

Prioritizing Bellman Backups Without a Priority Queue

Peng Dai

Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046
daipeng@uky.edu

Eric A. Hansen

Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762-9637
hansen@cse.msstate.edu

Abstract

Several researchers have shown that the efficiency of value iteration, a dynamic programming algorithm for Markov decision processes, can be improved by prioritizing the order of Bellman backups to focus computation on states where the value function can be improved the most. In previous work, a priority queue has been used to order backups. Although this incurs overhead for maintaining the priority queue, previous work has argued that the overhead is usually much less than the benefit from prioritization. However this conclusion is usually based on a comparison to a non-prioritized approach that performs Bellman backups on states in an arbitrary order. In this paper, we show that the overhead for maintaining the priority queue can be greater than the benefit, when it is compared to very simple heuristics for prioritizing backups that do not require a priority queue. Although the order of backups induced by our simple approach is often sub-optimal, we show that its smaller overhead allows it to converge faster than other state-of-the-art priority-based solvers.

Introduction

Markov decision processes are a widely-used framework for planning under uncertainty (Boutilier, Dean, & Hanks 1999). Dynamic programming algorithms for solving Markov decision processes, such as value iteration and policy iteration, iteratively improve a solution by performing a succession of *Bellman backups* (defined in the next section), one backup per state in each iteration of the algorithm. Eventual convergence is guaranteed no matter what order states are updated in each iteration, but convergence can be significantly accelerated by choosing an intelligent ordering of backups. This was most notably shown in the *prioritized sweeping* algorithm of Moore and Atkinson (1993). Moore and Atkinson showed that performing Bellman backups in order of the likely change in the value of the states, instead of in arbitrary order, significantly improves convergence time. Subsequently, several other researchers developed variations of this approach that further improve its performance (Andre, Friedman, & Parr 1998; Dearden 2001; Ferguson & Stentz 2004b; McMahan & Gordon 2005; Rayner *et al.* 2007).

These approaches use a priority queue to determine the order in which to perform the Bellman backups. But maintaining this priority queue introduces a logarithmic overhead per backup. Assuming that each state has a bounded number of successors (so that the asymptotic complexity of the backup itself is constant), the use of a priority queue makes the asymptotic complexity of a single iteration of backups $O(n \log n)$ instead of $O(n)$. Despite this overhead, previous researchers have shown that prioritizing backups significantly improves the performance of dynamic programming by reducing the number of iterations and the time to convergence. However, the baseline for this improvement is usually the performance of dynamic programming when it performs backups in arbitrary order.

In this paper, we describe a prioritized value iteration algorithm that uses very simple heuristics for prioritizing backups that do not require a priority queue. For example, the order of backups can be determined as follows. Beginning with the goal state, the algorithm traverses the transpose of the state space graph using a breadth-first (or depth-first) search and checks for duplicates so that it does not perform a backup for the same state more than once in the same iteration. States traversed during the search are backed up in the order they are encountered. In addition, the traversal only considers the predecessors of a state when they are associated with an action that is likely to be part of the current optimal policy, as determined by the value associated with that action compared to the value associated with other actions. The traversal terminates when there are no more reachable states. This method of ordering backups in value iteration is extremely simple. Yet, for a range of test domains, our results show that it is also very effective, and consistently and significantly outperforms methods that use a priority queue.

The paper is organized as follows. We begin with a review of algorithms for Markov decision processes that use priority-based backups to improve performance, including multiple variations of prioritized sweeping. Then, we describe our simple heuristics for ordering backups, show empirically that they result in algorithms that outperform other priority-based value iteration solvers on a range of test problems, and analyze the reasons for this. The paper concludes with a discussion of some potential extensions of this work.

Background

We begin with a brief overview of Markov decision processes and the value iteration algorithm for solving them. Then we review prioritized sweeping and some of its variations.

