Communications in Information and Systems
Vol. 7, No. 1, pp. 59-92, 2007

A SURVEY OF SOME SIMULATION-BASED ALGORITHMS FOR MARKOV DECISION PROCESSES

HYEONG SOO CHANG[*], MICHAEL C. FU[†], JIAQIAO HU[‡], AND STEVEN I. MARCUS[§]

Abstract. Many problems modeled by Markov decision processes (MDPs) have very large state and/or action spaces, leading to the well-known curse of dimensionality that makes solution of the resulting models intractable. In other cases, the system of interest is complex enough that it is not feasible to explicitly specify some of the MDP model parameters, but simulated sample paths can be readily generated (e.g., for random state transitions and rewards), albeit at a non-trivial computational cost. For these settings, we have developed various sampling and population-based numerical algorithms to overcome the computational difficulties of computing an optimal solution in terms of a policy and/or value function. Specific approaches presented in this survey include multi-stage adaptive sampling, evolutionary policy iteration and evolutionary random policy search.

Key words: (adaptive) sampling, Markov decision process, population-based algorithms

1. Introduction.

Markov decision process (MDP) models are widely used for modeling sequential decision-making problems that arise in engineering, economics, computer science, and the social sciences. However, it is well-known that many real-world problems modeled by MDPs have very large state and/or action spaces, leading to the well-known curse of dimensionality that makes practical solution of the resulting models intractable. In other cases, the system of interest is complex enough that it is not feasible to explicitly specify some of the MDP model parameters, but simulated sample paths can be readily generated (e.g., for random state transitions and rewards), albeit at a non-trivial computational cost. For these settings, we have developed various sampling and population-based numerical algorithms to overcome the computational difficulties of computing an optimal solution in terms of a policy and/or value function. Specific approaches include multi-stage adaptive sampling, evolutionary policy iteration and evolutionary random policy search. This paper brings together these algorithms and presents them in a unified manner. These approaches are distinct from but complementary to those computational approaches for solving MDPs based on explicit state-space reduction, such as neuro-dynamic programming or reinforcement learning; in fact, the computational gains achieved

[*]Department of Computer Science and Engineering, Sogang University, Seoul, Korea, E-mail: hschang@ccs.sogang.ac.kr
[†]Robert H. Smith School of Business & Institute for Systems Research, University of Maryland, College Park, MD 20742, E-mail: mfu@rhsmith.umd.edu
[‡]Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794, E-mail: jqhu@ams.sunysb.edu
[§]Department of Electrical and Computer Engineering & Institute for Systems Research, University of Maryland, College Park, MD 20742, E-mail: marcus@umd.edu

through approximations and parameterizations to reduce the size of the state space can be incorporated into our proposed algorithms.

We begin with a formal description of the general discounted cost MDP framework in Section 2, including both the finite and infinite horizon settings and summarizing the associated optimality equations. We then present the well-known exact solution algorithms, value iteration and policy iteration. We conclude with a brief survey of other recently proposed MDP solution techniques designed to break the curse of dimensionality.

In Section 3, we present simulation-based algorithms for estimating the optimal value function in *finite* horizon MDPs with *large (possibly uncountable) state spaces*, where the usual techniques of policy iteration and value iteration are either computationally impractical or infeasible to implement. We present two adaptive sampling algorithms that estimate the optimal value function by choosing actions to sample in each state visited on a finite horizon simulated sample path. The first approach builds upon the expected regret analysis of multi-armed bandit models and uses upper confidence bounds to determine which action to sample next, whereas the second approach uses ideas from learning automata to determine the next sampled action.

Section 4 considers infinite horizon problems and presents *non-simulation-based* approaches for finding an optimal policy. The algorithms in this chapter work with a *population* of policies — in contrast to the usual policy iteration approach, which updates a single policy — and are targeted at problems with *large action spaces* (again possibly uncountable) and relatively small state spaces (which could be reduced, for instance, through approximate dynamic programming techniques). The algorithms assume that the distributions on state transitions and rewards are known explicitly. Extension to the setting when this is not the case is discussed, where finite-horizon simulated sample paths would be used to estimate the value functions for each policy in the population.

For a more complete discussion of the methods presented in this paper, including convergence proofs and numerical results, the reader is referred to the book [11].

2. Markov Decision Processes and Previous Work.

Define a Markov decision process (MDP) by the five-tuple $(X, A, A(\cdot), P, R)$, where X denotes the state space, A the action space, $A(x) \subseteq A$ denotes the set of admissible actions in state x, $P(x,a)(y)$ is the probability of transitioning from state $x \in X$ to state $y \in X$ when action $a \in A(x)$ is taken, and $R(x,a)$ is the non-negative bounded (by $R_{\max} \in \Re^+$) reward obtained when in state $x \in X$ and action $a \in A(x)$ is taken. More generally, $R(x,a)$ may itself be a random variable, or viewed as the (conditioned on x and a) expectation of an underlying random cost. For simplicity, we will usually assume that X is a countable set, but the discussion and notation can be generalized to uncountable state spaces. We have assumed that the components of the model are stationary (not explicitly time-dependent); the nonstationary case can easily be incorporated into this model by augmenting the state with a time variable.

The evolution of the system is as follows. Let x_t denote the state at time (*stage* or *period*) $t \in \{0, 1, ...\}$ and a_t the action chosen at that time. If $x_t = x \in X$ and $a_t = a \in A(x)$, then the system transitions from state x to state $x_{t+1} = y \in X$ with probability $P(x,a)(y)$, and a reward of $R(x,a)$ is obtained. Once the transition to the next state has occurred, a new action is chosen, and the process is repeated.

Let Π be the set of all (possibly nonstationary) Markovian policies $\pi = \{\pi_t, t = 0, 1, ...\}$, where $\pi_t(x) \in A(x), x \in X$. The goal is to find a policy π that maximizes the *expected total discounted reward* given by

$$V^{\pi}(x) = E\left[\sum_{t=0}^{H-1} \gamma^t R(x_t, \pi_t(x_t)) \middle| x_0 = x\right], \tag{1}$$

for some given initial state $x \in X$, where $0 < \gamma \leq 1$ is the discount factor, and H may be infinite (if $\gamma < 1$). The optimal value will be denoted by

$$V^*(x) = \sup_{\pi \in \Pi} V^{\pi}(x), x \in X, \tag{2}$$

and a corresponding optimal policy in Π that achieves $V^*(x)$ for all $x \in X$ will be denoted by π^*.

Sometimes we will describe an MDP using a *simulation model* or *noise-driven model*, denoted by $(X, A, A(\cdot), f, R')$, where $R'(x, a, w) \leq R_{\max}, x \in X, a \in A(x), w \in [0,1]$, is the non-negative reward, and f is the next-state transition function such that the system dynamics are given by

$$x_{t+1} = f(x_t, a_t, w_t) \text{ for } t = 0, 1, ..., H-1, \tag{3}$$

where $x_t \in X$ is the state at time t, $a_t \in A(x)$ is the action at time t, and w_t is a random disturbance uniformly and independently selected from [0,1] at time t, representing the uncertainty in the system. Note that any simulation model $(X, A, A(\cdot), f, R')$ with dynamics (3) can be transformed into a model $(X, A, A(\cdot), P, R)$ with state-transition function P. Conversely, given a standard MDP model $(X, A, A(\cdot), P, R)$, it can be represented as a simulation model $(X, A, A(\cdot), f, R')$ [6]; [14], Sec. 2.3. It will be assumed that the same random number is associated with the reward and next state functions in the simulation model.

2.1. Optimality Equations. For the finite-horizon problem ($H < \infty$), we define the *optimal reward-to-go value* for state $x \in X$ in stage i by

$$V_i^*(x) = \sup_{\pi \in \Pi} V_i^{\pi}(x), \tag{4}$$

where the *reward-to-go value for policy* π for state x in stage i is defined by

$$V_i^\pi(x) = E\left[\sum_{t=i}^{H-1} \gamma^t R(x_t, \pi_t(x_t)) \middle| x_i = x\right], \tag{5}$$

$i = 0, ..., H-1$, with $V_H^*(x) = 0$ for all $x \in X$. Note that $V^\pi(x) = V_0^\pi(x)$ and $V^*(x) = V_0^*(x)$, and we will refer to V^π and V^* as the *value function for* π and the *optimal value function*, respectively. It is well known that V_i^* can be written recursively as follows: for all $x \in X$ and $i = 0, ..., H-1$,

$$V_i^*(x) = \sup_{a \in A(x)} Q_i^*(x, a), \tag{6}$$

$$Q_i^*(x, a) = R(x, a) + \gamma \sum_{y \in X} P(x, a)(y) V_{i+1}^*(y). \tag{7}$$

The solution of these optimality equations is usually referred to as (stochastic) dynamic programming, for which we have

$$V_0^*(x_0) = \sup_{\pi \in \Pi} V_0^\pi(x_0).$$

For an infinite horizon problem ($H = \infty$), we consider the set $\Pi_s \subseteq \Pi$ of all stationary Markovian policies such that $\Pi_s = \{\pi | \pi \in \Pi \text{ and } \pi_t = \pi_{t'} \text{ for all } t, t'\}$, since under mild regularity conditions, an optimal policy always exists in Π_s for the infinite horizon problem. In a slight abuse of notation, we use π for the policy $\{\pi, \pi, \ldots, \}$ for the infinite horizon problem, and we define the *optimal value* associated with an initial state $x \in X$: $V^*(x) = \sup_{\pi \in \Pi} V^\pi(x), x \in X$, where for $x \in X, 0 < \gamma < 1, \pi \in \Pi$,

$$V^\pi(x) = E\left[\sum_{t=0}^{\infty} \gamma^t R(x_t, \pi(x_t)) \middle| x_0 = x\right],$$

for which the well-known Bellman's optimality principle holds as follows: For all $x \in X$,

$$V^*(x) = \sup_{a \in A(x)} \left\{ R(x, a) + \gamma \sum_{y \in X} P(x, a)(y) V^*(y) \right\}, \tag{8}$$

where $V^*(x)$, $x \in X$, is unique, and there exists a policy π^* satisfying

$$\pi^*(x) \in \arg\sup_{a \in A(x)} \left\{ R(x, a) + \gamma \sum_{y \in X} P(x, a)(y) V^*(y) \right\}, x \in X, \tag{9}$$

and $V^{\pi^*}(x) = V^*(x)$ for all $x \in X$.

In order to simplify the notation, we use V^* and V^π to denote the optimal value function and value function for policy π, respectively, in both the finite and infinite horizon settings.

Define

$$Q^*(x,a) = R(x,a) + \gamma \sum_{y \in X} P(x,a)(y) V^*(y), \quad x \in X, a \in A(x). \tag{10}$$

Then it immediately follows that $\sup_{a \in A(x)} Q^*(x,a) = V^*(x), x \in X$, and that Q^* satisfies the following fixed-point equation: for $x \in X$ and $a \in A(x)$,

$$Q^*(x,a) = R(x,a) + \gamma \sum_{y \in X} P(x,a)(y) \sup_{a' \in A(y)} Q^*(y,a'). \tag{11}$$

Our goal for the infinite horizon problem is to find an (approximate) optimal policy $\pi^* \in \Pi_s$ that achieves the (approximate) optimal value with an initial state, where the initial state is distributed with a probability distribution δ defined over X.

