \hat{V}^\pi(s_0)|$ for all $\pi \in \Pi$. We again begin with the case of finite Π.

4.3.1 Random Trajectory Analysis: Finite Strategy Classes

Let D_π be the distribution on observable histories t induced by π; thus, $D_\pi(t)$ is the probability that following policy π from s_0 for H steps generates t. Similarly, let $D_\$$ be distribution on observable histories induced by the truly random trajectory generator.

We start with a few simple lemmas about these distributions. We leave the straightforward proofs for these lemmas to the reader. Lemma 6 establishes that for any fixed strategy π, there is a $1/2^H$ probability that the random trajectory generator will produce a history that is accepted by (that is, consistent with) π. Lemma 7 establishes that for any fixed strategy π, a history generated by the random trajectory generator and accepted by π can be viewed as a Monte Carlo trial of π.

Lemma 6 *For any strategy π,*

$$\mathbf{Pr}_{t\sim D_\$}[\mathrm{acc}_\pi(t) = 1] = \frac{1}{2^H}. \tag{27}$$

Lemma 7 *For any policy π, and any augmented history t,*

$$D_\pi(t) = D_\$(t|\mathrm{acc}_\pi(t) = 1). \tag{28}$$

Next we show that, if we let the set $\mathcal{T}$ of histories returned by the random trajectory generator be sufficiently large, with high probability, the sets S_π of accepted histories are "large" for every $\pi \in \Pi$, where Π is a finite class of deterministic strategies.

Lemma 8 *Let $\mathcal{T}$ be a set of m observable histories returned by the random trajectory generator, and let Π be a finite class of L deterministic strategies. Provided that $m > 2^{H+3}\log(2L/\delta)$, the probability that there exists a strategy $\pi \in \Pi$ for which $|S_\pi(\mathcal{T})| < m/2^{H+1}$ is at most $\delta/2$.*

Proof: For a given strategy π, the expected number of trajectories in S_π is $\sum_{t\in\mathcal{T}}\mathrm{acc}_\pi(t) = m/2^H$ by Lemma 6. Therefore the probability that π will accept fewer than $m/2^{H+1}$ trajectories is bounded by $e^{-m/2^{H+3}}$. The probability that some π has $|S_\pi| < m/2^{H+1}$ is bounded by $Le^{-m2^{H+3}}$. $\square$

The final step in the finite Π analysis is showing that if S_π is sufficiently large, then the value of $\hat{V}^\pi(s_0)$ is a good approximation of the expected return of π.

Lemma 9 *Let $\mathcal{T}$ be a set of observable histories returned by the random trajectory generator, and let Π be a finite class of L deterministic strategies. Conditioned on the event that*

$$|S_\pi(\mathcal{T})| > m_1 = \left(\frac{V_{\max}}{\epsilon}\right)^2 \log(2L/\delta) \tag{29}$$

for every π in Π, with probability $1 - \delta/2$,

$$|\hat{V}^\pi(s_0) - V^\pi(s_o)| \leq \epsilon \tag{30}$$

for every $\pi \in \Pi$.

Proof: Under the conditioning event, for a given strategy π the probability that the estimate deviates by ϵ is at most $e^{-\epsilon^2 m_1/(4{V_{max}}^2)}$; here we are using the fact that the histories in $S_\pi(\mathcal{T})$ are distributed according to $D_\pi(\cdot)$ (Lemma 7). The probability that some π deviates by ϵ is thus bounded by $Le^{-\epsilon^2 m_1/{V_{max}}^2}$. □

We can now state and prove our main theorem for the case of finite Π.

Theorem 10 *Let $\mathcal{T}$ be a set of m observable histories returned by the random trajectory generator, and let Π be a finite class of L deterministic strategies. Provided that*

$$m > 2^{H+3}(V_{\max}/\epsilon)^2 \log(2L/\delta) \tag{31}$$

with probability at least $1 - \delta$ we have

$$\left|V^\pi(s_0) - \hat{V}^\pi(s_0)\right| \leq \epsilon \tag{32}$$

simultaneously for all $\pi \in \Pi$.

Proof: By our choice of m, Lemma 8 guarantees that with probability $1-\delta/2$, for each $\pi \in \Pi$ we have that $|S_\pi(\mathcal{T})| > m_1$. By Lemma 9 this implies that, with probability $1 - \delta/2$, we have $\hat{V}^\pi(s_0) - V^\pi(s_o)| \leq \epsilon$ for all $\pi \in \Pi$. □

If we compare Theorem 10 to Theorem 2 (the uniform convergence result for trajectory trees in the finite Π case), we see that the total amount of experience that must be generated in the POMDP is quite similar. The main differences are in how that experienced is *organized* — in one case into trajectory trees, and in the current case a flat list of random histories — and in how it is *generated*, in the current case from a much weaker simulative model than is required to build trajectory trees.

4.3.2 Random Trajectory Analysis: Infinite Strategy Classes

We now move on to sketch the derivation of our results for infinite classes of strategies, still limiting our attention to the case where all strategies are deterministic for simplicity. Furthermore, as was done in Section 4.2.2, we will also limit ourselves to two-action POMDP's, so as to be able to build on the familiar and standard notion of VC dimension. As we have already mentioned, our results generalize naturally to the cases of stochastic strategies and multi-action POMDP's, but require heavier uniform convergence machinery that we defer for a later version of the paper. However, the following sketch of the development should provide an idea of the tools and arguments involved.