Markov Decision Process

A Markov decision process is a four-tuple $\langle S, A, T, C \rangle$, where S is a finite set of system states, A is a finite set of actions, T is the transition function, and C is the immediate cost function. We consider a discrete-time MDP that unfolds over a sequence of time steps called *stages*. In each stage t , the system is in a particular state $s \in S$, an action $a \in A_s$ is taken (where A_s is the set of actions that can be taken in state s), and the system makes a transition to a successor state $s' \in S$ at stage $t + 1$. The *Markovian property* guarantees that s' only depends on the pair (s, a) . Therefore, the transition function for each action, $T_a: S \times S \rightarrow [0, 1]$, gives the probability of a transition from s to s' as a result of action a . The cost function $C: S \times A \rightarrow \mathbb{R}$ specifies the expected immediate cost of performing a particular action in a particular state.

The *horizon* of an MDP is the total number of stages over which the system evolves. In problems where the horizon is a finite number H , our aim is to find the action to take at each stage and state that minimizes expected total cost. More concretely, the chosen actions a^0, \dots, a^{H-1} should minimize the expected value of $f(s) = \sum_{i=0}^{H-1} C(s^i, a^i)$, where $s_0 = s$. For infinite-horizon problems, in order to favor shorter-term consequences and ensure that the expected total cost is finite, we introduce a discount factor $\gamma \in (0, 1]$ for future costs. In this case, the objective is to minimize $f(s) = \sum_{i=0}^{\infty} \gamma^i C(s^i, a^i)$.

In our paper, we consider a class of MDPs called *goal-based MDPs*. A goal-based MDP has two additional components, s_0 and G , where $s_0 \in S$ is the initial state and $G \subseteq S$ is a set of goal states. In particular, we consider a type of goal-based MDP called a *stochastic shortest path problem* where $\gamma=1$ and $\forall g \in G, A_g = \emptyset$, which means the process terminates upon entering a goal state. In a stochastic shortest path problem, we want to find a shortest, or more generally least costly, stochastic path (which is actually a reachability graph) between s_0 and any goal state. We further assume that at least one goal state is always reachable from s_0 , which means, there is always a solution to this problem.

Given an MDP, we define a *policy* $\pi: S \rightarrow A$ to be a mapping from the state space to the action space. A *value function* for a policy π , denoted $V_\pi: S \rightarrow \mathbb{R}$, gives the expected total cost of the system, starting from state s and acting according to the policy π until termination at a goal state. An *optimal policy* π^* is a policy that has the minimal expected cumulative cost, and its value function $V_{\pi^*}(\cdot)$ is the *optimal value function* $V^*(\cdot)$. To solve an MDP, we normally want to find an optimal policy and optimal value function. Bellman (1957) shows that the optimal value function satisfies the following equation for all states s ,

$$V^*(s) = \min_{a \in A_s} \left[C(s, a) + \gamma \sum_{s' \in S} T_a(s'|s) V^*(s') \right].$$

This is called the *Bellman optimality equation*. An optimal policy is easily extracted from the optimal value function by choosing the best action for each state. The Bellman optimality equation provides a basis for dynamic programming algorithms for solving MDPs. When treated as an assignment statement, we call the application of the Bellman equation to a particular state s a *Bellman backup* for state s . The *Bellman residual* of a state s is defined as the difference of $V(s)$ after and before a backup. The *Bellman error* is the maximum Bellman residual over the state space.

Algorithm 1 Backup

Input: s
 $oldV \leftarrow V(s)$
 $V(s) \leftarrow \min_a \{C(s, a) + \gamma \sum_{s' \in S} T_a(s'|s) V(s')\}$
 $\pi(s) \leftarrow \operatorname{argmin}_a \{C(s, a) + \gamma \sum_{s' \in S} T_a(s'|s) V(s')\}$
return $V(s) - oldV$

Value Iteration and Its Inefficiency

Value iteration (Bellman 1957) is a simple and general dynamic programming algorithm for MDPs. Beginning with an initial value function, it iteratively improves the value function until convergence. In each iteration, it performs a backup for all states in some order (which is nondeterministic). When the Bellman error is smaller than a threshold value ϵ , value iteration has converged and terminates.