For a simulation model $(X, A, A(\cdot), f, R')$ with dynamics (3), the optimal reward-to-go value function V_i^* in stage i over a horizon H is given by

$$V_i^*(x) = \sup_{\pi \in \Pi} E_w \left[\sum_{t=i}^{H-1} R(x_t, \pi_t(x_t), w_t) \middle| x_i = x \right],$$

where $x \in X, w = (w_i, w_{i+1}, ..., w_{H-1}), w_j \sim U(0,1), j = i, ..., H-1$, and $x_t = f(x_{t-1}, \pi_{t-1}(x_{t-1}), w_{t-1})$ is a random variable denoting the state at stage t following policy π. V^* is also similarly given for the MDP model, and the Q-function can be defined analogously:

$$Q_i^*(x,a) = E[R(x,a,U)] + \gamma E[V_{i+1}^*(f(x,a,U))], U \sim U(0,1). \tag{12}$$

2.2. Previous Work on Computational Methods. While an optimal policy can, in principle, be obtained by the methods of dynamic programming, policy iteration, and value iteration [4] [25] such computations are often prohibitively time-consuming. In particular, the size of the state space grows exponentially with the number of *state variables*, a phenomenon referred to by Bellman as the *curse of dimensionality.* Similarly, the size of the action space can also lead to computational intractability. Lastly, the transition function/probabilities (f or P) and/or random rewards may not be explicitly known, but a simulation model may be available for producing sample paths, which means that traditional approaches cannot be applied. These diverse computational challenges have given rise to a number of approaches intended to result in more tractable computations for estimating the optimal value function and finding optimal or good suboptimal policies. Some of these approaches can be categorized as follows:

1. structural analysis and proof of structural properties,
2. approximating the problem with a simpler problem,
3. approximating the dynamic programming equations or the value function,

4. algorithms in policy space,

The first approach can be exact, and involves the use of structural properties of the problem or the solution, such as monotonicity, convexity, modularity, or factored representations, to facilitate the process of finding an optimal solution or policy (cf. [21], [28], [12], [13]).

The remaining approaches all involve approximations or suboptimal policies. The second class of approaches can involve (i) approximation of the model with a simpler model (e.g., via state aggregation, linearization, or discretization), or (ii) restricting the structure of the policies (e.g., linear policies, certainty equivalent policies, or open-loop-feedback control policies [4]. The third approach is to approximate the value function and/or the dynamic programming equations using techniques such as state aggregation, basis function representations, and feature extraction [5] [14]. The fourth class of algorithms work in policy space, as does Policy Iteration, and are intended to provide more tractable algorithms than PI. The algorithms presented in this paper use randomization, sampling, or simulation in the context of the third and fourth approaches listed above.

To put our work in better perspective, we now briefly compare our approaches with some other important randomized/simulation-based methods. Most of this work has involved approximate solution of the dynamic programming equations or approximation of value functions, and is referred to as *reinforcement learning* or *neuro-dynamic programming* (NDP) (cf. [5], [29], [31], [27], [22]).

Q-learning, perhaps the most well-known example of reinforcement learning, is a stochastic-approximation based solution approach to solving (11) [32] [30]. It is a *model-free* algorithm that works for the case in which the parameters of P and R are unknown. In asynchronous Q-learning, a sequence of estimates $\{Q_t(\cdot,\cdot), t = 0, 1, ..., \}$ of Q^* is constructed as follows. The decision maker observes state x_t and takes an action $a_t \in A(x_t)$. The action a_t is chosen according to a randomized policy (a randomized policy is a generalized type of policy, in which, for an observed state x_t, an action is chosen randomly from a probability distribution over $A(x_t)$). The decision maker receives the reward $R(x_t, a_t)$, updates $Q_t(x_t, a_t)$ by

$$\begin{aligned} Q_{t+1}(x_t, a_t) \leftarrow Q_t(x_t, a_t) + \alpha_t(x_t, a_t) \times \\ \left[R(x_t, a_t) + \gamma \sup_{a' \in A(N(x_t, a_t))} Q_t(N(x_t, a_t), a') - Q_t(x_t, a_t) \right], \end{aligned}$$

where $\alpha_t(x_t, a_t)$ is a nonnegative stepsize coefficient, and sets $Q_{t+1}(x, a) = Q_t(x, a)$ for all pairs $(x, a) \neq (x_t, a_t)$. Here, $N(x_t, a_t)$ is a simulated next state that is equal to y with $P(x_t, a_t)(y)$. The values of $R(x_t, a_t)$ and $N(x_t, a_t)$ are observed by simulation.

It has been shown that under general conditions, the sequence $\{Q_t\}$ converges to the function Q^* for finite state and action MDPs [30]. The critical condition for the convergence is that the randomized policy employed by the decision maker needs to ensure that each state is visited infinitely often and every action is taken (explored) in every state infinitely often. Only limited results exist for the convergence rate of Q-learning, and it is well-known that the convergence of stochastic-approximation-based algorithms for solving MDPs can be quite slow. Furthermore, because straightforward Q-learning is implemented with a lookup table of size $|X| \times |A|$, it suffers from the curse of dimensionality.

Another important aspect of the work involves approximating the optimal value function V^* using neural networks and/or simulation. $V^*(x), x \in X$, is replaced with a suitable function-approximation $\tilde{V}(x, r)$, called a "scoring function," where r is a vector of parameters, and an approximate optimal policy is obtained by taking an action in

$$\arg\max_{a \in A(x)} E\left[R(x,a) + \gamma \tilde{V}(N(x,a), r)\right]$$

at state x. The general form of $\tilde{V}$ is known and is such that the evaluation of $\tilde{V}(x, r)$ is simple once the vector r is determined. By the use of scoring functions which involve few parameters, we can compactly represent a large state space. For example, $\tilde{V}(x, r)$ may be the output of some neural network in response to the input x, and r is the associated vector of weights or parameters of the neural network. Or we can select features/basis-functions to represent states, and r in this case is the associated vector of relative weights of the features/basis-functions. Once the architecture of scoring functions are determined, NDP then needs to "learn" the right parameter vector value that closely approximates the optimal value. The success of the approach depends heavily on the choice of a good architecture, which is generally problem dependent. Furthermore, the quality of the approximation is usually often difficult to gauge in terms of useful theoretical bounds on the error from optimality.

Up to now, the majority of the solution methods have concentrated on reducing the size of the state space in order to address the state space "curse of dimensionality." The key idea throughout is to avoid enumerating the entire state space. However, most of the above approaches generally require the ability to search the entire action space in order to choose the best action at each step of the iteration procedure; thus problems with very large action spaces may still pose a computational challenge. The approach proposed in Section 4 is meant to complement these highly successful techniques. In particular, there we focus on MDPs where the state space is relatively small but the action space is very large, so that enumerating the entire action space becomes practically inefficient. From a more general point of view, if one of the aforementioned state space reduction techniques is considered, for instance, say state aggregation, then MDPs with small state spaces and large action spaces can also be regarded as the outcomes resulting from the aggregation of MDPs with large state and action spaces.

3. Multi-stage Adaptive Sampling Algorithms. In this section, we present algorithms intended to accurately and efficiently estimate the optimal value function under the constraint that there is a finite number of simulation replications to be allocated per state in stage i. The straightforward approach to this would be simply to sample each action feasible in a state equally, but this is clearly not an efficient use of computational resources, so the main question to be decided is which action to sample next. The algorithms in this section adaptively choose which action to sample as the sampling process proceeds, based on the estimates obtained up to that point, and lead to value function estimators that converge to the true value asymptotically in the total number of samples. These algorithms are targeted at MDPs with large, possibly *uncountable*, state spaces and relatively smaller *finite* action spaces. The primary setting in this chapter will be *finite horizon* models, which lead to a recursive structure, but the algorithms can also be modified for infinite horizon problems. For example, once we have an algorithm that estimates the optimal value/policy for finite horizon problems, we can create a nonstationary randomized policy in an on-line manner in the context of "planning" or receding horizon control for solving infinite horizon problems.

3.1. Upper Confidence Bound Sampling. The first algorithm, called the upper confidence bound (UCB) sampling algorithm, is based on the expected *regret* analysis for multi-armed bandit problems, in which the sampling is done based on upper confidence bounds generated by simulation-based estimates. We have developed several estimators based on the algorithm, and established the asymptotic unbiasedness of these estimators [8]. For one of the estimators, the convergence proof also leads to a characterization of the rate of convergence; specifically, we show that an upper bound for the bias converges to zero at rate $O\left(\sum_{i=0}^{H-1} \frac{\ln N_i}{N_i}\right)$, where N_i is the total number of samples that are used per state sampled in stage i; furthermore, the logarithmic bound in the numerator is achievable uniformly over time. The running time-complexity of the algorithm is at worst $O\left((|A| \max_{i=0,\dots,H-1} N_i)^H\right)$, which is independent of the state space size, but depends on the size of the action space, because the algorithm requires that each action be sampled at least once for each sampled state.

The goal of the multi-armed bandit problem is to play as often as possible the machine that yields the highest (expected) reward (cf. [19], [1], [2]). The regret quantifies the exploration/exploitation dilemma in the search for the true "optimal" machine, which is unknown in advance. During the search process, we wish to explore the reward distribution of different machines while also frequently playing the machine that is empirically best thus far. The regret is the expected loss due to not always playing the true optimal machine. For an optimal strategy the regret grows at least logarithmically in the number of machine plays, and the logarithmic regret is also achievable uniformly over time with a simple and efficient sampling algorithm for arbitrary reward distributions with bounded support. We incorporate these results into a sampling-based process for finding an optimal action in a state for a single stage of an MDP by appropriately converting the definition of regret into the difference between the true optimal value and the approximate value yielded by the sampling process. We then extend the one-stage sampling process into multiple stages in a recursive manner, leading to a multi-stage (sampling-based) approximation algorithm for solving MDPs.

3.1.1. Algorithm Description and Convergence. Suppose we estimate $Q_i^*(x,a)$ by a sample mean $\hat{Q}_i(x,a)$ for each action $a \in A(x)$, where

$$\hat{Q}_i(x,a) = \frac{1}{N_a^i(x)} \sum_{j=1}^{N_a^i(x)} R'(x,a,w_j^a) + \gamma \hat{V}_{i+1}^{N_{i+1}}(f(x,a,w_j^a)), \tag{13}$$

where $N_a^i(x)$ is the number of times action a has been sampled from state x in stage i ($\sum_{a\in A(x)} N_a^i(x) = N_i$), the sequence $\{w_j^a, j = 1, ..., N_a^i(x)\}$ is the corresponding random numbers to generate the next states. Note that the number of next state samples depends on the state x, action a, and stage i. Suppose also that we estimate the optimal value of $V_i^*(x)$ by

$$\hat{V}_i^{N_i}(x) = \sum_{a\in A(x)} \frac{N_{a,i}^x}{N_i} \hat{Q}_i(x,a).$$

This leads to the following recursion:

$$\hat{V}_i^{N_i}(x) := \sum_{a\in A(x)} \frac{N_a^i(x)}{N_i} \left(\frac{1}{N_a^i(x)} \sum_{j=1}^{N_a^i(x)} R'(x,a,w_j^a) + \gamma \hat{V}_{i+1}^{N_{i+1}}(f(x,a,w_j^a)) \right),$$

$i = 0, ..., H-1$, with $\hat{V}_H^{N_H}(x) = 0$ for all $x \in X$ and any $N_H > 0$.

In the above definition, the total number of sampled (next) states is $O(N^H)$ with $N = \max_{i=0,...,H-1} N_i$, which is independent of the state space size. One approach is to select "optimal" values of $N_a^i(x')$ for $i = 0, ..., H-1$, $a \in A(x')$, and $x' \in X$, such that the expected error between the values of $\hat{V}_0^{N_0}(x)$ and $V_0^*(x)$ is minimized, but this problem would be difficult to solve. So instead we seek the values of $N_a^i(x')$ for $i = 0, ..., H-1$, $a \in A(x')$, and $x' \in X$ such that the expected difference is *bounded* as a function of $N_a^i(x')$ and N_i, $i = 0, ..., H-1$, and that the bound (from above and from below) goes to zero as N_i, $i = 0, ..., H-1$, go to infinity. We present an allocation rule (sampling algorithm) that adaptively chooses which action to sample, updating the value of $N_a^i(x')$ as the sampling process proceeds, such that the value function estimator is asymptotically unbiased, i.e., $E[\hat{V}_0^{N_0}(x)] \to V_0^*(x)$, as $N_i \to \infty, \forall\, i = 0, ..., H-1$, and an upper bound on the bias converges to zero at rate $O(\sum_i \frac{\ln N_i}{N_i})$, where the logarithmic bound in the numerator is achievable uniformly over time.