Thus, let Π be an infinite class of deterministic strategies in a two-action POMDP. Let us associate with Π the class $\mathcal{F}_\pi^H = \{\mathrm{acc}_\pi(t) : \pi \in \Pi\}$ of binary functions mapping observable histories t of length exactly H to $\{0,1\}$. We first bound the VC dimension of $\mathcal{F}_\pi^H$ in terms of the VC dimension of Π (which is well-defined, since each $\pi \in \Pi$ maps histories of any length to $\{a_1, a_2\}$).

Lemma 11 *Let Π be any class of deterministic strategies in a two-action POMDP, and let $dim(\Pi)$ be the VC dimension of Π. Then*

$$dim(\mathcal{F}_\pi^H) = O(dim(\Pi)\log(H)). \tag{33}$$

Proof: Let $\mathcal{T}$ be any set of observable histories of length H. Note that if $\pi_1, \pi_2 \in \Pi$ have identical behavior on all prefixes of histories in $\mathcal{T}$ — that is, $\pi_1(t[j]) = \pi_2(t[j])$ for all $t \in \mathcal{T}$ and all $1 \le j \le H$ — then we must have $\mathrm{acc}_{\pi_1}(t) = \mathrm{acc}_{\pi_2(t)}$ for all $t \in \mathcal{T}$. Thus, each different labeling of $\mathcal{T}$ by the class $\mathcal{F}_\pi^H$ gives a different labeling by Π of the set of all prefixes of $\mathcal{T}$. If $\mathcal{F}_\pi^H$ shatters d length-H histories, we must have $2^d \le \Phi_{dim(\Pi)}(dH) \le (dH/dim(\Pi))^{dim(\Pi)}$, where $\Phi_d(m)$ is the dichotomy counting function of uniform convergence theory [Vap82]. The lemma follows from some algebraic manipulations. □

Let us review a bit. Lemma 6 shows that the expected fraction of the $t \in \mathcal{T}$ satisfying $\mathrm{acc}_\pi(t) = 1$ is exactly $1/2^H$, while Lemma 11 establishes that the VC dimension of the variables $\mathrm{acc}_\pi(t)$ is bounded. From this it would not be hard to prove a generalization of Lemma 8 — namely, that for sufficiently large sample size m, all the sets $S_\pi(\mathcal{T})$ will be large (or more accurately, $|S_\pi(\mathcal{T})|/m \approx 1/2^H$ for all $\pi \in \Pi$). However, this is not quite enough — merely having large samples of Monte Carlo trials of each

π (which by Lemma 7 is exactly what the $S_\pi(\mathcal{T})$ are) does not ensure that the average return over each $S_\pi(\mathcal{T})$ will be near $V^\pi(s_0)$. In other words, since the reader can readily verify that $V^\pi(s_0) = \mathbf{E}_{t\sim D_\$}[2^H \mathrm{acc}_\pi(t)r(t)]$ and $\hat{V}^\pi(s_0) = (1/m_\pi)\sum_{t\in\mathcal{T}} \mathrm{acc}_\pi(t)r(t)$ where $m_\pi = |S_\pi(\mathcal{T})|$, it is not uniform convergence of the variables $\mathrm{acc}_\pi(t)$ we are concerned with so much as uniform convergence of the variables $\mathrm{acc}_\pi(t)r(t)$. Here is where the need for generalizations of VC dimension such as combinatorial dimension [Hau92] begin to creep in, since the $\mathrm{acc}_\pi(t)r(t)$ are now real-valued rather than binary variables. We will not delve into the details in this version of the paper, but it suffices to say that in the current situation very little of substance changes from the standard binary setting. This is because the multiplication of all the $\mathrm{acc}_\pi(t)$ by the *fixed* (that is, independent of Π) function $r(t)$ does not increase the complexity of the class. The theorem that finally emerges is given below.

Theorem 12 *Let Π be any class of deterministic strategies in a two-action POMDP M, and let $\mathcal{T} = \{t_1, \ldots, t_m\}$ be m observable histories from the random trajectory generator for M. If*

$$m = O\left(\left(\frac{2^H V_{\max}}{\epsilon}\right)^2 (dim(\Pi)\log(H)(H + \log(V_{\max}/\epsilon)) + \log(1/\delta))\right) \tag{34}$$

then with probability at least $1-\delta$ we have

$$|V^{\pi_i}(s_0) - \hat{V}^\pi(s_0)| \leq \epsilon, \tag{35}$$

simultaneously for all $\pi \in \Pi$.

Acknowledgements

We give warm thanks to Satinder Singh for many enlightening discussions and numerous insights on the ideas presented here.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[CB96] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1017–1023, 1996.

[DNM98] Scott Davies, Andrew Y. Ng, and Andrew Moore. Applying online-search to reinforcement learning. In *Proceedings of AAAI-98*, pages 753–760. AAAI Press, 1998.

[Hau92] David Haussler. Decision-theoretic generalizations of the PAC model for neural networks and other applications. *Information and Computation*, 100:78–150, 1992.

[KS98] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the 1998 International Conference on Machine Learning*, pages 260–268. Morgan Kaufmann, 1998.

[KS99] Michael Kearns and Satinder Singh. Finite-sample convergence rates for Q-learning and indirect algorithms. In *Neural Information Processing Systems 12*. MIT Press, to appear, 1999.

[KYK95] H. Kimura, M. Yamamura, and S. Kobayashi. Reinforcement learning by stochastic hill climbing on discounted reward. In *Proceedings of the 12th International Conference on Machine Learning*, pages 295–303, 1995.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[SY94] Satinder Singh and Richard Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.

[Vap82] V.N. Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer, 1982.

Appendix A: Proof of Theorem 1

In this appendix, we prove Theorem 1, our main result for unconstrained planning.