Although value iteration is a powerful algorithm, it has some potential inefficiencies that are related to how the nondeterministic aspects of the algorithm are implemented. First, some backups can be useless. Technically, a useful backup is one that is performed on a state whose successors have experienced value changes after its most recent backup. In addition, in stochastic-shortest path problems, backing up states that are unreachable from the initial state is useless, since they cannot contribute to the final $V^*(s_0)$. Second, backups may not be ordered intelligently. For example, performing backups on a state after backing up its successors can be more helpful than backups performed in the opposite order. In general, backups performed with some clever ordering can be more efficient than those performed in an arbitrary order. Priority-based methods for value iteration have been proposed with these considerations in mind. The basic idea is to order backups so that each backup is only done when it is potentially useful or necessary, so that the problem can be solved by performing the least number of backups.

Algorithm 2 Value iteration

Input: S, ϵ
repeat
 for all $s \in S$ **do**
 $Residual(s) \leftarrow \text{Backup}(s)$
 end for
until $\forall s, Residual(s) < \epsilon$

Algorithm 3 Prioritized sweeping

Input: $S, goal, \epsilon$
 $V(goal) \leftarrow 0$
 $Q \leftarrow \{goal\}$
while $Q \neq \emptyset$ **do**
 remove the first state s from Q
 $residual(s) \leftarrow Backup(s)$
 if $residual(s) > \epsilon$ **or** $s = goal$ **then**
 for all $s' \in Pred(s)$ **do**
 calculate $priority(s')$
 insert s' into Q according to $priority(s')$
 end for
 end if
end while

Prioritized Sweeping

The prioritized sweeping (PS) algorithm (Moore & Atkeson 1993) was first introduced in the reinforcement learning literature, but is a general technique that has also been used in dynamic programming (Andre, Friedman, & Parr 1998; McMahan & Gordon 2005). The main idea of PS is to order backups intelligently by maintaining a queue, where the priority of each element (state) in the queue represents the potential improvement for other state values as a result of backing up that state. The priority queue is updated as the algorithm sweeps through the state space. When used in an offline dynamic programming algorithm, such as value iteration, PS can begin by inserting the goal state in the priority queue. At each step, PS pops a state s from the queue with the highest priority and performs a Bellman backup of that state. If the Bellman residual, denoted $residual(s)$, is greater than some threshold value ϵ , then PS considers all predecessors of state s ; for each state s' for which there is some action a for which $T_a(s|s') > 0$, PS inserts the predecessor state s' into the queue according to its priority. (If s' is already in the queue and its priority can be potentially enhanced, it updates its priority.) We discuss various priority metrics below.

McMahan and Gordon (2005) make a useful distinction between two implementations of PS, which they call *sweeps* and *multiple updates*. The *sweeps* implementation is similar to classical value iteration in that it performs a sequence of iterations, and updates each state no more than once in each iteration. It has the same convergence test as classic value iteration. The *multiple updates* implementation continues to update predecessor states without checking whether and when they have been updated before; thus its updates cannot be naturally divided into iterations. It converges when the priority queue is empty. McMahan and Gordon report that multiple updates typically performs better. It is also the most common implementation of PS. Algorithm 3 shows high-level pseudocode for this implementation of PS.

Priority Metrics

The performance of PS depends on the priority metric that is used to order states in the priority queue. In the original version of prioritized sweeping, after a backup is performed

for state s , the priority of any state s' that is a predecessor of state s via some action a is set as follows:

$$priority(s') \leftarrow T_a(s|s') \times residual(s).$$

Several researchers have investigated alternative priority metrics (Wingate & Seppi 2005; McMahan & Gordon 2005; Ferguson & Stentz 2004b). In the rest of this section, we consider two alternatives that have been shown to lead to improved performance.