As mentioned before, the main idea behind the adaptive allocation rule is based on a simple interpretation of the regret analysis of the multi-armed bandit problem, a well-known model that captures the exploitation/exploration trade-off. An M-armed bandit problem is defined by random variables $\eta_{i,n}$ for $1 \leq i \leq M$ and $n \geq 1$, where successive plays of machine i yield "rewards" $\eta_{i,1}, \eta_{i,2}, ...$, which are independent and identically distributed according to an unknown but fixed distribution η_i with unknown expectation μ_i. The rewards across machines are also independently generated. Let $T_i(n)$ be the number of times machine i has been played by an algorithm during the first n plays. Define the *expected regret* $\rho(n)$ of an algorithm after n plays by

$$\rho(n) = \mu^* n - \sum_{i=1}^{M} \mu_i E[T_i(n)] \text{ where } \mu^* := \max_i \mu_i.$$

Any algorithm that attempts to minimize this expected regret must play a best machine (one that achieves μ^*) exponentially (asymptotically) more often than the other machines, leading to $\rho(n) = \Theta(\ln n)$. One way to achieve the asymptotic logarithmic regret is to use upper confidence bounds, which capture the trade off between exploitation – choosing the machine with the current highest sample mean – and exploration – trying other machines that might have higher actual means. This leads to an easily implementable algorithm in which the machine with the current highest upper confidence bound is chosen.

For an intuitive description of the allocation rule, consider first only the one-stage approximation. That is, we assume for now that $V_1^*(x)$-value for each sampled state $x \in X$ is known. To estimate $V_0^*(x)$, obviously we need to estimate $Q_0^*(x, a^*)$, where $a^* \in \arg\max_{a \in A(x)}(Q_0^*(x, a))$. The search for a^* corresponds to the search for the best machine in the multi-armed bandit problem. We start by sampling a random number $w^a \sim U(0, 1)$ for each possible action once at x, which leads to the next (sampled) state $f(x, a, w^a)$ according to f and reward $R'(x, a, w^a)$. We then iterate as follows (see **Loop** in Figure 1). The next action to sample is the one that achieves the maximum among the current estimates of $Q_0^*(x, a)$ plus its current upper confidence bound (see Equation (15)), where the estimate $\hat{Q}_0(x, a)$ is given by the *sample mean* of the immediate reward plus V_1^*-values (multiplied by the discount factor) at the *sampled next states that have been sampled so far* (see Equation (16)).

Among the N_0 samples for state x, $N_a^0(x)$ denotes the number of samples using action a. If the sampling is done appropriately, we might expect that $N_a^0(x)/N_0$ provides a good estimate of the likelihood that action a is optimal in state x, because in the limit as $N_0 \to \infty$, the sampling scheme should lead to $N_{a^*}^0(x)/N_0 \to 1$ if a^* is the unique optimal action, or if there are multiple optimal actions, say a set A^*, then $\sum_{a\in A^*} N_a^0(x)/N_0 \to 1$, i.e., $\{N_a^0(x)/N_0\}_{a\in A(x)}$ should converge to a probability distribution concentrated on the set of optimal actions. For this reason, we use a weighted (by $N_a^0(x)/N_0$) sum of the currently estimated value of $Q_0^*(x,a)$ over $A(x)$ to approximate $V_0^*(x)$ (see Equation (17)). Ensuring that the weighted sum concentrates on a^* as the sampling proceeds will ensure that in the limit the estimate of $V_0^*(x)$ converges to $V_0^*(x)$.

Figure 1 presents a high-level description of the upper confidence bound (UCB) adaptive sampling algorithm for estimating $V_0^*(x)$ for a given state x. The inputs to the algorithm are a state $x \in X$, $N_i \geq \max_{x\in X} |A(x)|$, and stage i, and the output is $\hat{V}_i^{N_i}(x)$, the estimate of $V_i^*(x)$. Whenever we encounter $\hat{V}_k^{N_k}(y)$ for a state $y \in X$ and stage k in the **Initialization** and **Loop** portions of the algorithm, we need to call the algorithm recursively (at Equation (14) and (16)). The initial call to the algorithm is done with $i = 0$, the initial state x_0, and N_0, and every sampling is done independently of the previously samplings. To help understand how the recursive calls are made sequentially, in Figure 2, we graphically illustrate the sequence of calls with two actions and $H = 3$ for the **Initialization** portion.

The running time complexity of the UCB sampling algorithm is $O((|A|N)^H)$ with $N = \max_i N_i$. To see this, let M_i be the number of recursive calls made to compute $\hat{V}_i^{N_i}$ in the *worst* case. At stage i, the algorithm makes at most $M_i = |A|N_iM_{i+1}$ recursive calls (in **Initialization** and **Loop**), leading to $M_0 = O((|A|N)^H)$. In contrast, backward induction has $O(H|A||X|^2)$ running time-complexity. Therefore, the main benefit of the UCB sampling algorithm is independence from the state space size, but this comes at the expense of exponential (versus linear, for backwards induction) dependence on both the action space and the horizon length.

The main convergence theorem for the UCB sampling algorithm is as follows [8].

THEOREM 3.1. *Assume that $|A(x)| > 1$ for all $x \in X$. Suppose the UCB sampling algorithm is run with the input N_i for stage $i = 0, ..., H-1$ and an arbitrary initial state $x \in X$. Then*

(1)

$$\lim_{N_0\to\infty} \lim_{N_1\to\infty} \cdots \lim_{N_{H-1}\to\infty} E[\hat{V}_0^{N_0}(x)] = V_0^*(x).$$

(2) Moreover, the bias induced by the algorithm is bounded by a quantity that

Upper Confidence Bound (UCB) Sampling Algorithm

Input: a state $x \in X$, $N_i \geq \max_{x\in X} |A(x)|$, and stage i. **Output:** $\hat{V}_i^{N_i}(x)$.

Initialization: Sample a random number $w^a \sim U(0,1)$ for each action $a \in A(x)$ and set

$$\hat{Q}_i(x,a) = \begin{cases} 0 \text{ if } i = H \text{ and go to } \textbf{Exit} \\ R'(x,a,w^a) + \gamma \hat{V}_{i+1}^{N_{i+1}}(f(x,a,w^a)) \text{ if } i \neq H \end{cases} \tag{14}$$

and set $\bar{n} = |A(x)|$.

Loop: Sample $w^{\tilde{a}^*} \sim U(0,1)$ for an action $\tilde{a}^*$, an estimate of the optimal action a^*, that achieves

$$\max_{a\in A(x)} \left(\hat{Q}_i(x,a) + R_{\max}(H-i)\sqrt{\frac{2\ln \bar{n}}{N_a^i(x)}} \right), \tag{15}$$

where $N_a^i(x)$ is the number of random number samples for action a that has been sampled so far, $\bar{n}$ is the overall number of samples so far for this stage,
and $\hat{Q}_i$ is defined by

$$\hat{Q}_i(x,a) = \frac{1}{N_a^i(x)} \sum_{j=1}^{N_a^i(x)} R'(x,a,w_j^a) + \gamma \hat{V}_{i+1}^{N_{i+1}}(f(x,a,w_j^a)), \tag{16}$$

where $\{w_j^a\}$, $j = 1, ..., N_a^i(x)$ refers to the sampled random number sequence thus far for the sampled execution of the action a.

- Update $N_{\tilde{a}^*}^i(x) \leftarrow N_{\tilde{a}^*}^i(x) + 1$.
- Update $\hat{Q}_i(x,\tilde{a}^*)$ with the $\hat{V}_{i+1}^{N_{i+1}}(f(x,\tilde{a}^*,w^{\tilde{a}^*}))$ value.
- $\bar{n} \leftarrow \bar{n} + 1$. If $\bar{n} = N_i$, then exit **Loop**.

Exit: Set $\hat{V}_i^{N_i}(x)$ such that

$$\hat{V}_i^{N_i}(x) = \begin{cases} \sum_{a\in A(x)} \frac{N_a^i(x)}{N_i} \hat{Q}_i(x,a) \text{ if } i = 0, ..., H-1 \\ 0 \text{ if } i = H. \end{cases} \tag{17}$$

and return $\hat{V}_i^{N_i}(x)$.

FIG. 1. *Upper Confidence Bound (UCB) sampling algorithm description*

converges to zero at rate $O\big(\sum_{i=0}^{H-1} \frac{\ln N_i}{N_i}\big)$, *i.e.,*

$$V_0^*(x) - E[\hat{V}_0^{N_0}(x)] \leq O\Big(\sum_{i=0}^{H-1} \frac{\ln N_i}{N_i}\Big), \ x \in X.$$

3.1.2. Two Additional Estimators. In addition to the original estimator given by Equation (17), two alternative estimators are considered in [8]. First, consider the estimator that replaces the weighted sum of the Q-value estimates in Equation (17) by the maximum of the estimates, i.e., for $i < H$,

$$\hat{V}_i^{N_i}(x) = \max_{a\in A(x)} \hat{Q}_i(x,a), \tag{18}$$

$$\hat{Q}_i(x,a) = \frac{1}{N_a^i(x)} \sum_{j=1}^{N_a^i(x)} R'(x,a,w_j^a) + \gamma \hat{V}_{i+1}(f(x,a,w_j^a)).$$

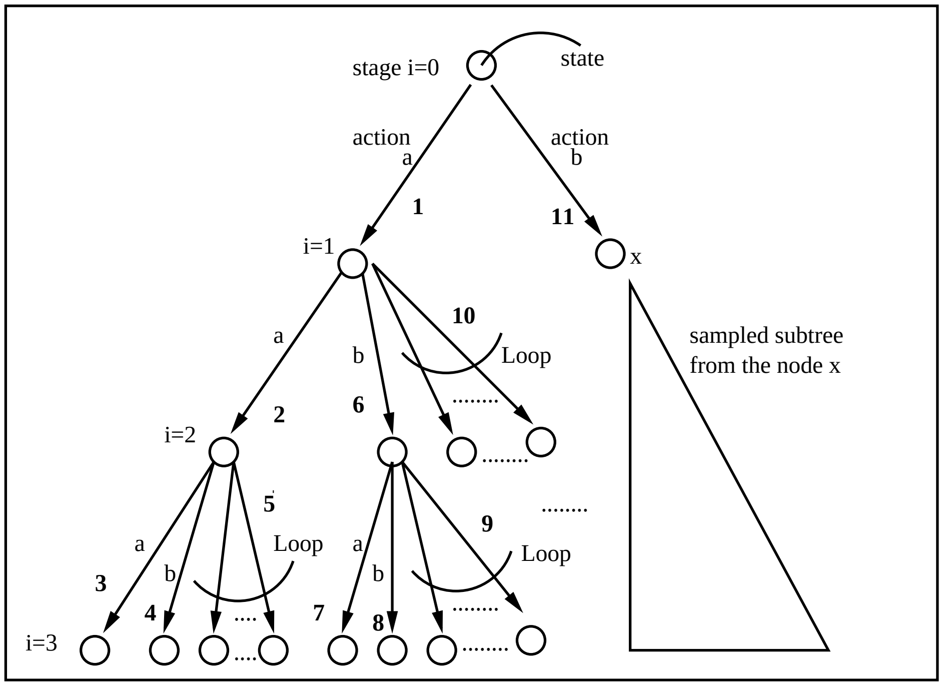

FIG. 2. *Graphical illustration of the sequence of recursive calls made in* ***Initialization*** *of the UCB sampling algorithm. Each circle corresponds to a state and each arrow with noted action signifies a sampling of a random number for the action (and a recursive call). The bold-face number near each arrow is the sequence number for the recursive calls made. For simplicity, the entire* ***Loop*** *process is signified by one call number.*

For the nonadaptive case, it can be shown that this estimator is also asymptotically unbiased, but with a finite-sample "optimistic" bias in the opposite direction as the original estimator (i.e., upwards for maximization problems and downwards for minimization problems such as the inventory control problem).