THEOREM 1 *There is a randomized algorithm $\mathcal{A}$ that, given access to a generative model for any MDP M, takes as input any state $s \in S$ and any value $\varepsilon > 0$, outputs an action, and satisfies the following two conditions:*

- *(Efficiency) The running time of $\mathcal{A}$ is $O((kC)^H)$, where*

$$H = \log_\gamma(\lambda/V_{\max}),$$

$$C = \frac{{V_{\max}}^2}{\lambda^2}\left(2H\log\frac{H{V_{\max}}^2}{\lambda^2} + \log\frac{1}{\delta}\right),$$

$$\lambda = (\epsilon(1-\gamma)^2)/4,$$

$$\delta = \lambda/R_{\max},$$

$$V_{\max} = R_{\max}/(1-\gamma).$$

  In particular, the running time depends only on $R_{\max}$, γ, and ε, and does not *depend on $N = |S|$. If we view $R_{\max}$ as a constant, this bound can be written as*

$$\left(\frac{k}{\varepsilon(1-\gamma)}\right)^{O\left(\frac{1}{1-\gamma}\log\left(\frac{1}{\varepsilon(1-\gamma)}\right)\right)}. \tag{36}$$

- *(Near-Optimality) The value function of the stochastic policy implemented by $\mathcal{A}$ satisfies*

$$|V^{\mathcal{A}}(s) - V^*(s)| \le \varepsilon \tag{37}$$

  simultaneously for all states $s \in S$.

Throughout the analysis we will rely on the pseudo-code provided for algorithm $\mathcal{A}$ given in Figure 1.

The claim on the running time is immediate from the definition of algorithm $\mathcal{A}$. Each call to **EstimateQ** generates kC calls to **EstimateV**, C calls for each action. Each recursive call also reduces the depth parameter h by one, so the depth of the recursion is at most H. Therefore the running time is $O((kC)^H)$. The main part of the proof is showing that our setting of the parameters C and H results in computing a near-optimal policy.

The main effort is in showing that the values of **EstimateQ** are indeed good estimates of Q^*. There are two source of inaccuracy in these estimates. The first is that we use only a finite sample to approximate an expectation

— we draw only C states from the next-state distributions. The inaccuracy due to this step will be handled in a fairly straightforward way. The second source of inaccuracy is that in computing **EstimateQ** we are not actually using the values of $V^*(\cdot)$ but rather values returned by **EstimateV**, which are themselves only estimates. The crucial step in the proof is to show that as h increases, the overall inaccuracy *decreases*.

Let us first define an intermediate random variable that will capture the inaccuracy due to the limited sampling. Define $U^*(s,a)$ as follows:

$$U^*(s,a) = R_{s,a} + \gamma \frac{1}{C} \sum_{i=1}^{C} V^*(s_i) \tag{38}$$

where the s_i are drawn according to $P_{s,a}(\cdot)$. Note that $U^*(s,a)$ is averaging values of $V^*(\cdot)$, the unknown value function. Since $U^*(s,a)$ is used only for the proof, and not in the algorithm, there is no problem in defining it this way.

We first claim that with high probability, the difference between $U^*(s,a)$ and $Q^*(s,a)$ is at most λ.

Lemma 13 *For any state s and action a, with probability at least $1 - e^{-\lambda^2 C / V_{\max}{}^2}$ we have*

$$|Q^*(s,a) - U^*(s,a)| = \gamma \left| \mathbf{E}_{s \sim P_{s,a}(\cdot)}[V^*(s)] - \frac{1}{C} \sum_i V^*(s_i) \right| \leq \lambda, \tag{39}$$

where the probability is taken over the draw of the s_i from $P_{s,a}(\cdot)$.

Proof: Note that $Q^*(s,a) = R_{s,a} + \gamma \mathbf{E}_{s \sim P_{s,a}(\cdot)}[V^*(s)]$. The proof is immediate from the Chernoff bound. □

Now that we have quantified the error due to finite sampling, we can bound the error due to the fact we are using values returned by **EstimateV** rather than $V^*(\cdot)$. We will bound this error as the difference between the values $U^*(s,a)$ and **EstimateV**. In order to make our notation simpler, let $V^n(s)$ be the value returned by the call **EstimateV**(n, C, γ, G, s), and let $Q^n(s,a)$ be the component in the output of **EstimateQ**(n, C, γ, G, s) that corresponds to action a. Using this notation, our algorithm computes

$$Q^n(s,a) = R_{s,a} + \gamma \frac{1}{C} \sum_i V^{n-1}(s_i) \tag{40}$$

where $V^{n-1}(s) = \max_a\{Q^{n-1}(s,a)\}$, and $Q^0(s,a) = 0$ for every state s and action a.

We now define a parameter α_n that we will eventually show bounds the difference between $Q^*(s,a)$ and $Q^n(s,a)$ — that is, $|Q^*(s,a) - Q^n(s,a)| \leq \alpha_n$. We define α_n recursively:

$$\alpha_{n+1} = \gamma(\lambda + \alpha_n) \tag{41}$$

where $\alpha_0 = V_{max}$. Solving for α_H we obtain

$$\alpha_H = \left(\sum_i \gamma^i \lambda\right) + \gamma^H V_{max} \leq \frac{\lambda}{1-\gamma} + \gamma^H V_{max}. \tag{42}$$

The next lemma bounds the error in the estimation, at level n, by α_n. Intuitively, the error due to finite sampling contributes λ, while the errors in estimation contribute α_n. The combined error is $\lambda + \alpha_n$, but since we are discounting, the effective error is only $\gamma(\lambda + \alpha_n)$, which by definition is α_{n+1}.