Improved Prioritized Sweeping McMahan and Gordon (2005) introduce a variant of prioritized sweeping, called *improved prioritized sweeping* (IPS), that is designed to behave like Dijkstra's algorithm for problems with deterministic state transitions, and can also be used to solve MDPs with stochastic transitions. They argue that it has an advantage for MDPs with transitions that are almost deterministic.

Improved PS differs from standard PS in how it computes the priority of a state. The priority of a predecessor state s' is computed as follows:

$$priority(s') \leftarrow residual(s')/V(s').$$

Because the priority depends on the the residual of the state itself after a backup, rather than the residual of a successor state, improved prioritized sweeping differs from the pseudocode shown in Algorithm 3 in one other way. It performs a backup of a state as soon as it pushes the state on the priority queue, instead of waiting until it pops it off the priority queue.

Focussed Dynamic Programming Focussed dynamic programming (FDP) (Ferguson & Stentz 2004b; 2004a) is a variant of prioritized sweeping that exploits knowledge of the start state to focus computation on states that are reachable from the start state. To do this, it uses a priority metric that is defined using two heuristic functions: a static heuristic function $h(\cdot)$, where $h(s)$ is an admissible estimate of the expected cost for reaching state s from the start state, and a dynamic heuristic function $g(\cdot)$, where $g(s)$ is an estimate of the expected cost for reaching the goal state from state s . When a state s is removed from the priority queue, a backup is performed for each of its predecessor states s' , just as in improved prioritized sweeping. If the residual for state s' is greater than the threshold ϵ , it is inserted into the priority queue with the following priority:

$$priority(s') \leftarrow h(s') + g(s').$$

In contrast to other forms of PS, FDP removes the state with the lowest priority value from the priority queue, rather than the state with the highest priority value, since it is interested in states through which the shortest path passes. The algorithm terminates when the lowest priority value in the queue is greater than the current value $V(s_0)$, where s_0 is the start state, since this means $V(s_0)$ cannot be improved further by considering the remaining states in the queue.

Priority-Queue Free Methods

Priority-based methods have proven useful in solving MDPs because they order backups in an intelligent way. A state

Algorithm 4 Backwards value iteration

Input: $S, goal, \epsilon$
 $iter \leftarrow 0$
repeat
 $iter \leftarrow iter + 1$
 $LargestResidual \leftarrow 0$
 $Q \leftarrow \{goal\}$
 while $Q \neq \emptyset$ **do**
 remove the first state s from Q
 $s.visited \leftarrow iter$
 $residual(s) \leftarrow Backup(s)$
 if $residual(s) > LargestResidual$ **then**
 $LargestResidual \leftarrow residual(s)$
 end if
 for all $s' \in PolicyPred(s)$ **do**
 if $s'.visited < iter$ **then**
 append s' to Q
 end if
 end for
 end while
until $LargestResidual < \epsilon$

that does not need an immediate backup will stay in the queue until its backup is potentially productive or there are no more urgent backups. However, maintaining a priority queue can be expensive, especially for problems with a large state space. Assuming that each state has a bounded number of successors (so that the asymptotic complexity of the backup itself is constant), the asymptotic complexity of a single iteration of backups is $O(n \log n)$ instead of $O(n)$.

In the following, we propose some very simple yet effective heuristics for ordering backups without a priority queue.

Backwards value iteration

We call our first method of ordering backups without a priority queue *backwards value iteration* (BVI). In each iteration or sweep, it considers states in order of backwards reachability from the goal state. To determine this ordering, it begins from the goal state, and performs a simple breadth-first (or depth-first) traversal of the transpose of the *policy graph*. By policy graph, we mean a subgraph of the state-space graph corresponding to the MDP, where the subgraph consists of all the states of the MDP, but only those transitions that result from taking the action for each state that is specified by the current policy. Note that prioritized sweeping traverses a state space by considering all predecessors of state s via any action, defined as

$$Pred(s) = \{s' | \exists a, T_a(s|s') > 0\},$$

whereas backwards value iteration considers only those predecessors of state s via the current policy, defined as

$$PolicyPred(s) = \{s' | T_{\pi(s')}(s|s') > 0\}.$$

The pseudocode of backwards value iteration can be found in Algorithm 4.