Next, consider an estimator that chooses the action that has been sampled the most thus far in order to estimate the value function. It can be easily shown that this estimator is less optimistic than the previous alternative, and so we combine it with the original estimator to obtain the following alternative estimator:

$$\bar{V}_i(x) = \max\left\{ \bar{Q}_i(x, a^*), \sum_{a \in A(x)} \frac{N_a^i(x)}{N_i} \bar{Q}_i(x,a) \right\}, \tag{19}$$

$$\text{where } a^* = \arg\max_a \{N_a^i(x)\},$$

$$\bar{Q}_i(x,a) = \frac{1}{N_a^i(x)} \sum_{j=1}^{N_a^i(x)} R'(x, a, w_j^a) + \gamma \bar{V}_{i+1}(f(x, a, w_j^a)).$$

Intuitively, the rationale behind combining via the max operator, instead of just the straightforward $\bar{V}_i(x) = \bar{Q}_i(x, a^*)$, is that the estimator would be choosing the best between two possible estimates of the Q-function. Actually, a similar combination could also be used for Equation (18), as well, to derive yet another possible estimator.

It is conjectured that both of these two alternatives are asymptotically unbiased, with the estimator given by Equation (18) having an "optimistic" bias (i.e., high for maximization problems, low for minimization problems). If so, valid, albeit conservative, confidence intervals for the optimal value could also be easily derived by combining the two oppositely biased estimators. Such a result can be established for the nonadaptive versions of the estimators, but proving these results in our setting and characterizing the convergence rate of the estimator given by Equation (18) in a similar manner as for the original estimator are ongoing research problems.

Numerical results for this algorithms with all three estimators are presented in [8].

3.2. Pursuit Learning Automata Sampling. The second algorithm in the section is the pursuit learning automata (PLA) sampling algorithm. In [10], we present and analyze the finite-time behavior of the PLA sampling algorithm, providing a bound on the probability that a given initial state takes the optimal action, and a bound on the probability that the difference between the optimal value and the estimate of it exceeds a given error.

Similar to the UCB algorithm, the PLA sampling algorithm constructs a sampled tree in a recursive manner to estimate the optimal value at an initial state and incorporates an adaptive sampling mechanism for selecting which action to sample at each branch in the tree. However, whereas the sampling in the UCB algorithm is based on the exploration-exploitation tradeoff captured by a multi-armed bandit model, the PLA sampling algorithm's sampling is based on a probability estimate for the optimal action. The analysis of the UCB sampling algorithm is given in terms of the expected bias, whereas for the PLA sampling algorithm we provide a probability bound.

The PLA sampling algorithm extends in a recursive manner (for sequential decisions) the Pursuit algorithm of Rajaraman and Sastry [26] to the context of solving MDPs. The Pursuit algorithm is based on learning automata and is designed to solve (non-sequential) stochastic optimization problems. A learning automaton [23] [24] [33] [20] is associated with a finite set of actions (candidate solutions) and updates a probability distribution over the set by iterative interaction with an environment and takes (samples) an action according to the newly updated distribution. The environment provides a certain reaction (reward) to the action taken by the automaton, where the reaction is random and the distribution is unknown to the automaton. The automaton's aim is to learn to choose the optimal action that yields the highest average reward. In the Pursuit algorithm, the automaton *pursues* the current best optimal action obtained from the current estimates of the average rewards of taking each action. Rajaraman and Sastry [26] derived bounds on the number of iterations and the parameter of the learning algorithm for a given accuracy of performance of the automaton, characterizing the finite time behavior of the automaton.

The PLA sampling algorithm's sampling process of taking an action at the sampled state is adaptive at each stage. At each sampled state at a stage, a fixed sampling budget is allocated among feasible actions as in the UCB sampling algorithm, and the budget is used with the current probability estimate for the optimal action. A sampled state corresponds to an automaton and updates certain functions (including the probability distribution over the action space) at each iteration of the algorithm.

Based on the finite-time analysis of the Pursuit algorithm, we have analyzed the finite-time behavior of the PLA sampling algorithm, providing a bound on the probability that the initial state at stage 0 takes the optimal action in terms of sampling parameters of the PLA sampling algorithm and a bound on the probability that the difference between the estimate of $V_0^*(x_0)$ and $V_0^*(x_0)$ exceeds a given error.

We first provide a high-level description of the PLA sampling algorithm to estimate $V_0^*(x_0)$ for a given initial state x_0 via this set of equations. The inputs to the PLA sampling algorithm are the stage i, a state $x \in X$, and sampling parameters $N_i > 0$, $\mu_i \in (0,1)$, and the output of the PLA algorithm is $\hat{V}_i^{N_i}(x)$, the estimate of $V_i^*(x)$, where $\hat{V}_H^{N_H}(x) = V_H^{N_H}(x) = 0 \ \ \forall N_H, x \in X$. Whenever $\hat{V}_{i'}^{N_{i'}}(y)$ (for stage $i' > i$ and state y) is encountered in the **Loop** portion of the PLA sampling algorithm, a recursive call to the PLA sampling algorithm is required (at Equation (20)). The initial call to the PLA sampling algorithm is done with stage $i = 0$, the initial state x_0, N_0, and μ_0, and every sampling is independent of previous samplings.

Basically, the PLA sampling algorithm builds a sampled tree of depth H with the root node being the initial state x_0 at stage 0 and a branching factor of N_i at each level i (level 0 corresponds to the root). The root node x_0 corresponds to an automaton and initializes the probability distribution over the action space P_{x_0} as the uniform distribution (refer to **Initialization** in the PLA sampling algorithm). The x_0-automaton then samples an action and a random number (an action and a random number together corresponding to an edge in the tree) independently with respect to the current probability distribution $P_{x_0}(k)$ and $U(0,1)$, respectively, at each iteration $k = 0, ..., N_0 - 1$. If action $a(k) \in A(x_0)$ is sampled, the count variable $N_{a(k)}^0(x_0)$ is incremented. From the sampled action $a(k)$ and the random number w_k, the automaton obtains an environment response

$$R'(x_0, a(k), w_k) + \hat{V}_1^{N_1}(f(x_0, a(k), w_k)).$$

Then by averaging the responses obtained for each action $a \in A(x_0)$ such that

```
Pursuit Learning Automata (PLA) Sampling Algorithm
    • Input: stage $i \neq H$, state $x \in X$, $N_i > 0$, $\mu_i \in (0,1)$.
      (If $i = H$, then Exit immediately, returning $V_H^{N_H}(x) = 0$.)
    • Initialization: Set $P_x(0)(a) = 1/|A(x)|, N_a^i(x) = 0, M_i^{N_i}(x,a) = 0 \;\; \forall a \in A(x); k = 0$.
    • Loop until $k = N_i$:
        – Sample $a(k) \sim P_x(k)$, $w_k \sim U(0,1)$.
        – Update $Q$-value estimate for $a = a(k)$ only:
          (20)  $M_i^{N_i}(x,a(k)) \leftarrow M_i^{N_i}(x,a(k)) + R'(x,a(k),w_k) + \hat{V}_{i+1}^{N_{i+1}}(f(x,a(k),w_k))$,
                $N_{a(k)}^i(x) \leftarrow N_{a(k)}^i(x) + 1$,
                $\hat{Q}_i^{N_i}(x,a(k)) \leftarrow \frac{M_i^{N_i}(x,a(k))}{N_{a(k)}^i(x)}$.
        – Update optimal action estimate: (ties broken arbitrarily)
          (21)  $\hat{a}^* = \arg\max_{a \in A(x)} \hat{Q}_i^{N_i}(x,a)$.
        – Update probability distribution over action space:
          (22)  $P_x(k+1)(a) \leftarrow (1-\mu_i)P_x(k)(a) + \mu_i I\{a = \hat{a}^*\} \;\; \forall a \in A(x)$,
          where $I\{\cdot\}$ denotes the indicator function.
        – $k \leftarrow k+1$.
    • Exit: Return $\hat{V}_i^{N_i}(x) = \hat{Q}_i^{N_i}(x,\hat{a}^*)$.
```

FIG. 3. *Pursuit Learning Automata (PLA) sampling algorithm description*

$N_a^0(x_0) > 0$, the x_0-automaton computes a sample mean for $Q_0^*(x_0,a)$:

$$\hat{Q}_0^{N_0}(x_0,a) = \frac{1}{N_a^0(x_0)} \sum_{j:a(j)=a} R'(x_0,a,w_j) + \hat{V}_1^{N_1}(f(x_0,a,w_j)),$$

where $\sum_{a \in A(x_0)} N_a^0(x_0) = N_0$. At each iteration k, the x_0-automaton obtains an estimate of the optimal action by taking the action that achieves the current best Q-value (cf. Equation (21)) and updates the probability distribution $P_{x_0}(k)$ in the direction of the current estimate of the optimal action $\hat{a}^*$ (cf. Equation (22)). In other words, the automaton *pursues* the current best action, and hence the nonrecursive one-stage version of this algorithm is called the Pursuit algorithm [26]. After N_0 iterations, the PLA sampling algorithm estimates the optimal value $V_0^*(x_0)$ by $\hat{V}_0^{N_0}(x_0) = \hat{Q}_0^{N_0}(x_0,\hat{a}^*)$.

The overall estimate procedure is replicated recursively at those sampled states (corresponding to nodes in level 1 of the tree) $f(x_0,a(k),w_k)$-automata, where the estimates returned from those states $\hat{V}_1^{N_1}(f(x_0,a(k),w_k))$ comprise the environment responses for the x_0-automaton.

The running time complexity of the PLA sampling algorithm is $O(N^H)$ with

$N = \max_i N_i$, independent of the state space size. (For some performance guarantees, the value N depends on the size of the action space; see the next section.)

3.2.1. Convergence Results. We first make an assumption *for the purpose of the analysis.* The assumption states that at each stage, the optimal action is unique at each state. In other words, the given MDP has a unique optimal policy. We will comment on this at the end of this section.

ASSUMPTION 3.1. *For all $x \in X$ and $i = 0, 1, ..., H-1$,*

$$\theta_i(x) := Q_i^*(x, a^*) - \max_{a \neq a^*} Q_i^*(x, a) > 0,$$

where $V_i^*(x) = Q_i^*(x, a^*)$.

Some notation is necessary in order to state the convergence results, which are proved in [10]. Define $\theta := \inf_{x \in X, i=0,...,H-1} \theta_i(x)$. Given $\delta_i \in (0,1), i = 0, ..., H-1$, define

$$\rho := (1-\delta_0) \prod_{i=1}^{H-1} (1-\delta_i)^{\prod_{j=1}^{i} N_j}. \tag{23}$$

Also, let

$$\lambda(\epsilon, \delta) = \left\lceil \frac{2M_{\epsilon,\delta}}{\ln l} \ln \left[\frac{l M_{\epsilon,\delta}}{\ln l} \left(\frac{2M_{\epsilon,\delta}}{\delta} \right)^{\frac{1}{M_{\epsilon,\delta}}} \right] \right\rceil \tag{24}$$

with

$$M_{\epsilon,\delta} = \max \left\{ 6, \left\lceil \frac{R_{\max} H \ln(4/\delta)}{(R_{\max} H + \epsilon) \ln((R_{\max} H + \epsilon)/R_{\max} H) - \epsilon} \right\rceil \right\}$$

and $l = 2|A(x)|/(2|A(x)|-1)$.

The convergence results for the PLA sampling algorithm are as follows.

THEOREM 3.2. *Assume that Assumption 3.1 holds. Given $\delta_i \in (0,1), i = 0, ..., H-1$, select $N_i > \lambda(\theta/2^{i+2}, \delta_i)$ and $0 < \mu_i < \mu_i^* = 1 - 2^{-1/N_i}, i = 1, ..., H-1$. If*

$$N_0 > \lambda(\theta/4, \delta_0) + \left\lceil \frac{\ln \frac{1}{\epsilon}}{\ln \frac{1}{1-\mu_0^*}} \right\rceil$$

and $0 < \mu_0 < \mu_0^* = 1 - 2^{-1/\lambda(\theta/4, \delta_0)}$, then under the PLA sampling algorithm with ρ in Equation (23), for all $\epsilon \in (0,1)$,

$$P\left(P_{x_0}(N_0)(a^*) > 1 - \epsilon\right) > \rho,$$

where $a^* \in \arg\max_{a \in A(x_0)} Q_0^*(x_0, a)$.