Lemma 14 *With probability at least* $1 - (kC)^n e^{-\lambda^2 C/V_{\max}{}^2}$ *we have that*

$$|Q^*(s,a) - Q^n(s,a)| \leq \alpha_n. \tag{43}$$

Proof: The proof is by induction on n. It clearly holds for $n = 0$. Now

$$\begin{aligned}
|Q^*(s,a) - Q^n(s,a)| &= \gamma \left| \mathbf{E}_{s\sim P_{s,a}(\cdot)}[V^*(s)] - \frac{1}{C}\sum_i V^{n-1}(s_i) \right| \\
&\leq \gamma \left(\left| \mathbf{E}_{s\sim P_{s,a}(\cdot)}[V^*(s)] - \frac{1}{C}\sum_i V^*(s_i) \right| \right. \\
&\qquad \left. + \left| \frac{1}{C}\sum_i V^*(s_i) - \frac{1}{C}\sum_i V^{n-1}(s_i) \right| \right) \\
&\leq \gamma(\lambda + \alpha_n) \\
&= \alpha_{n+1}
\end{aligned}$$

We require that all of the C child estimates be good, for each of the k actions. This means that the probability of a bad estimate increases by a factor of kC, for each n. By Lemma 13 the probability of a single bad estimate is bounded by $e^{-\lambda^2 C/V_{max}{}^2}$. Therefore the probability of some bad estimate is bounded by $1 - (kC)^n e^{-\lambda^2 C/V_{max}{}^2}$. □

From the bound $\alpha_H \leq \gamma^H V_{max} + \lambda/(1-\gamma)$, we can see that for

$$H = \log_\gamma(\lambda/V_{max}) = O((1/1-\gamma)\log(V_{max}/\lambda)) \tag{44}$$

with probability $1-(2C)^H e^{-\lambda^2 C/V_{max}{}^2}$ all the final estimates $Q^H(s_0,a)$ are within $2\lambda/(1-\gamma)$ from the true Q-values. The next step is to choose C such that $\delta = (2C)^H e^{-\lambda^2 C/V_{max}{}^2}$, which will bound the probability of a bad estimate during the entire computation. Specifically,

$$C = \frac{V_{max}{}^2}{\lambda^2}\left(2H\log\frac{HV_{max}{}^2}{\lambda^2} + \log\frac{1}{\delta}\right) \tag{45}$$

is sufficient to ensure that with probability $1-\delta$ all the estimates are accurate.

At this point we have shown that with high probability, algorithm $\mathcal{A}$ computes a good estimate of $Q^*(s_0,a)$ for all a, where s_0 is the root (input) state. To complete the proof, we need to relate this high probability result to the expected value of a stochastic policy. We will prove a fairly general result about MDP, which does not depend on our specific algorithm.

Lemma 15 *Assume that π is a stochastic policy, so that $\pi(s)$ is a random variable. If for each state s, the probability that $Q^*(s,\pi^*(s)) - Q^*(s,\pi(s)) < \lambda$ is at least $1-\delta$, then the discounted infinite horizon return of π is at most $(\lambda + \delta V_{\max})/(1-\gamma)$ from the optimal return, that is, for any state s $V^*(s) - V^\pi(s) \leq (\lambda + 2\delta V_{\max})/(1-\gamma)$.*

Since we assume that the rewards are bounded, it implies that the expected return $\mathbf{E}[Q^*(s,\pi(s))]$ at each state s is at least

$$(1-\delta)(Q^*(s,\pi^*(s)) - \lambda) - \delta V_{max} \geq Q^*(s,\pi^*(s)) - \lambda - 2\delta V_{max}. \tag{46}$$

Now we show that if π has the property that at each state s the difference between $\mathbf{E}[Q^*(s,\pi(s))]$ and $Q^*(s,\pi^*(s))$ is at most β, then $V^*(s) - V^\pi(s) \leq \beta/(1-\gamma)$. (A similar result was proved by Singh and Yee [SY94], for the case that *each* action chosen has $Q^*(s,\pi^*(s)) - Q(s,\pi(s)) \leq \beta$. It is easy to extend their proof to handle the case here, and we sketch a proof only for completeness.)

The assumption on the Q^* values immediately implies $|R(s,\pi^*(s)) - \mathbf{E}[R(s,\pi(s))]| \leq \beta$. Consider a policy π_j that executes π for the first $j+1$ steps and then executes π^*. We can show by induction on j that for every state s, $V^*(s) - V^{\pi_j}(s) \leq \sum_{i=0}^{j} \beta\gamma^i$. This implies that $V^*(s) - V^{\pi}(s) \leq \sum_{i=0}^{\infty} \beta\gamma^i = \beta/(1-\gamma)$.

By setting $\beta = \lambda + 2\delta V_{max}$ the lemma follows. □

Proof of Theorem 1: As discussed before, the running time is immediate from the algorithm, and the main work is showing that we compute a near-optimal policy. By Lemma 14 we have that the error in the estimation of Q^* is at most α_H, with probability $1-(kC)^H$. Using the values we chose for C and H we have that with probability $1-\delta$ the error is at most $2\lambda/(1-\gamma)$. By Lemma 15 this implies that such a policy π has the property that from every state s,

$$V^*(s) - V^{\pi}(s) \leq \frac{2\lambda}{(1-\gamma)^2} + \frac{2\delta V_{max}}{1-\gamma}. \tag{47}$$

Now we can use the values that we chose for $\delta = \lambda/R_{max}$ and $\lambda = \epsilon(1-\gamma)^2/4$. From this it follows that,

$$V^*(s) - V^{\pi}(s) \leq \frac{2\lambda}{(1-\gamma)^2} + \frac{2\delta V_{max}}{1-\gamma} = \frac{4\lambda}{(1-\gamma)^2} = \epsilon. \tag{48}$$

□

Appendix B: Efficient Algorithms for Stochastic Gradient Ascent

In this appendix, we sketch the gradient ascent ideas described in Section 4.2 for efficiently performing stochastic gradient ascent either on the empirical trajectory tree return $(1/m)\sum_{i=1}^{m} R(\pi_\theta, T_i)$ or directly on $V^{\pi_\theta}(s_0)$, the latter of which eventually leads us close to a line of research pursued by Kimura, Yamamura and Kobayashi [KYK95]. We now start by treating gradient ascent on $(1/m)\sum_{i=1}^{m} R(\pi_\theta, T_i)$, giving an algorithm that should be used in conjunction with lazy construction of the trees T_i.