By performing backups in order of a backwards traversal of the policy graph, backwards value iteration ensures that

Algorithm 5 Forwards value iteration

Input: $S, goal, start, \epsilon$
{Assumes initial state values are admissible}
 $iter \leftarrow 0$
repeat
 $iter \leftarrow iter + 1$
 $LargestResidual \leftarrow 0$
 $dfs(start)$
until $LargestResidual < \epsilon$
{End of main algorithm.}

procedure $dfs(s)$
 $s.visited \leftarrow iter$
 for $s' \in Successors(s, \pi(s))$ **do**
 if $s'.visited < iter$ **then**
 $dfs(s')$
 end if
 end for
 $residual(s) \leftarrow backups(s)$
 if $residual(s) > LargestResidual$ **then**
 $LargestResidual \leftarrow residual(s)$
 end if
{End of procedure}

the value of a state is not updated until one or more of its successor states is updated. Because accurate state values tend to propagate backwards from the goal state, this heuristic for ordering backups has the effect of reducing the number of iterations of the algorithm before convergence, compared to an implementation of value iteration that updates states in an arbitrary order. Another advantage of performing backups in order of backwards traversal of the current policy graph is that backwards value iteration only considers states that can reach the goal, in contrast to standard value iteration, which updates the values of all states each iteration. However this advantage is also shared by prioritized sweeping.

The major difference from prioritized sweeping, of course, is that the latter uses a priority queue, with its significant overhead for insertions and deletions, and backwards value iteration does not. It uses either a FIFO queue, if the backwards traversal of the policy graph is breadth-first, or a LIFO queue (i.e., stack), if the backwards traversal is depth-first. To ensure that the same state is not backed-up more than once in the same iteration, a simple check is made that takes constant time using a bit map of the state space. The test for convergence is the same as for standard value iteration: all states backed-up in an iteration have a Bellman residual of at most ϵ . A complication is that some states may not be backed-up at all because they are *deadend states*, that is, states from which a goal state is not reachable. To ensure convergence, these states must be assigned initial values of infinite cost. The same condition must be satisfied by any prioritized sweeping algorithm that begins by inserting the goal state in the priority queue.

Forwards value iteration

Our backwards value iteration algorithm exploits reachability in the sense that it only considers states that can reach the goal state. But it does not consider whether states are reachable from the start state. Focussed dynamic programming (Ferguson & Stentz 2004b), which we considered earlier, is a variant of prioritized sweeping that focusses computation on states that are reachable from the start state by following an optimal policy. Like prioritized sweeping, it uses a priority queue with its associated overhead.

Algorithm 5 gives the pseudocode of a *forwards value iteration* (FVI) algorithm that exploits reachability from the start state without using a priority queue. It simply performs a depth-first search of the policy graph beginning from the start state. Instead of searching backwards via the predecessors of a state, it searches forward via the successors of the state under the current policy, defined as follows:

$$\text{Successors}(s, \pi(s)) = \{s' | T_{\pi(a)}(s'|s) > 0\}.$$

The search forward terminates upon reaching the goal state or a state that has been previously traversed in the same iteration, since the same state is not backed-up more than once in the same iteration. The test for convergence is the same as for backwards value iteration: all states backed-up in an iteration have a Bellman residual of at most ϵ . But in addition to requiring that deadend states have an initial value of infinity, convergence to optimality requires all initial state values to be admissible. In fact, forwards value iteration is not a new algorithm. It is identical to the efficient implementation of LAO* described by Hansen and Zilberstein (2001), and its requirement that initial state values be admissible is characteristic of any heuristic search algorithm. It is interesting to consider it together with backwards value iteration because of their similarity. One traverses the state space backwards from the goal, and the other traverses it forwards from the start state; in most other respects, that are the same.