Theorem 3.3. *Assume that Assumption 3.1 holds. Given $\delta_i \in (0,1), i = 0,...,H-1$ and $\epsilon \in (0,\theta]$, select $N_i > \lambda(\frac{\epsilon}{2^{i+2}}, \delta_i), 0 < \mu_i < \mu_i^* = 1 - 2^{-\frac{1}{N_i}}, i = 0,...,H-1$. Then under the PLA sampling algorithm with ρ in Equation (23),*

$$P\left(\left|\hat{V}_0^{N_0}(x_0) - V_0^*(x_0)\right| > \frac{\epsilon}{2}\right) < 1 - \rho.$$

From the statements of Theorems 3.2 and 3.3, the performance of the PLA sampling algorithm depends on the value of θ. If $\theta_i(x)$ is very small or even 0 (failing to satisfy Assumption 3.1) for some $x \in X$, the PLA sampling algorithm requires a very high sampling-complexity to distinguish between the optimal action and the second best action or multiple optimal actions *if x is in the sampled tree of* the PLA sampling algorithm. In general, the larger θ is, the more effective the algorithm will be (the smaller the sampling complexity). Therefore, in the actual implementation of the PLA sampling algorithm, if multiple actions' performances are very close after "enough" iterations in **Loop**, it would be advisable to keep only one action among the competitive actions (transferring the probability mass). The parameter θ can thus be viewed as a measure of problem difficulty.

Furthermore, to achieve a certain approximation guarantee at the root level of the sampled tree (i.e., the quality of $\hat{V}_0^{N_0}(x_0)$), we need a *geometric* increase in the accuracies of the optimal reward-to-go values for the sampled states at the lower levels, making it necessary that the total number of samples at the lower levels increase geometrically (N_i depends on $2^{i+2}/\theta$). This is because the estimate error of $V_i^*(x_i)$ for some $x_i \in X$ affects the estimate of the sampled states in the higher levels in a recursive manner (the error in a level "adds up recursively").

However, the probability bounds in Theorem 3.2 and 3.3 are obtained with coarse estimation of various parameters/terms. For example, we used the worst case values of $\theta_i(x), x \in X, i = 0,...,H-1$ and $(R_{\max}H)^2$ for bounding $\sup_{x\in X} V_i^*(x), i = 0,...,H-1$, and used some conservative bounds in the proofs and in relating the probability bounds for the estimates at the two adjacent levels. Considering this, the performance of the PLA sampling algorithm should probably be more effective in practice than the analysis indicates here.

The application of the PLA sampling algorithm to partially observed MDPs is also discussion in [10].

4. Population-Based Evolutionary Approaches. In this section, we present evolutionary population-based algorithms for finding optimal (stationary) policies *infinite* horizon MDPs. These algorithms are primarily intended for problems with large (possibly uncountable) action spaces where the policy *improvement* step in Policy Iteration (PI) becomes computationally prohibitive, and value iteration is also impractical. In particular, for PI, maximizing over the entire action space may require enumeration or random search methods. The computational complexity of each iteration of our algorithms is polynomial in the size of the state space, but unlike PI and VI, it is insensitive to the size of the action space, making the algorithms most suitable for problems with relatively small state spaces compared to the size of the action spaces. In the case of uncountable action spaces, our approach avoids the need for any discretization; discretization can lead to computational difficulties, either resulting in an action space that is too large or in a solution that is not accurate enough.

The approach taken by the algorithms in this section directly searches the policy space to avoid carrying out an optimization over the entire action space at each PI step, and resembles that of a standard genetic algorithm (GA), updating a *population* of policies using appropriate analogous operations for the MDP setting. One key feature of the algorithms presented here is the determination of an elite policy that is superior to the performances of all policies in the previous population. This monotonicity property ensures that the algorithms converge with probability one to a population in which the elite policy is an optimal policy.

We begin with a basic algorithm called Evolutionary Policy Iteration (EPI) that contains the main features of the population-based approach and present results for the theoretical convergence of the EPI algorithm. We then enhance the algorithm considerably to allow it to be more efficient for practical problems.

4.1. Evolutionary Policy Iteration. In this section, we present the Evolutionary Policy Iteration (EPI) algorithm (see [9] for more detail and proofs), which eliminates the operation of maximization over the entire action space in the policy improvement step by directly manipulating policies via a method called "policy switching" that generates an improved policy from a set of given policies. The computation time for generating such an improved policy is on the order of the state space size.

As with all evolutionary/GA algorithms, we define the kth generation population, $(k = 0, 1, 2, ...)$, denoted by $\Lambda(k)$, which is a set of policies in Π_s, and $n = |\Lambda(k)| \geq 2$ is the population size, which we take to be constant in each generation. Given the fixed initial state probability distribution δ defined over X, we define the *average value of* π *for* δ or *fitness value of* π: $J_\delta^\pi = \sum_{x \in X} V^\pi(x)\delta(x)$. Note that an *optimal policy* π^* satisfies for any $\pi \in \Pi_s$, $J_\delta^{\pi^*} \geq J_\delta^\pi$. We denote P_m as the mutation selection probability, P_g the global mutation probability, and P_l the local mutation probability. For each state $x \in X$, we also define *action selection distribution* $\mathcal{P}_x$ as a probability distribution over $A(x)$ such that $\sum_{a \in A(x)} \mathcal{P}_x(a) = 1$ and $\mathcal{P}_x(a) > 0$ for all $a \in A(x)$. The mutation probability determines whether or not $\pi(x)$ is changed (mutated) for

```
Evolutionary Policy Iteration (EPI)
        Initialization:
        Select population size $n$ and $K > 0$. $\Lambda(0) = \{\pi_1, ..., \pi_n\}$, where $\pi_i \in \Pi_s$.
        Set $N = k = 0$, $P_m, P_g, P_l \in (0, 1]$, and $\pi^*(-1) = \pi_1$.
        Repeat:
        Policy Switching:
          – Obtain $V^\pi$ for each $\pi \in \Lambda(k)$.
          – Generate the elite policy of $\Lambda(k)$ defined as
                $\pi^*(k)(x) \in \{\arg\max_{\pi \in \Lambda(k)}(V^\pi(x))(x)\}, x \in X.$
          – Stopping Rule:
              * If $J_\delta^{\pi^*(k)} \neq J_\delta^{\pi^*(k-1)}$, $N = 0$.
              * If $J_\delta^{\pi^*(k)} = J_\delta^{\pi^*(k-1)}$ and $N = K$, terminate EPI.
              * If $J_\delta^{\pi^*(k)} = J_\delta^{\pi^*(k-1)}$ and $N < K$, $N \leftarrow N + 1$.
          – Generate $n - 1$ random subsets $S_i, i = 1, ..., n - 1$ of $\Lambda(k)$
            by selecting $m \in \{2, ..., n - 1\}$ with equal probability
            and selecting $m$ policies in $\Lambda(k)$ with equal probability.
          – Generate $n - 1$ policies $\pi(S_i)$ defined as:
                $\pi(S_i)(x) \in \{\arg\max_{\pi \in S_i}(V^\pi(x))(x)\}, x \in X.$
        Policy Mutation: For each policy $\pi(S_i), i = 1, ..., n - 1$,
          – Generate a "globally" mutated policy $\pi^m(S_i)$ with $P_m$ using $P_g$ and $\mathcal{P}_x$,
            or a "locally" mutated policy $\pi^m(S_i)$ with $1 - P_m$ using $P_l$ and $\mathcal{P}_x$.
        Population Generation:
          – $\Lambda(k + 1) = \{\pi^*(k), \pi^m(S_i)\}$, $i = 1, ..., n - 1$.
          – $k \leftarrow k + 1$.
```

FIG. 4. *Evolutionary Policy Iteration (EPI)*

each state x, and the action selection distribution $\mathcal{P}_x$ is used to change $\pi(x)$ (see Section 4.1.2 for details). A high-level description of EPI is shown in Figure 4, where some steps (e.g., mutation) are described at a conceptual level, with details provided in the following subsections.

The EPI algorithm's convergence is independent of the initial population $\Lambda(0)$ mainly due to the **Policy Mutation** step. We can randomly generate an initial population or start with a set of heuristic policies. A simple example initialization is to set $\Lambda(0)$ such that for each policy in $\Lambda(0)$, the same action is prescribed for all states, but each policy in the initial population prescribes a different action.

4.1.1. Policy Switching. One of the basic procedural steps in GA is to select members from the current population to create a "mating pool" to which "crossover" is applied; this step is called "parent selection". Similarly, we can design a "policy selection" step to create a mating pool; there are many ways of doing this. The

Policy Switching step includes this selection step implicitly.

Given a nonempty subset Δ of Π_s, we define a policy $\pi_{\rm ps}$ generated by *policy switching* with respect to Δ as

$$\pi_{\rm ps}(x) \in \{\arg\max_{\pi\in\Delta}(V^\pi(x))(x)\}, x \in X. \tag{25}$$

It has been shown that the policy generated by policy switching improves any policy in Δ (Theorem 3 in [7]):

THEOREM 4.1. *Consider a nonempty subset Δ of Π_s and the policy $\pi_{\rm ps}$ generated by policy switching with respect to Δ given in (25). Then, for all $x \in X$, $V^{\pi_{\rm ps}}(x) \geq \max_{\pi\in\Delta} V^\pi(x)$.*

This theorem immediately can be used to obtain the following result relevant for the EPI algorithm.

COROLLARY 4.1. *Consider a nonempty subset Δ of Π_s and the policy $\pi_{\rm ps}$ generated by policy switching with respect to Δ given in (25). Then, for any initial state distribution δ, $J_\delta^{\pi_{\rm ps}} \geq \max_{\pi\in\Delta} J_\delta^\pi$.*

We first generate a policy $\pi^*(k)$, called the elite policy with respect to the current population $\Lambda(k)$, which improves any policy in $\Lambda(k)$ via policy switching. Thus, the new population $\Lambda(k+1)$ contains a policy that is superior to any policy in the previous population, i.e., the following monotonicity property holds:

LEMMA 4.1. *For any δ and for all $k \geq 0$, $J_\delta^{\pi^*(k)} \geq J_\delta^{\pi^*(k-1)}$.*

We then generate $n-1$ random subsets $S_i (i = 1, ..., n-1)$ of $\Lambda(k)$ as follows. We first select $m \in \{2, ..., n-1\}$ with equal probability and then select m policies from $\Lambda(k)$ with equal probability. By applying policy switching, we generate $n-1$ policies defined as

$$\pi(S_i)(x) \in \{\arg\max_{\pi\in S_i}(V^\pi(x))(x)\}, x \in X.$$

These policies will be mutated to generate a new population (see the next subsection).

Because policy switching directly manipulates policies, eliminating the operation of maximization over the entire action space, its computational time-complexity is $O(m|X|)$, where $m = |S_i|$, independent of the action space size, leading to $O(nm|X|)$ complexity in the **Policy Switching** step, and hence $O(nm|X|^3)$ overall when including the $O(|X|^2)$ complexity for the policy evaluation needed to compute V^π. On the other hand, applying a single-policy improvement step of PI directly to each policy in $\Lambda(k)$, instead of generating $\pi(S_i),\ i = 1, ..., n-1$, is of complexity $O(n|X|^2|A|)$.

4.1.2. Policy Mutation and Population Generation. Policy mutation takes a given policy, and for each state, alters the specified action probabilistically. The main reason to generate mutated policies is to avoid being caught in local optima, making a probabilistic convergence guarantee possible.

We distinguish between two types of mutation – "local" and "global" – which are differentiated by how much of the policy is likely be changed (mutated). Local mutation is intended to guide the algorithm in obtaining an exact optimal policy through local search of "nearby" policies, whereas the global mutation allows EPI to escape from local optima. A high mutation probability indicates that many components of the policy vector are likely to be mutated, representing a more global change, whereas a low mutation probability implies that very little mutation is likely to occur, meaning a more localized perturbation. For this reason, we assume that $P_l \ll P_g$, with P_l being very close to zero and P_g being very close to one. The **Policy Mutation** step first determines whether the mutation will be global or local, with probability P_m. If the policy π is globally (locally) mutated, for each state x, $\pi(x)$ is changed with probability $P_g(P_l)$. If a mutation does occur, it is carried out according to the action selection distribution $\mathcal{P}_x$, i.e., if the mutated policy is dented by π', then the new policy is generated according to $P(\pi'(x) = a) = \mathcal{P}_x(a)$, for all mutated states x (the actions for all other states remain unchanged). For example, one simple $\mathcal{P}_x$ is the uniform action selection distribution, in which case the new (mutated) policy would randomly select a new action for each mutated state x (independently) with equal probability over the set of admissible actions $A(x)$.