Definition 8 *A* **smoothly parameterized family of stochastic strategies** *parameterized by* $\mathbb{R}^d$ *for a* k*-action POMDP is a set of stochastic strategies* $\Pi = \{\pi_\theta | \theta \in \mathbb{R}^d\}$*, such that for any action* a *and fixed observable history* $t = \langle (o_1, a_1, r_1), \ldots, (o_{i-1}, a_{i-1}, r_{i-1}), (o_i, \_, \_) \rangle$, $\mathbf{Pr}[\pi_\theta(t) = a]$ *is continuously differentiable in* θ.

To simplify our notation, we use $(\pi_\theta(t))_i$ to denote $\mathbf{Pr}[\pi_\theta(t) = a_i]$; a convenient way to think about it is as π_θ mapping finite histories to k-dimensional vectors (in the $k-1$-dimensional simplex), where the i-th element of the output is the probability of choosing action a_i. Since our goal

Function: **EstimateTreeGradient**$(\Pi, \theta_0, T, \gamma)$
Input: Strategy class Π, parameter setting θ_0, trajectory tree T, discount γ.
Output: A vector that is an unbiased estimate of $\frac{d}{d\theta}|_{\theta=\theta_0} R(\pi_\theta, T)$.

1. Randomly choose a depth $d \in \{0, \ldots, H-1\}$, with $\mathbf{Pr}[d = h] \propto \gamma^h$.

2. Perform a random walk on the tree T starting from the root with actions chosen by π_{θ_0} (that is, where the action at each node is determined by π_{θ_0} applied to the history of the path to that node), down to depth d on the tree. Let s_d denote the node we arrive at, and t_d the history of s_d.

3. For each action $a \in a_1, \ldots, a_k$, start from the node s_d, execute action a (meaning go to the action-a child), and the again follow π_{θ_0} down the tree until we reach a leaf at depth H. Let r_a be the sum of discounted reinforcements received executing the trajectory from s_d down to depth H taking action a at s_d, where the discounting starts at 1, not γ^d, for the sequence. (Thus, the reinforcement for taking a at s_d is undiscounted, the next reinforcement is discounted by γ and so on, with the last step's reinforcement discounted by γ^{H-d-1}.)

4. Output
$$\frac{1-\gamma^H}{1-\gamma} \sum_{i=1}^{k} r_{a_i} \frac{d}{d\theta}\bigg|_{\theta=\theta_0} (\pi_\theta(t_d))_i \tag{49}$$
(where as before, $(\pi_\theta(t_d))_i$ denotes $\mathbf{Pr}[\pi_\theta(t_d) = a_i]$).

Figure 5: Algorithm for finding an unbiased estimate of the gradient of $R(\pi_\theta, T)$ with respect to θ.

is, given a current θ_0, to find a bounded-variance estimate of a gradient involving π_θ, we clearly also need some bound on how much $(\pi_\theta(t))_i$ itself varies with θ. Thus, we assume that for all finite histories t and all i, $|\frac{d}{d\theta}(\pi_\theta(t))_i| \leq B$ for some constant B. Finally, to reduce clutter in our notation, we will frequently drop the "$|_{\theta=\theta_0}$" to denote evaluation at the current θ_0 (for example, as in $\frac{d}{d\theta}|_{\theta=\theta_0} R(\pi_\theta, T)$) when there is no risk of confusion.

We are now ready to describe our algorithm for finding an unbiased estimate of the gradient of $(1/m)\sum_{i=1}^m R(\pi_\theta, T_i)$ with respect to θ. Let us fix one trajectory tree T and describe how we find an unbiased estimate of the gradient of $R(\pi_\theta, T)$; the extension to a sum over m trees is then trivial.

Figure 5 describes our algorithm for finding our gradient estimate. (For brevity, it is stated in a slightly less formal way than our previous algorithms.) We now sketch its rationale and the strategy for how we prove its correctness – that it gives an unbiased and bounded-variance estimate of the gradient (running time $O(kH)$ is obvious) – before giving the result more formally. Our strategy will be, for the fixed tree T, to consider the change to $R(\pi_\theta, T)$ if any of the probabilities $(\pi_\theta(t_i))_j$ change for some node i in the tree, where t_i denotes the observed history at node i. Because of the discounting, the result of a change at some node i at depth d in the tree has weight proportional to γ^d, because it is d steps removed from s_0 and the effects of a change there is therefore discounted by γ^d. Moreover, along a different dimension of "weighting," the same node i also, among all the nodes at depth d, has a (separate) weight proportional to the probability that a random walk according to π_{θ_0} will reach node i. For example, if a node i is never reached when executing π_{θ_0}, then the action-probabilities at node i are irrelevant to $R(\pi_{\theta_0}, T)$, and so the contribution of node i's action to the overall gradient should have a weight of 0. Finally, we will see that what the algorithm is doing is randomly sample a node s_d in the tree with probably proportional to its weight – this is why the depth-h layer of the tree has weight proportional to γ^h (Step 1 of the algorithm), and why we then sample one of these nodes with probability equal to the probability that a random walk will reach it (Step 2 of the algorithm). Having randomly sampled such a node of the tree with probability proportional to its weight, we output just the component of the gradient that this node contributes to, in the style of standard Monte Carlo estimates of means. The $(1-\gamma^H)/(1-\gamma)$ factor in our output is then just a normalization constant of the sampling distribution that makes sure we do not change the expectation. Here follows a slightly more rigorous proof that **EstimateTreeGradient** outputs an unbiased and bounded-variance (bounded in terms of k, γ, R_{max}, and B) estimate of $\frac{d}{d\theta}|_{\theta=\theta_0} R(\pi_\theta, T)$.