Although forwards value iteration (or LAO*) traverses the state space forwards from the start state, it still performs backups in backwards order from the goal. Because it performs a backup for a state when it pops the state off the depth-first stack, and not when it pushes it on the stack, it typically postpones the backup of a state until the successors of the state have been backed-up.

Multiple goal or start states For simplicity in presenting all of these algorithms, we assumed that there is only one goal state and start state. If there are multiple goal states (or start states), a simple solution is to put all of them in the queue (or stack) at the beginning of the algorithm.

Experimental Results

We compared the performance of all of these algorithms in solving several test domains from the literature. All algorithms were coded in C and ran on the same processor, an Intel Pentium 4 1.50GHz with 512M main memory and a cache size of 256kB. The priority queue data structure was implemented as efficiently as possible using a max(min)-heap (Cormen *et al.* 2001).

Domains

We tested the MDP planners on four MDP domains from the literature: racetrack (RT) (Barto, Bradke, & Singh 1995), mountain car (MCar), single-arm pendulum (SAP), and double-arm pendulum (DAP) (Wingate & Seppi 2005).

Racetrack MDPs are simulations of race cars on tracks of various sizes and shapes. The state is defined by the position of the car on the track and its instantaneous velocity. In a given state, the car can take one of nine possible actions. It can move in one of eight possible directions, or it can stay put. Each move action has a small probability of failure, which leads the car to an unintended neighboring state. When a car runs into a state that represents a wall, it is sent back to the start state.

Mountain car is a well-known optimal control problem from the reinforcement learning literature. The goal is to have a car reach the top of a mountain with sufficient momentum, in a minimum amount of time. As in the racetrack domain, the state space is described by the position and velocity of the car. Single-arm pendulum and double-arm pendulum are minimum-time optimal control problems in two and four dimensions respectively. In the MCar, SAP and DAP domains, the goal states are the states that have positive instant reward from the original model of (Wingate & Seppi 2005). Although the SAP and DAP domains have only one goal state, the MCar problem domain has multiple goal states, since the destination can be reached with various speeds. In the SAP and DAP domains, unlike the MCar domain, the goal state is reachable from the entire state space. For more detailed descriptions of the MCar, SAP and DAP domains, see (Wingate & Seppi 2005).

Although the racetrack domain has a fixed start state, the other domains do not. To test algorithms that exploit a start state, we randomly picked 10 different start states (which we verified could reach the goal), and report an average result for these start states.

In the following, we refer to a domain of specific size as a *problem*, and we refer to a problem with a specific initial state as a *problem instance*. For example, an MCar domain of dimension 300×300 is a problem P and has 90,000 states s_1, \dots, s_{90000} . If we pick s_{10000} as the initial state, then it is a particular instance of problem P .

Results

Table 1 shows our experimental results for ten problems of increasing size. For these experiments, we set the cut-off time to be an hour for a problem. If an algorithm did not finish in that time, we did not report results. All algorithms were started with the same initial value function (except FDP which requires its value function to be an upper bound). For each algorithm, we recorded convergence time and total number of Bellman backups performed.

The first thing to notice is that FVI outperformed all the other algorithms. But it is only directly comparable to FDP since these are the only two algorithms that exploit knowledge of the start state to focus computation on reachable states. Usually the size of the state space that is reachable from the start state by following an optimal policy is significantly smaller than the size of the overall state space. For the