At the kth generation, the new population $\Lambda(k+1)$ is simply given by the elite policy generated from $\Lambda(k)$ and $n-1$ mutated policies from $\pi(S_i), i = 1, ..., n-1$. This population generation method allows a policy that is poor in terms of the performance, but might be in the neighborhood of an optimal value located at the top of the very narrow hill, to be kept in the population so that a new search region can be started from the policy. This helps EPI avoid from being caught in the region of local optima.

Once we have a new population, we need to test whether EPI should terminate. Even if the fitness values for the two consecutive elite policies are identical, this does not necessarily mean that the elite policy is an optimal policy as in PI; thus, we run the EPI algorithm K more times so that these random jumps by the mutation step will eventually bring EPI to a neighborhood of the optimum. As the value of K gets larger, the probability of being in a neighborhood of the optimum increases. Therefore, the elite policy at termination is optimal with more confidence as K increases.

4.1.3. Convergence.

THEOREM 4.2. *Given $P_m > 0$, $P_g > 0$, $P_l > 0$, and an action selection distribution $\mathcal{P}_x$ such that $\sum_{a\in A(x)} \mathcal{P}_x(a) = 1$ and $\mathcal{P}_x(a) > 0$ for all $a \in A(x)$ and all $x \in X$, as $K \to \infty$, $V^{\pi^*(k)}(x) \to V^{\pi^*}(x), x \in X$ with probability one uniformly over X, regardless of $\Lambda(0)$.*

REMARK 4.1. *In our setting, mutation of a specified action in a state is carried out using a given action selection distribution. If the action space is continuous, say*

$[0,1]$, *a straightforward implementation would only change the least significant digit for local mutation and the most significant digit for global mutation, if the numbers in* $[0,1]$ *are represented by a certain number of significant digits.*

4.1.4. Parallelization. The EPI algorithm can be naturally parallelized and by doing so, we can improve the running rate. Basically, we partition the policy space Π_s into subsets of $\{\Pi_i\}$ such that $\bigcup_i \Pi_i = \Pi_s$ and $\Pi_i \cap \Pi_j = \emptyset$ for all $i \neq j$. We then apply EPI into each Π_i in parallel and then once each part terminates, the best policy π_i^* from each part is taken. We apply then policy switching to the set of best policies $\{\pi_i^*\}$. We state a general result regarding parallelization of an algorithm that solves an MDP.

THEOREM 4.3. *Given a partition of* Π_s *such that* $\bigcup_i \Pi_i = \Pi$ *and* $\Pi_i \cap \Pi_j = \emptyset$ *for all* $i \neq j$, *consider an algorithm* $\mathcal{A}$ *that generates the best policy* π_i^* *for* Π_i *such that for all* $x \in X$, $V^{\pi_i^*}(x) \geq \max_{\pi \in \Pi_i} V^{\pi}(x)$. *Then, the policy* $\bar{\pi}$ *defined as*

$$\bar{\pi}(x) \in \{\arg\max_{\pi_i^*}(V^{\pi_i^*}(x))(x)\}, x \in X,$$

is an optimal policy for Π_s.

Note that we cannot just pick the best policy among π_i^* in terms of the fitness value J_δ^π. The condition that $J_\delta^\pi \geq J_\delta^{\pi'}$ for $\pi \neq \pi'$ does not always imply that $V^\pi(x) \geq V^{\pi'}(x)$ for all $x \in X$ even though the converse is true. In other words, we need a policy that improves all policies π_i^*. Picking the best policy among such policies does not necessarily guarantee an optimal policy for Π_s.

If the number of subsets in the partition is N, the overall convergence of the algorithm $\mathcal{A}$ is faster by a factor of N. For example, if at state x, the action a or b can be taken, let $\Pi_1 = \{\pi | \pi(x) = a, \pi \in \Pi_s\}$ and $\Pi_2 = \{\pi | \pi(x) = b, \pi \in \Pi_s\}$. By using this partition, the convergence rate of the algorithm $\mathcal{A}$ will be twice as fast.

By Theorem 4.3, this idea can be applied to PI via policy switching, yielding a "*distributed*" PI. Apply PI to each Π_i; once PI for each part terminates, combine the resulting policy for each part by policy switching. The combined policy is an optimal policy, so that this method will speed up the original PI by a factor of N if the number of subsets in the partition is N. However, this distributed variant of PI requires the maximization operation over the action space in the policy improvement step. The result of Theorem 4.3 also naturally extends to a *dynamic programming version* of PI, similarly to EPI. For example, we can partition Π_s into Π_1 and Π_2, and further partition Π_1 into Π_{11} and Π_{12}, and Π_2 into Π_{21} and Π_{22}. The optimal substructure property is preserved by policy switching. If the number of subsets generated in this way is β, then the overall computation time of an optimal policy is $O(\beta|X|C)$, where C is the maximum size of the subsets in terms of the number of policies, because policy switching is applied N times with $O(|X|)$ complexity and C is an upper bound on PI-complexity.

How do we partition the policy space so that PI or EPI converges faster? For some partitions, we can even obtain the best policies for some subsets *analytically*. However, in general, partitioning the policy space to speed up a convergence would be a difficult problem and would require a structural analysis on the problem or the policy space. Here, we briefly speculate on three possible ways of partitioning the policy space. Let N be the number of subspaces. The simplest way of partitioning the policy space is a random partition, where each policy is assigned to a particular subspace according to some probability distribution (e.g., uniformly). Another possible approach is to build a "decision tree" on the policy space based on identification of suboptimal actions. Assume that there exist a lower bound function $V^L(x)$ and an upper bound function $V^U(x)$ such that $V^L(x) \leq V^*(x) \leq V^U(x)$ for all $x \in X$. Then, if for $x \in X$ and $b \in A(x)$,

$$R(x,b) + \gamma \sum_{y \in X} P(x,b)(y) V^U(y) < V^L(x),$$

any stationary policy that uses action b in state x is nonoptimal (cf. Proposition 6.7.3 in [25]). Based on this fact, we start with a particular state $x \in X$ and identify a nonoptimal action set $\varphi(x)$ at x and build subsets $\Pi_{x,a}$ of the policy space with $a \in A(x) - \varphi(x)$. Effectively, we are building a $|A(x) - \varphi(x)|$-ary tree with x being a root. We then repeat this with another state $y \in X$ at each child $\Pi_{x,a}$ of the tree. We continue building a tree in this way until the number of children at the leaf level is N. Note that for some problems, nonoptimal actions are directly observable. For example, for a simple multiclass deadline job scheduling problem of minimizing the weighted loss, if a job's deadline is about to expire, all actions selecting a pending job from a less important class than that of the dying job are nonoptimal. A third approach is to select some features on policies, and then use the features for building the decision tree to partition the policy space.

Our discussion on the parallelization of PI and EPI can be viewed in some sense as an aggregation in the policy space, where the distributed version of PI can be used to generate an approximate solution of a given MDP.

4.2. Evolutionary Random Policy Search. We now consider a substantial enhancement of EPI called Evolutionary Random Policy Search (ERPS), introduced in [17], which proceeds iteratively by constructing and solving a sequence of sub-MDP problems defined on smaller policy spaces. At each iteration of the algorithm, two steps are fundamental: (1) The sub-MDP problem constructed in the previous iteration is approximately solved by using a variant of the policy improvement technique called policy improvement with reward swapping (PIRS), and a policy called an elite policy is generated. (2) Based on the elite policy, a group of policies is then obtained by using a "nearest neighbor" heuristic and random sampling of the entire action space, from which a new sub-MDP is created by restricting the original MDP problem (e.g., reward structure, transition probabilities) to the current available subsets of actions. Under appropriate assumptions, it is shown in [17] that the sequence of elite policies converges with probability one to an optimal policy. The theoretical convergence results include the uncountable action space case, whereas those for EPI required the action space to be finite.

Whereas EPI treats policies as the most essential elements in the action optimization step, and each "elite" policy is directly generated from a group of policies, in ERPS policies are regarded as intermediate constructions from which sub-MDP problems are then constructed and solved. Furthermore, ERPS combines global search with a local enhancement step (the "nearest neighbor" heuristic) that leads to rapid convergence once a policy is found in a small neighborhood of an optimal policy. This modification substantially improves the performance while maintaining the computational complexity at essentially the same level.

A high-level description of the ERPS algorithm is summarized in Figure 5. Detailed discussion of each of the steps follows.

4.2.1. Initialization. We start by specifying an *action selection distribution* $\mathcal{P}_x$ for each state $x \in X$, the exploitation probability $q_0 \in [0,1]$, the population size n, and a search range r_i for each state $x_i \in X$. Once chosen, these parameters are fixed throughout the algorithm. We then select an initial group of policies; however, because of the exploration step used in ERPS, the performance of the algorithm is relatively insensitive to this choice. One simple method is to choose the initial policies uniformly from the policy space Π.

The *action selection distribution* $\mathcal{P}_x$ is a probability distribution over the set of admissible actions $A(x)$, and will be used to generate sub-MDPs (see Section 4.2.3). The exploitation probability q_0 and the search range r_i will be used to construct sub-MDPs; the detailed discussion of these two parameters is deferred to later.

4.2.2. Policy Improvement with Reward Swapping. As mentioned earlier, the idea behind ERPS is to randomly split a large MDP problem into a sequence of smaller, manageable MDPs, and to extract a possibly convergent sequence of policies via solving these smaller problems. For a given policy population $\Lambda = \{\pi_1, \pi_2, \ldots, \pi_n\}$, if we restrict the original MDP (e.g., rewards, transition probabilities) to the subsets of actions $\Lambda(x) := \{\pi_1(x), \pi_2(x), \ldots, \pi_n(x)\}$ $\forall\, x \in X$, then a sub-MDP problem is induced from Λ as $\mathcal{G}_\Lambda := (X, \Gamma, \Lambda(\cdot), P, R)$, where $\Gamma := \bigcup_x \Lambda(x) \subseteq A$. Note that in general $\Lambda(x)$ is a multi-set, which means that the set may contain repeated elements;

```
Evolutionary Random Policy Search (ERPS)
  • Initialization: Specify an action selection distribution $\mathcal{P}_x$ for each $x \in X$, the population size $n > 1$, and the exploitation probability $q_0 \in [0,1]$. Specify a search range $r_i$ for each state $x_i \in X$, $i = 1, \dots, |X|$. Select an initial population of policies $\Lambda_0 = \{\pi_1^0, \pi_2^0, \dots, \pi_n^0\}$. Construct the initial sub-MDP as $\mathcal{G}_{\Lambda_0} := (X, \Gamma_0, \Lambda_0(\cdot), P, R)$, where $\Gamma_0 = \bigcup_x \Lambda_0(x)$. Set $\pi_*^{-1} := \pi_1^0$, $k = 0$.
  • Repeat until a specified stopping rule is satisfied:
    – Policy Improvement with Reward Swapping (PIRS):
      ∗ Obtain the value function $V^{\pi_j^k}$ for each $\pi_j^k \in \Lambda_k$.
      ∗ Generate the elite policy for $\mathcal{G}_{\Lambda_k}$ as
        $$\pi_*^k(x) \in \arg\max_{u \in \Lambda_k(x)} \left\{ R(x,u) + \gamma \sum_{y \in X} P(x,u)(y)[\max_{\pi_j^k \in \Lambda_k} V^{\pi_j^k}(y)] \right\}, \ x \in X.$$
    – Sub-MDP Generation:
      ∗ for $j = 2$ to $n$
          for $i = 1$ to $|X|$
            generate random number $U \sim U(0,1)$,
            if $U \le q_0$ (exploitation)
              choose the action $\pi_j^{k+1}(x_i)$ in the neighborhood of $\pi_*^k(x_i)$
              by using the "nearest neighbor" heuristic.
            elseif $U > q_0$ (exploration)
              choose the action $\pi_j^{k+1}(x_i) \in A(x_i)$ according to $\mathcal{P}_{x_i}$.
            end if
          end for
        end for
      ∗ Set the next population generation as $\Lambda_{k+1} = \left\{\pi_*^k, \pi_2^{k+1}, \dots, \pi_n^{k+1}\right\}$.
      ∗ Construct a new sub-MDP as $\mathcal{G}_{\Lambda_{k+1}} := (X, \Gamma_{k+1}, \Lambda_{k+1}(\cdot), P, R)$,
        where $\Gamma_{k+1} = \bigcup_x \Lambda_{k+1}(x)$.
      ∗ $k \leftarrow k + 1$.
```

FIG. 5. *Evolutionary Random Policy Search*

however, we can always discard the redundant members and view $\Lambda(x)$ as the set of admissible actions at state x. Since ERPS is an iterative random search algorithm, rather than attempting to solve $\mathcal{G}_\Lambda$ exactly, it is more efficient to use approximation schemes and obtain an improved policy and/or good candidate policies with worst-case performance guarantee.