Proof (Sketch): Let $\ell = 2^{H+1}-1$ be the number of nodes in our trajectory tree T, and let us fix some labeling of the nodes from 1 to ℓ, so that we may refer to the "i-th node" in the tree. Also, for $i = 1, \ldots, \ell$, let t_i denote the history at the i-th node in the tree (from the root down to that node), and let $p_{ij} = (\pi_\theta(t_i))_j$ be the probability of taking action a_j at node i, as an implicit function of θ. Then for fixed T, $R(\pi_\theta, T)$ is completely determined by $\{p_{ij} | i = 1, \ldots, \ell, j = 1, \ldots, k\}$, since the p_{ij} completely specify the probabilities of taking each action at any node in the tree T. Thus writing R to denote $R(\pi_\theta, T)$, by the chain rule of differentiation,

$$\frac{d}{d\theta}R(\pi_\theta, T) = \sum_{i=1}^{\ell}\left(\sum_{j=1}^{k}\frac{dR}{dp_{ij}}\frac{dp_{ij}}{d\theta}\right) \tag{50}$$

where we recall $p_{ij} = (\pi_\theta(t_i))_j$ are implicit functions of θ. We compare this with (49), the expression of the algorithm's output in Figure 5, and notice (49) bears some similarity to the term in parenthesis above, with r_{a_j} playing the role of $\frac{dR}{dp_{ij}}$. Now notice also that, if p_{ij} is increased by ϵ, then $R = R(\pi, T)$ will increase by ϵ times the probability of reaching the subtree rooted at node i times the average reward of the action-a_j subtree of node i. But if the i-th node is s_d, then $\gamma^d r_{a_j}$ is an unbiased estimate of the (discounted) value of this action-a_j subtree, since r_{a_j} is just a Monte Carlo estimate of the value of this subtree. Letting $P[i]$ denote the probability that a π_{θ_0}-random walk will reach node i, we therefore see, by our argument, that $P[i]\gamma^d r_{a_j}$ is in fact an *unbiased* estimate of $\frac{dR}{dp_{ij}}$ when the node i is s_d (e.g. when r_{a_j} is gotten from a random walk as in Step 3 of the algorithm, starting from node i and taking a_j first). Putting this back into Equation 50 and letting $d(i)$ denote the depth of node i in the tree, we see that

$$\begin{aligned}\frac{d}{d\theta}R(\pi_\theta, T) &= \sum_{i=1}^{\ell}\left(\sum_{j=1}^{k}\gamma^{d(i)}\mathrm{P}[i]\mathbf{E}[r_{a_j}|s_d = \text{node } i]\frac{dp_{ij}}{d\theta}\right) && (51)\\ &= \sum_{i=1}^{\ell}\gamma^{d(i)}\mathrm{P}[i]\left(\sum_{j=1}^{k}\mathbf{E}[r_{a_j}|s_d = \text{node } i]\frac{dp_{ij}}{d\theta}\right) && (52)\end{aligned}$$

This is just a sum over nodes i, with the node-i term having a "weight" of $\gamma^{d(i)}\mathrm{P}[i]$. To find an unbiased estimate of the sum, we may, as in Steps 1-2 of the algorithm, randomly choose one of the terms with probability proportional to its weight and then (with appropriate normalization, by $\sum_{i=i}^{\ell}\gamma^{d(i)}\mathrm{P}[i] = (1-\gamma^H)/(1-\gamma)$ in our case to make the "weights" sum to one and be a probability distribution) output just that one term in the sum, to get an unbiased estimate of the entire sum. This is of course just a standard Monte Carlo estimate of a mean. Doing this, and further replacing the expectation with a sample of r_{a_j} then recovers the expression (49) used in our algorithm.

Finally, by our assumption of $|\frac{d}{d\theta}(\pi_\theta(t))_i|$ being bounded as $\leq B$, we see the output (49) is also bounded (in terms of k, γ, R_{max}, and B). So, our estimator is also bounded, and therefore has bounded variance. □

We are now ready to move on to finding unbiased estimates of $\frac{d}{d\theta}V^{\pi_\theta}(s_0)$. Kimura et. al. [KYK95] have a nice algorithm in which they try to do something similar, though their algorithm only converges asymptotically to an unbiased estimate, and even then can have arbitrarily large (or even infinite) variance (though we will also describe a criteria that would ensure bounded variance for their algorithm). Here, we describe an approach that is a simple extension of the **EstimateTreeGradient** algorithm we have described so far, and which will terminate in finite time (in fact with expected running time $O(k/(1-\gamma))$,) and output an unbiased, bounded-variance estimate of the desired gradient.

Given what we have done so far, the idea is simple. Consider building an "infinite"-depth trajectory tree T (lazily, of course!) so that $\mathbf{E}_T[R(\pi_{\theta_0}, T)] = V^{\pi_{\theta_0}}(s_0)$, exactly, and

$$\mathbf{E}_T\left[\frac{d}{d\theta}R(\pi_\theta, T)\right] = \frac{d}{d\theta}V^{\pi_{\theta_0}}(s_0) \tag{53}$$

(To see this, apply $\frac{d}{d\theta}$ to both sides of the equality we just pointed out before this equation, and then, since differentiation and expectation are both linear operators, they may be exchanged, giving this result.[2]) Thus, if we could apply our **EstimateTreeGradient** algorithm to a randomly sampled "infinite" trajectory tree T, we are done.