	DAP_1	SAP_1	RT_1	SAP_2	$MCar_1$	SAP_3	$MCar_2$	DAP_2	$MCar_3$	$MCar_4$
	Convergence time									
$ S $	10000	10000	21371	40000	40000	90000	90000	160000	250000	490000
$ S^R $	10000	10000	14953	40000	35819	90000	81196	90000	222757	444662
VI	1.52	3.03	3.57	19.70	14.30	62.32	44.01	44.71	223.46	579.01
PS	2.50	6.39	48.99	36.19	150.76	392.10	174.69	612.43	1339.10	–
IPS	1.89	4.99	9.22	101.58	25.17	209.13	90.12	94.95	383.31	1201.77
FDP	4.24	6.50	10.71	54.69	54.78	202.11	148.00	332.34	240.15	1373.24
BVI	1.45	4.27	4.56	24.40	4.80	74.57	16.20	31.44	59.72	160.02
DFBVI	2.46	4.14	15.39	32.57	8.56	111.69	31.63	36.64	93.73	350.61
FVI	1.59	0.98	7.32	4.04	2.29	9.85	4.93	13.87	18.67	33.05
	Number of backups performed									
VI	840000	3210000	1303631	20480000	14043180	59400000	38039625	20480000	153972846	385182280
PS	295249	1483497	1950254	6831923	26099435	78992457	13187843	84462767	84475387	–
IPS	630653	2547527	2735216	39823914	9206009	73175495	30073911	24200774	112561141	316014001
FDP	1831535	4102411	3176598	33912510	39467741	109794283	80106968	136374088	106789110	656146226
BVI	310000	1270000	812098	6520000	1353598	19170000	4204616	6080000	14875474	39418990
DFBVI	730000	1740000	4231458	12040000	3348374	38250000	11320120	9440000	30417014	107184220
FVI	518854	291102	745275	2756448	1199150	4165310	1341434	5933171	10750117	19788609

Table 1: Convergence time and number of Bellman backups for a range of test problems ($\epsilon = 10^{-6}$)

DAP and SAP problems, it is always more than 30% of the state space, but for the MCar and racetrack problems, it is around 10%. The exact percentage depends on the location of the random start state.

Among the algorithms that solve the problem for all states that can reach the goal, the two implementations of backwards value iteration performed best. We let BVI denote the implementation that traverses the state space in breadth-first backwards order, and let DFBVI denote the implementation that uses a depth-first traversal. The breadth-first traversal resulted in faster convergence, and required many fewer backups. The reason for this is that it results in a more intelligent ordering of backups. In the ordering that results from a breadth-first traversal, a state is much more likely to be backed-up after all its successors have been backed-up than with a depth-first traversal.

Although the two implementations of backwards value iteration have much less overhead than algorithms that use a priority queue, they have a small amount of overhead that is not shared by a value iteration algorithm that simply sweeps through the states in a fixed order. As a result, standard value iteration is slightly faster for very small problems. But as the problem size increases, backwards value iteration outperform simple value iteration by an increasing amount.

There are two reasons for this. First, (DF)BVI does not need to consider the entire state space. States that cannot reach the goal are not considered, although they are considered by simple value iteration. In Table 1, the row labeled $|S^R|$ shows the number of states that can reach the goal state. The second and more important reason is that performing backups in backwards order from the goal state, instead of in arbitrary order, improves state values more quickly, and allows the algorithm to converge after fewer backups.

For all problems, the algorithms that use a priority queue take the longest to converge. In large part, this is due to the overhead of managing a priority queue. In the worst case, insertion and deletion operations take $O(\log n)$ time (where

n is the size of the queue). The larger the state space, the greater the overhead of these operations. Even for states already on the priority queue, simply updating their priority value is expensive; typically, an update operation involves both a deletion and an insertion. In fact, the reason IPS converges in less time than PS, even though it performs more backups for most problems (especially SAP_1 and $MCar_2$), is that it updates the priority of a state much less frequently.

Another important reason for the slower performance of the prioritized sweeping algorithms is that they perform many more backups than BVI, as Table 1 shows. For the $MCar_4$ problem, which is the most dramatic example, IPS and FDP performs more than 15 times more backups than BVI. The apparent reason for this is that prioritized sweeping considers all predecessors of a state, and not simply the predecessors in the policy graph. If the best action for state s' does not have s as a successor, then $V(s')$ does not depend directly on $V(s)$, and performing a backup of state s' after performing a backup of state s can be useless, resulting in no change in the value of state s' , even if the value of state s was changed significantly. Although unnecessary backups can also be performed by BVI, the results in Table 1 suggest they occur less frequently, and this appears due to the fact that BVI only considers predecessors in the policy graph.