Here we adopt a variant of the policy improvement technique to find an "elite" policy, one that is superior to all of the policies in the current population, by executing the following two steps:

Step 1: Obtain the value functions V^{π_j}, $j = 1, \dots, n$, by solving the equations:

$$V^{\pi_j}(x) = R(x, \pi_j(x)) + \gamma \sum_{y \in X} P(x, (\pi_j(x))(y) V^{\pi_j}(y), \ \forall\, x \in X. \tag{26}$$

Step 2: Compute the elite policy π_* by

$$\pi_*(x) \in \arg\max_{u \in \Lambda(x)} \left\{ R(x,u) + \gamma \sum_{y \in X} P(x,u)(y)[\max_{\pi_j \in \Lambda} V^{\pi_j}(y)] \right\}, \ \forall\, x \in X. \tag{27}$$

Since in Equation (27), we are basically performing the policy improvement on the "swapped reward" $\max_{\pi_j \in \Lambda} V^{\pi_j}(x)$, we call this procedure "policy improvement with reward swapping" (PIRS). We remark that the "swapped reward" $\max_{\pi_j \in \Lambda} V^{\pi_j}(x)$ may not be the value function corresponding to any policy. The following theorem shows that the elite policy generated by PIRS improves any policy in Λ (for proofs, see [17]).

THEOREM 4.4. *Given $\Lambda = \{\pi_1, \pi_2, \ldots, \pi_n\}$, let $\bar{V}(x) = \max_{\pi_j \in \Lambda} V^{\pi_j}(x)\ \forall\, x \in X$, and let*

$$\mu(x) \in \arg\max_{u \in \Lambda(x)} \left\{ R(x,u) + \gamma \sum_{y \in X} P(x,u)(y)\bar{V}(y) \right\}.$$

Then $V^{\mu}(x) \geq \bar{V}(x),\ \forall\, x \in X$. Furthermore, if μ is not optimal for $\mathcal{G}_\Lambda$, then $V^{\mu}(x) > \bar{V}(x)$ for at least one $x \in X$.

At the *kth* iteration, given the current policy population Λ_k, we compute the *kth* elite policy π_*^k via PIRS. According to Theorem 4.4, the elite policy improves any policy in Λ_k, and since π_*^k is directly used to generate the $(k+1)$*th* sub-MDP (see Figure 5 and Section 4.2.3), the following monotonicity property follows by induction.

COROLLARY 4.2. *For all $k \geq 0$,*

$$V^{\pi_*^{k+1}}(x) \geq V^{\pi_*^k}(x), \quad \forall\, x \in X.$$

We now provide an intuitive comparison between PIRS and policy switching, which is used in EPI and directly operates upon individual policies in the population via (25), with a computational complexity is $O(n|X|)$. For a given group of policies Λ, let Ω be the policy space for the sub-MDP $\mathcal{G}_\Lambda$; it is clear that the size of Ω is on the order of $n^{|X|}$. Policy switching only takes into account each individual policy in Λ, while PIRS tends to search the entire space Ω, which is much larger than Λ. Although it is not clear in general that the elite policy generated by PIRS improves the elite policy generated by policy switching, since the policy improvement step is quite fast and it focuses on the best policy updating directions, we believe this will be the case in many situations. For example, consider the case where one particular policy, say $\bar{\pi}$, dominates all other policies in Λ. It is obvious that policy switching will choose $\bar{\pi}$ as the elite policy; thus, no further improvement can be achieved. In contrast, PIRS considers the sub-MDP $\mathcal{G}_\Lambda$; as long as $\bar{\pi}$ is not optimal for $\mathcal{G}_\Lambda$, a better policy can always be obtained (cf. the related discussion in [7]).

The computational complexity of each iteration of PIRS is approximately the same as that of policy switching, because step 1 of PIRS, i.e., Equation (26), which is also used by policy switching, requires solution of n systems of linear equations, and the number of operations required by using a direct method (e.g., Gaussian Elimination) is $O(n|X|^3)$, and this dominates the cost of step 2, which is at most $O(n|X|^2)$.

4.2.3. Sub-MDP Generation. The description of the "sub-MDP generation" step in Figure 5 is only at a conceptual level. In order to elaborate, we need to distinguish between two cases. We first consider the discrete action space case; then we discuss the setting where the action space is continuous.

4.2.4. Discrete Action Spaces. According to Corollary 4.2, the performance of the elite policy at the current iteration is no worse than the performances of the elite policies generated at previous iterations; thus the PIRS step alone can be viewed as a local search procedure with memory. Our concern now is how to achieve continuous improvements among the elite policies found at consecutive iterations. One possibility is to use unbiased random sampling and choose at each iteration a sub-MDP problem by making use of the *action selection distribution* $\mathcal{P}_x$. The sub-MDPs at successive iterations are then independent of one another, and it is intuitively clear that we may obtain improved elite policies after a sufficient number of iterations. Such an unbiased sampling scheme is very effective in escaping local optima and is often useful in finding a good candidate solution. However, in practice, persistent improvements will be more and more difficult to achieve as the number of iterations (sampling instances) increases, since the probability of finding better elite policies becomes smaller and smaller. Thus, it appears that a biased sampling scheme could be more helpful, which can be accomplished by using a "nearest neighbor" heuristic.

To achieve a biased sampling configuration, ERPS combines exploitation ("nearest neighbor" heuristic) with exploration (unbiased sampling). The key to balance these two types of searches is the use of the exploitation probability q_0. For a given elite policy π, we construct a new policy, say $\hat{\pi}$, in the next population generation as follows: At each state $x \in X$, with probability q_0, $\hat{\pi}(x)$ is selected from a small neighborhood of $\pi(x)$; and with probability $1 - q_0$, $\hat{\pi}(x)$ is chosen by using the unbiased random sampling. The preceding procedure is performed repeatedly until we have obtained $n - 1$ new policies, and the next population generation is simply formed by the elite policy π and the $n-1$ newly generated policies. Intuitively, on the one hand, the use of exploitation will introduce more robustness into the algorithm and helps to locate the exact optimal policy, while on the other hand, the exploration step will help the algorithm to escape local optima and to find attractive policies quickly. In effect, we see that this idea is equivalent to altering the underlying *action selection distribution*, in that $\mathcal{P}_x$ is artificially made more peaked around the action $\pi(x)$.

If we assume that A is a non-empty metric space with a defined metric $d(\cdot,\cdot)$, then the "nearest neighbor" heuristic in Figure 5 could be implemented as follows:

Let r_i, a positive integer, be the search range for state x_i, $i = 1, 2, \ldots, |X|$. We assume that $r_i < |A(x_i)|$ for all i, where $|A(x_i)|$ is the size of $A(x_i)$.

- Generate a random variable $l \sim DU(1, r_i)$, where $DU(1, r_i)$ represents the discrete uniform distribution between 1 and r_i. Set $\pi_j^{k+1}(x_i) = a \in A(x_i)$ such that a is the lth closest action to $\pi_*^k(x_i)$ (measured by $d(\cdot,\cdot)$).

Remark 4.2. *Sometimes the above procedure is not easy to implement. It is often necessary to index a possibly high-dimensional metric space, whose complexity will depend on the dimension of the problem and the cost in evaluating the distance functions. However, we note that the action spaces of many MDP problems are subsets of $\Re^N$, where many efficient methods can be applied, such as Kd-trees [3] and R-trees [15]. The most favorable situation is an action space that is "naturally ordered", e.g., in inventory control problems where actions are the number of items to be ordered $A = \{0, 1, 2, \cdots\}$, in which case the indexing and ordering becomes trivial.*

Remark 4.3. *In EPI, policies in a new generation are generated by the so-called "policy mutation" procedure, where two types of mutations are considered: "global mutation" and "local mutation". The algorithm first decides whether to mutate a given policy π "globally" or "locally" according to a mutation probability P_m. Then at each state x, $\pi(x)$ is mutated with probability P_g or P_l, where P_g and P_l are the respective predefined global mutation and local mutation probabilities. It is assumed that $P_g \gg P_l$; the idea is that "global mutation" helps the algorithm to get out of local optima and "local mutation" helps the algorithm to fine-tune the solution. If a mutation is to occur, the action at the mutated state x is changed by using the action selection probability $\mathcal{P}_x$. As a result, we see that each action in a new policy generated by "policy mutation" either remains unchanged or is altered by pure random sampling; although the so-called "local mutation" is used, no local search element is actually involved in the process. Thus, the algorithm only operates at the global level. We note that this is essentially equivalent to setting the exploitation probability $q_0 = 0$ in our approach.*

4.2.5. Continuous Action Spaces. The biased sampling idea in the previous section can be naturally extended to MDPs with continuous action spaces. We let $\mathcal{B}_A$ be the smallest σ-algebra containing all the open sets in A, and choose the *action selection distribution* $\mathcal{P}$ as a probability measure defined on $(A, \mathcal{B}_A)$. Again, denote the metric defined on A by $d(\cdot,\cdot)$.

By following the "nearest neighbor" heuristic, we now give a general implementation of the exploitation step in Figure 5.

Let $r_i > 0$ denote the search range for state x_i, $i = 1, 2, \ldots, |X|$.

- Choose an action uniformly from the set of neighbors $\{a : d(a, \pi_*^k(x_i)) \leq r_i,\ a \in A(x_i)\}$.

Note the difference in the search range r_i between the discrete action space case and the continuous action space case. In the former case, r_i is a positive integer indicating the number of candidate actions that are the closest to the current elite action $\pi_*^k(x_i)$, whereas in the latter case, r_i is the distance from the current elite action, which may take any positive value.

If we further assume that A is a non-empty open connected subset of $\Re^N$ with some metric (e.g., the infinity-norm), then a detailed implementation of the above exploitation step is as follows.

- Generate a random vector $\lambda^i = (\lambda_1^i, \ldots, \lambda_N^i)^T$ with each $\lambda_h^i \sim U[-1, 1]$ independent for all $h = 1, 2, \ldots, N$, and choose the action $\pi_j^{k+1}(x_i) = \pi_*^k(x_i) + \lambda^i r_i$.
- *If* $\pi_j^{k+1}(x_i) \notin A(x_i)$, then repeat the above step.

In this specific implementation, the same search range r_i is used along all directions of the action space. However, in practice, it may often be useful to generalize r_i to a N-dimensional vector, where each component controls the search range in a particular direction of the action space.