If we do apply **EstimateTreeGradient** to an "infinite" tree, we encounter no difficulties in the first and second steps – we choose a depth $d \in \{0, 1, 2, \ldots\}$ according to a geometric distribution ($\mathbf{Pr}[d = h] = \gamma^h/(1-\gamma)$,) and then take a random walk down to depth d. But then in Step 3, we require a further "infinite" random walk to the infinitely distant leaves, and this we cannot do.

But fortunately, a simple trick saves us: it is well-known that the infinite-horizon value function of a discounted POMDP is the same as that of an *undiscounted* POMDP, but where there is a $1-\gamma$ probability of transitioning to a zero-reward absorbing state after each step. Thus, rather than summing the undiscounted sum of rewards in the discounted POMDP, we may imagine ourselves to be in the undiscounted POMDP, and take a random trajectory according to π_{θ_0} as before, except the trajectory has a $1-\gamma$ probability of terminating (in a "zero-reward absorbing state") at each step. It is easy

[2] For readers familiar with measure theory, we should comment it can indeed separately be verified, via a Fubini-style argument, that the interchanging of the two operators is indeed valid.

to see that this takes an average of $1/(1-\gamma)$ steps to terminate, giving an expected running time of $O(k/(1-\gamma))$. Moreover, because each output value is then a sum of a geometrically distributed number of individually bounded terms, and because the geometric distribution has bounded variance (for fixed γ,) our estimator also has bounded variance.

Figure 6, with the parameter $H' = 0$, is algorithm we have just described. The parameter H' is a slight modification that we added to try to reduce the variance of our estimator. Basically, the problem is that, in the new formulation of the algorithm, the value we are returning involves the sum of d' reinforcement terms from the "undiscounted" POMDP, where d' is geometrically distributed. The variability in the number of terms we are summing adds another source of variance to our final estimator. Instead, with the H' "execution depth" parameter, we will always take at least H' steps from s_d, after which we take a further, geometrically distributed number of steps, except now the contribution of the geometrically distributed number of steps is discounted by $\gamma^{H'}$. Thus, with H' set on the order of $1/(1-\gamma)$ or to the ϵ-horizon time, we incur an additional expected H' steps per run, but can have a significantly smaller variance in our estimator.

Finally, to close this section, we very briefly address a number of issues (perhaps the most pressing of which is the lack of mention of exploration), consider possible modifications to the algorithm, and relate it also the Kimura et. al.'s algorithm.

Exploration. We have presented a reinforcement learning algorithm that we have argued can perform stochastic gradient ascent to optimize $V^{\pi_0}(s_0)$, but which does not seem to need to do any exploration. Is there any magic here? It turns out that, even though we have bounded the variance of our estimators of our gradient, the *gradient* itself may still be exponentially small in H. To see this, imagine a trajectory tree where only one of the leaves gives a reward; if we were choosing actions at random, it would take us $O(2^H)$ tries to find this node. Thus, this reinforcement learning algorithm may be best applied only if exploration is not really needed for the domain (for instance, if there are many intermediate rewards,) or if we are somehow able to initialize θ_0 to a "good" strategy, in which case this algorithm may be used to improve upon it from there. A particularly intriguing example is if we were to, by other means, estimate the Q-function (or value function) as $\hat{Q}_\theta(s,a)$, using some smooth function approximator parameterized by θ. Then adding a inverse-temperature parameter β, we may define a stochastic policy $(\pi_{\theta,\beta}(s))_j = \exp(\beta\hat{Q}_\theta(s,a_j))/\sum_i \exp(\beta\hat{Q}_\theta(s,a_i))$, and then

Function: **EstimateValueGradient**$(\Pi, \theta_0, s_0, \gamma, G, H')$
Input: strategy class Π, current parameters θ_0, start state s_0, discount γ generative model G, execution depth H'.
Output: A row vector that is an unbiased estimate of $\frac{d}{d\theta}|_{\theta=\theta_0} V^{\pi_\theta}(s_0)$.

1. Set s to be the root (s_0) node in the (to be lazily built) tree.

2. With probability $1-\gamma$, jump to step 4.

3. Let a be an action chosen randomly by the stochastic policy π_{θ_0} applied to the history at node s. Take action a from s, setting s to be the action-a child of s. Goto step 2.

4. Set node $s_d = s$, and let t_d be the history at s.

5. For each action $a \in \{a_1, \ldots, a_k\}$
   5.1 Set $r_a := 0$.
   5.2 (Using G,) execute action a from state s_d, getting some resulting reinforcement R and child-node s'. Set $r_a := r_a + R$.
   5.3 For $i = 1$ to H',
   5.3.1 (Using G), take a random step according to π_{θ_0} applied to the history at s, letting R be the reinforcement and s' the action's corresponding child node.
   5.3.2 Set $r_a := r_a + \gamma^i R$, and $s := s'$.
   5.4 With probability $1-\gamma$, jump to 5.6.
   5.5 (Using G) Take a random step from s according to π_{θ_0}, getting some resulting reinforcement R and next-state s'. Set $r_a := r_a + \gamma^{H'} R$, and $s := s'$. Goto step 5.4.
   5.6 end. (of for-loop)

6. Output

$$\frac{1}{1-\gamma} \sum_{j=1}^{k} \frac{d(\pi_\theta(s_d))_j}{d\theta}\bigg|_{\theta=\theta_0} r_{a_j} \tag{54}$$

Figure 6: Algorithm for finding an unbiased estimate of $\frac{d}{d\theta}|_{\theta=\theta_0} V^{\pi_\theta}(s_0)$.

do gradient ascent on $V^{\pi(\theta,\beta)}(s_0)$ with parameters (θ, β).