One way that BVI can perform unnecessary backups is that some states, such as states near the goal, can converge to their optimal values before other states have converged, and they will still be backed-up in each iteration of BVI. A possible way to avoid unnecessary backups is by using the labeling technique discussed in (Bonet & Geffner 2003). Our initial attempt to incorporate this technique into BVI led to additional backups and slower performance. Further study is needed of this technique and others. For example, when the state space can be decomposed into a number of strongly connected components, this decomposition can be leveraged to make backward sweeps of the state space more efficient (Dai & Goldsmith 2007).

Conclusion

This paper studies the performance of priority-based MDP planning algorithms for MDPs. Previous researchers have argued that priority-based algorithms, with the help of a priority queue, can outperform non-prioritized approaches. We discovered that the overhead introduced by maintaining a priority queue can outweigh its advantages. In many cases, we also found that prioritized sweeping methods require more backups to converge than very simple heuristics for ordering backups based on either a breadth-first or depth-first traversal of the state-space graph. We do not claim that our very simple heuristics for ordering backups provide the best method for solving MDPs. We expect they can be improved. But we find it interesting that such simple methods for ordering backups perform so well.

The high overhead of prioritized sweeping algorithms was previously called attention to by Wingate and Seppi (2005), who proposed a clever but much more complicated approach that requires a static partitioning of the state space, and solves the blocks of states that result from this partitioning in a prioritized order.

We should also point out that we have only considered the use of prioritized sweeping in offline dynamic programming. Although many researchers have used prioritized sweeping for this purpose, prioritized sweeping was originally developed – and is still often used – in online algorithms that interleave action, learning, and planning (Moore & Atkeson 1993; Peng & Williams 1993; Rayner *et al.* 2007). It remains to be seen whether a similar analysis extends to this case.

Acknowledgments

The first author acknowledges the support of his advisor, Judy Goldsmith, and funding provided by NSF grant ITR-0325063. The authors thank David Wingate and Kevin Seppi for making their MDP domains available and Nicholas S. Mattei for suggestions on an earlier draft of this paper.

References

- Andre, D.; Friedman, N.; and Parr, R. 1998. Generalized prioritized sweeping. In *Proc. of the 10th conference on Advances in neural information processing systems (NIPS-97)*, 1001–1007.
- Barto, A.; Bradke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence J.* 72:81–138.
- Bellman, R. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. 13th International Conference on Automated Planning and Scheduling (ICAPS-03)*, 12–21.
- Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *J. of Artificial Intelligence Research* 11:1–94.
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms, Second Edition*. The MIT Press.
- Dai, P., and Goldsmith, J. 2007. Topological value iteration algorithm for Markov decision processes. In *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 1860–1865.
- Dearden, R. 2001. Structured prioritised sweeping. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001)*, 82–89.
- Ferguson, D., and Stentz, A. 2004a. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Carnegie Mellon University, Pittsburgh, PA.
- Ferguson, D., and Stentz, A. 2004b. Focussed propagation of MDPs for path planning. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, 310–317.
- Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence J.* 129:35–62.
- McMahan, H. B., and Gordon, G. J. 2005. Fast exact planning in Markov decision processes. In *Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*.
- Moore, A., and Atkeson, C. 1993. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning J.* 13:103–130.
- Peng, J., and Williams, R. 1993. Efficient learning and planning within the dyna framework. *Adaptive Behavior* 1(4):437–454.
- Rayner, D. C.; Davison, K.; Bulitko, V.; Anderson, K.; and Lu, J. 2007. Real-time heuristic search with a priority queue. In *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2372–2377.
- Wingate, D., and Seppi, K. D. 2005. Prioritization methods for accelerating MDP solvers. *J. of Machine Learning Research* 6:851–881.