Remark 4.4. *Note that the action space does not need to have any structure other than being a metric space. The metric $d(\cdot, \cdot)$ used in the "nearest neighbor" heuristic implicitly imposes a structure on the action space. It follows that the efficiency of the algorithm depends on how the metric is actually defined. Like most of the random search methods for global optimizations, our approach is designed to explore the structure that good policies tend to be clustered together. Thus, in our context, a good metric should have a good potential in representing this structure. For example, the discrete metric (i.e., $d(a, a) = 0\ \forall\ a \in A$ and $d(a, b) = 1\ \forall\ a, b \in A,\ a \neq b$) should never be considered as a good choice, since it does not provide us with any useful information about the action space. For a given action space, a good metric always exists but may not be known a priori. In the special case where the action space is a subset of $\Re^N$, we take the Euclidean metric as the default metric, this is in accord with most of the optimization techniques employed in $\Re^N$.*

4.2.6. Stopping Rule. Different stopping criteria can be used. The algorithm is stopped when a predefined maximum number of iterations is reached or stopped when no further improvement in the value function is obtained for several, say K, consecutive iterations. Specifically, the algorithm is stopped if $\exists\ k > 0$, such that $\|V^{\pi_*^{k+m}} - V^{\pi_*^k}\| = 0\ \forall\ m = 1, 2, \ldots, K$.

4.3. Convergence Analysis. In this section, we discuss the convergence properties of ERPS. To do so, the following notation is necessary. As before, denote by $d(\cdot,\cdot)$ the metric on the action space A. We define the distance between two policies π^1 and π^2 by

$$d_\infty(\pi^1,\pi^2) := \max_{1\le i\le |X|} d(\pi^1(x_i),\pi^2(x_i)).$$

For a given policy $\hat{\pi}\in\Pi$ and any $\sigma>0$, we further define the σ-neighborhood of $\hat{\pi}$ by

$$\mathcal{N}(\hat{\pi},\sigma) := \{\pi|\ d_\infty(\hat{\pi},\pi)\le\sigma,\ \forall\,\pi\in\Pi\}.$$

For each policy $\pi\in\Pi$, we also define P_π as the transition matrix whose $(x,y)th$ entry is $P(x,\pi(x))(y)$ and R_π as the one-stage reward vector whose $(x)th$ entry is $R(x,\pi(x))$.

As the ERPS method is randomized, different runs of the algorithm will give different sequences of elite policies (i.e., sample paths); thus the algorithm induces a probability distribution over the set of all sequences of elite policies. We denote by $\hat{\mathcal{P}}(\cdot)$ and $\hat{E}(\cdot)$ the probability and expectation taken with respect to this distribution.

Let $\|\cdot\|_\infty$ denote the infinity-norm, given by $\|V\|_\infty := \max_{x\in X}|V(x)|$. We have the following convergence result for the ERPS algorithm.

THEOREM 4.5. *Let π^* be an optimal policy with corresponding value function V^{π^*}, and let the sequence of elite policies generated by ERPS together with their corresponding value functions be denoted by $\{\pi_*^k,\ k=1,2,\ldots\}$ and $\{V^{\pi_*^k},\ k=1,2,\ldots\}$, respectively. Assume that:*

1. $q_0<1$.
2. *For any given $\ell>0$, $\mathcal{P}_x(\{a|\ d(a,\pi^*(x))\le\ell,\ a\in A(x)\})>0,\ \forall\,x\in X$, (recall that $\mathcal{P}_x(\cdot)$ is a probability measure on the set of admissible actions $A(x)$).*
3. *There exist constants $\sigma>0,\ \phi>0,\ L_1<\infty$, and $L_2<\infty$, such that for all $\pi\in\mathcal{N}(\pi^*,\sigma)$ we have $\|P_\pi-P_{\pi^*}\|_\infty\le\max\left\{L_1 d_\infty(\pi,\pi^*),\frac{1-\gamma}{\gamma}-\phi\right\}$ $(0<\gamma<1)$, and $\|R_\pi-R_{\pi^*}\|_\infty\le L_2 d_\infty(\pi,\pi^*)$.*

Then for any given $\varepsilon>0$, there exists a random variable $\mathcal{M}_\varepsilon>0$ with $\hat{E}(\mathcal{M}_\varepsilon)<\infty$ such that $\|V^{\pi_*^k}-V^{\pi^*}\|_\infty\le\varepsilon\ \ \forall\ k\ge\mathcal{M}_\varepsilon$.

REMARK 4.5. *Assumption* 1 *restricts the exploitation probability from pure local search. Assumption* 2 *simply requires that any "ball" that contains the optimal policy will have a strictly positive probability measure. It is trivially satisfied if the set $\{a|d(a,\pi^*(x))\le\ell,\ a\in A(x)\}$ has a positive (Borel) measure $\forall\ x\in X$ and the* action selection distribution *$\mathcal{P}_x$ has infinite tails (e.g., Gaussian distribution). Assumption* 3 *imposes some Lipschitz type of conditions on P_π and R_π; as we will see, it formalizes the notion that near optimal policies are clustered together (see remark* 3*). The* *assumption can be verified if P_π and R_π are explicit functions of π. For a given $\varepsilon > 0$, such a policy π satisfying $\|V^\pi - V^{\pi^*}\|_\infty \leq \varepsilon$ is referred to as an ε-optimal policy.*

REMARK 4.6. *The result in Theorem 4.5 implies the a.s. convergence of the sequence $\{V^{\pi_*^k}, k = 0, 1, \ldots\}$ to the optimal value function V^{π^*}. To see this, note that Theorem 4.5 implies that $\hat{\mathcal{P}}(\|V^{\pi_*^k} - V^{\pi^*}\|_\infty > \varepsilon) \to 0$ as $k \to \infty$ for every given ε, which means that the sequence converges in probability. Furthermore, since $\|V^{\pi_*^k} - V^{\pi^*}\|_\infty \leq \varepsilon \;\; \forall\, k \geq \mathcal{M}_\varepsilon$ is equivalent to $\sup_{\bar{k}\geq k} \|V^{\pi_*^{\bar{k}}} - V^{\pi^*}\|_\infty \leq \varepsilon \;\; \forall\, k \geq \mathcal{M}_\varepsilon$, we will also have $\hat{\mathcal{P}}(\sup_{\bar{k}\geq k} \|V^{\pi_*^{\bar{k}}} - V^{\pi^*}\|_\infty > \varepsilon) \to 0$ as $k \to \infty$, and the a.s. convergence thus follows.*

The performance of the population-based algorithms is illustrated with some numerical examples in [17], including a comparison with PI to illustrate the substantial computational efficiency gains attainable by the approach.

5. Conclusions.

This paper provides an overview of a number of recent contributions to the literature on simulation-based algorithms for MDPs. These contributions include algorithms and convergence results for multi-stage adaptive sampling, evolutionary policy iteration, and evolutionary random policy search. For a more complete discussion of these methods, including convergence proofs and numerical results, the reader is referred to the book [11]; this book also includes detailed discussion of other approaches, including the application of randomized global optimization algorithms and approximate receding horizon control to the solution of MDPs.

Acknowledgments. This work was supported in part by the National Science Foundation under Grants DMI-9988867 and DMI-0323220, and in part by the Air Force Office of Scientific Research under Grants F496200110161, FA95500410210, and FA95500710366. The work of H. S. Chang was also supported by Sogang University Research Grants in 2007.

REFERENCES

[1] R. AGRAWAL, *Sample mean based index policies with $O(\log n)$ regret for the multi-armed bandit problem*, Advances in Applied Probability, 27(1995), pp. 1054–1078.

[2] P. AUER, N. CESA-BIANCHI, AND P. FISHER, *Finite-time analysis of the multiarmed bandit problem*, Machine Learning, 47(2002), pp. 235–256.

[3] J. BENTLEY, *Multidimensional binary search trees in database applications*, IEEE Transactions on Software Engineering, 5(1979), pp. 333–340.

[4] D. P. BERTSEKAS, *Dynamic Programming and Optimal Control, Volumes 1 and 2.* Athena Scientific, 1995.

[5] D. P. BERTSEKAS AND J. N. TSITSIKLIS, *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[6] V. S. BORKAR, *White-noise representations in stochastic realization theory*, SIAM J. on Control and Optimization, 31(1993), pp. 1093–1102.

[7] H. S. Chang, R. Givan, and E. K. P. Chong, *Parallel rollout for on-line solution of partially observable Markov decision processes*, Discrete Event Dynamic Systems: Theory and Application, 15:3(2004), pp. 309–341.

[8] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, *An adaptive sampling algorithm for solving Markov decision processes*, Operations Research, 53:1(2005), pp. 126–139.

[9] H. S. Chang, H. G. Lee, M. C. Fu, and S. I. Marcus, *Evolutionary Policy Iteration for Solving Markov Decision Processes*, IEEE Transactions on Automatic Control, 50(2005), pp. 1804-1808.

[10] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, *An asymptotically efficient simulation-based algorithm for finite horizon stochastic dynamic programming*, IEEE Trans. on Automatic Control, 52:1(2007), pp. 89–94.

[11] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, *Simulation-Based Algorithms for Markov Decision Processes*. Springer-Verlag, New York, 2007.

[12] E. Fernández-Gaucherand, A. Arapostathis, and S. I. Marcus, *On the average cost optimality equation and the structure of optimal policies for partially observable Markov processes*, Annals of Operations Research, 29(1991), pp. 471–512.

[13] M. C. Fu, S. I. Marcus, and I-J. Wang, *Monotone optimal policies for a transient queueing staffing problem*, Operations Research, 46(2000), pp. 327–331.

[14] R. Givan, S. Leach, and T. Dean, *Bounded Markov decision processes*, Artificial Intelligence, 122(2000), pp. 71–109.

[15] A. Guttman, *R-trees: A Dynamic Index Structure for Spatial Searching*, in: Proceedings ACM SIGMOD'84, 47–57, 1984.

[16] O. Hernández-Lerma and J. B. Lasserre, *Discrete-Time Markov Control Processes: Basic Optimality Criteria*. Springer-Verlag, New York, 1996.

[17] J. Hu, M. C. Fu, V. Ramezani, and S. I. Marcus, *An Evolutionary Random Policy Search Algorithm for Solving Markov Decision Processes*, to appear in INFORMS Journal on Computing, 2007.

[18] L. Kaelbling, M. Littman, and A. Moore, *Reinforcement learning: a survey*, Artificial Intelligence, 4(1996), pp. 237–285.

[19] T. Lai and H. Robbins, *Asymptotically efficient adaptive allocation rules*, Advances in Applied Mathematics, 6(1985), pp. 4–22.

[20] K. S. Narendra and M. Thathachar, *Learning Automata: An Introduction*. Prentice-Hall, 1989.

[21] E. L. Porteus, *Conditions for characterizing the structure of optimal strategies in infinite-horizon dynamic programs*, J. Optim. Theory Appl., 36(1982), pp. 419–432.

[22] W. B. Powell, *Approximate Dynamic Programming for Asset Management: Solving the Curses of Dimensionality*. Springer, New York, 2006, in preparation.

[23] A. S. Poznyak and K. Najim, *Learning Automata and Stochastic Optimization*, Springer-Verlag, New York, 1997.

[24] A. S. Poznyak, K. Najim, and E. Gomez-Ramirez, *Self-Learning Control of Finite Markov Chains,* Marcel Dekker, Inc. New York, 2000.

[25] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[26] K. Rajaraman and P. S. Sastry, *Finite time analysis of the pursuit algorithm for learning automata*, IEEE Trans. on Systems, Man, and Cybernetics, Part B, 26:4(1996), pp. 590–598.

[27] J. Si, A. G. Barto, W. B. Powell, and D. W. Wunsch, editors, *Handbook of Learning and Approximate Dynamic Programming*. Wiley Inter-Science, Hoboken, NJ, 2004.

[28] J. E. Smith and K. F. McCardle, *Structural properties of stochastic dynamic programs*, Operations Research, 50(2002), pp. 796–809.

[29] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, Massachusetts, 1998.

[30] J. N. Tsitsiklis, *Asynchronous stochastic approximation and Q-learning*, Machine Learning, 16(1994), pp. 185–202.

[31] B. Van Roy, *Neuro-dynamic programming: overview and recent trends*, in: Handbook of Markov Decision Processes: Methods and Applications, E.A. Feinberg and A. Shwartz, eds., Kluwer, 2002.

[32] C. J. C. H. Watkins, *Q-learning*, Machine Learning, 8:3(1992), pp. 279–292.

[33] R. M. Wheeler, Jr. and K. S. Narendra, *Decentralized learning in finite Markov chains*, IEEE Trans. on Automatic Control, 31:6(1986), pp. 519–526.