Deterministic strategies. When learning deterministic strategies, the main algorithm we suggested was local greedy search with lazy tree construction. To avoid local maxima, it may also be worthwhile *embedding* our class of deterministic strategies in a larger, smoothly parameterized class of stochastic strategies, and then doing the search in that larger, smoother space, for example with our stochastic gradient ascent algorithm. Then, a standard trick from the neural network literature called the soft-barrier may be used to re-constrain ourselves to deterministic strategies (if so desired). Rather than doing gradient ascent on $V^{\pi_\theta}(s_0)$, we do gradient ascent on $V^{\pi_\theta}(s_0) - \beta \mathbf{E}[\mathcal{H}(\pi(t))]$ where $\mathcal{H}$ is the entropy function and the expectation is over an appropriate distribution of histories (say choose $t = t_d$, the history at s_d, on each step of the algorithm). This way, we penalize strategies that have entropy or randomness in their choices of actions. So, letting β slowly increase from 0 to ∞, we have an increasing bias towards deterministic strategies. Note however that we do not recommend learning a stochastic strategy π_θ, and then making it deterministic by picking the most likely action (under π_θ) at each step; this may well work for some problems, but seems to us much more difficult to justify.

A weaker simulator. Throughout this section, we have assumed a generative model that allows us to sample a next-state/next-observation pair starting from any state. What if, as in 4.3, our generative model only allows us to "reset" to s_0 and to take Monte Carlo runs from there, but could not reset to any intermediate state? This is easily handled. Recall that, after reaching s_d in our algorithm, we iterated through all k actions, and then our final output in (54) is a value that is a sum of k terms, with one term corresponding to each Monte Carlo run from s_0. If we have only the weaker simulator, we are able to try only one Monte Carlo run from s_d. To handle this, we may randomly choose one action, take just that one Monte Carlo run, and then return k times the one term (within the summation) corresponding to the chosen action. It is easy to show this still gives a bounded-variance unbiased estimator. Incidentally, if we have an estimate of the Q-function at s_d, this can also be used to reduce the variance resulting from the randomization of the action's choice.

A random walk? Kimura et. al. had some nice ideas regarding this. Using the weaker simulator described in the previous paragraph, we uniformly chose a random action upon reaching s_d, and then gave that one term in the sum an multiplier of k. Alternatively, we may choose our action at s_d according to the distribution given by $\pi_{\theta_0}(t_d)$ also, so that at every step we are taking a random walk according to π_{θ_0}. Then, to keep our estimator unbiased, we would multiply by $1/(\pi_{\theta_0}(t_d))_j$ if action a_j was chosen, making the algorithm's final output

$$\frac{1}{1-\gamma}\frac{1}{(\pi_{\theta_0}(t_d))_j}\left.\frac{d(\pi_\theta(t_d))_j}{d\theta}\right|_{\theta=\theta_0} r_{a_j} \tag{55}$$

This should be compared with (54). Unfortunately, the $1/(\pi_{\theta_0}(t_d))_j$ multiplier may be arbitrarily large, and so our estimator may have arbitrarily large or even infinite variance (and examples are easy to construct). This (minus the undiscounted POMDP idea to make the algorithm terminate in finite time, rather than only converge asymptotically) was essentially Kimura et. al.'s algorithm. (Actually, they also had a second version, and some interesting ideas regarding eligibility traces, that we will mention shortly.) Particularly if used with the method we described earlier to learn deterministic strategies, this would seem particularly disastrous, as we are driving most of the $(\pi_{\theta_0}(t_d))_j$ to zero. But fortunately, we found a simple, additional condition to ensure bounded variance; namely if $\left|\frac{1}{(\pi_{\theta_0}(t_d))_j}\frac{d(\pi_\theta(t_d))_j}{d\theta}\right|$ were bounded, then there would be no problems. As a simple example, if the π_θ were a neural network with a sigmoid activation function at the output (that, say, represents probability of taking action a_1) in a two-action POMDP, then probability of taking action a_1 is $(\pi_\theta(t_d))_1 = \sigma(f_\theta(t_d))$ for some f_θ, and its derivative is $\sigma(f_\theta(t_d))(1-\sigma(f_\theta(t_d))f'_\theta(t_d)$. Notice how the $\sigma(\cdot)$ term therefore cancels the $1/(\pi_{\theta_0}(t_d))_1 = 1/\sigma(\cdot)$ term, and so we are no longer worried about the estimator blowing up (making appropriate assumptions about f). A similar argument may be made about softmax (which generalizes sigmoids to many outputs) activation functions for representing k-action stochastic strategies.

Eligibility traces. Related to the last point, if we are taking just one random walk, then it seems that we might as well try letting every node on the random walk be s_d, to get rid of one more source of variance in our estimate. That is, rather than choosing a single depth d at random and then picking a single node s_d at that depth, we might as well enumerate over as many d as we can, letting each node on our random walk be s_d in turn, and then taking the appropiate weighted average. Kimura et. al.'s algorithm had a very nice way of doing this and efficiently taking care of the bookkeeping, by making use of eligibility traces. If indeed $\left|\frac{1}{(\pi_{\theta_0}(t_d))_j}\frac{d(\pi_{\theta_0}(t_d))_j}{d\theta}\right|$ were bounded, then their eligibility trace ideas (coupled with making use of the undiscounted POMDP to ensure finite termination) could be another very viable gradient ascent algorithm.