Applied Artificial Intelligence, 26:571–587, 2012

ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839514.2012.687662

SOLVING THE SAILING PROBLEM WITH A NEW PRIORITIZED VALUE ITERATION

M. G. Garcia-Hernandez[1], J. Ruiz-Pinales[1], E. Onaindia[2], and A. Reyes-Ballesteros[3]
[1]*Universidad de Guanajuato, Salamanca, Guanajuato, Mexico*
[2]*Universitat Politècnica de València, DSIC, Valencia, España*
[3]*Instituto de Investigaciones Eléctricas, Temixco, Morelos, Mexico*

☐ *In this paper we tackle the sailing strategies problem, a stochastic shortest-path Markov decision process. The problem of solving large Markov decision processes accurately and quickly is challenging. Because the computational effort incurred is considerable, current research focuses on finding superior acceleration techniques. For instance, the convergence properties of current solution methods depend, to a great extent, on the order of backup operations. On one hand, algorithms such as topological sorting are able to find good orderings, but their overhead is usually high. On the other hand, shortest path methods, such as Dijkstra's algorithm, which is based on priority queues, have been applied successfully to the solution of deterministic shortest-path Markov decision processes. Here, we propose improved value iteration algorithms based on Dijkstra's algorithm for solving shortest path Markov decision processes. The experimental results on a stochastic shortest-path problem show the feasibility of our approach.*

INTRODUCTION

In planning with uncertainty, the planner's objective is to find a policy that optimizes some expected utility. Most approaches for finding such policies are based on decision-theoretic planning (Boutilier, Dean, and Hanks 1999; Bellman 1954; Puterman 1994). Among these, Markov decision processes (MDPs) constitute a mathematical framework for modeling and deriving optimal policies. Value iteration is a dynamic programming algorithm (Bellman 1957) for solving MDPs, but it is usually not considered because of its slow convergence (Littman, Dean, and Kaelbling

This work has been partially supported by Consolider Ingenio 2010 CSD2007-00022, Spanish Government Project MICINN TIN2011-27652-C03-01, and Valencian Government Project Prometeo 2008/051.

Address correspondence to M. G. Garcia-Hernandez, Universidad de Guanajuato, Comunidad de Palo Blanco s/n, Z. C. 36885, Salamanca, Guanajuato, Mexico. E-mail: garciag@ugto.mx

1995). This is because its speed of convergence depends strongly on the order of the computations (or backups).

The slow convergence of value iteration for solving large MDPs is usually approached in one of two ways (Dai and Goldsmith 2007a): heuristic search (Hansen and Zilberstein 2001; Bhuma and Goldsmith 2003; Bonet and Geffner 2003a, b, 2006), or prioritization (Moore and Atkeson 1993; Ferguson and Stentz 2004; Dai and Goldsmith 2007b; Wingate and Seppi 2005). In the first case, heuristic search (combined with dynamic programming) is used to reduce the number of relevant states as well as the number of search expansions. Hansen and Zilberstein (2001) considered only part of the state space by constructing a partial solution graph, searching implicitly from the initial state toward the goal state, and expanding the most promising branch of an MDP according to an heuristic function. Bhuma and Goldsmith (2003) extended this approach by using a bidirectional heuristic search algorithm. Bonet and Geffner (2003a,b) proposed two other heuristic algorithms that use a clever labelling technique to mark irrelevant states. Later on, they explored depth-first search for the solution of MDPs (Bonet and Geffner 2006). In the second case, prioritization methods are based on the observation that, in each iteration, the value function usually changes only for a reduced set of states. So, they prioritize each backup in order to reduce the number of evaluations (Moore and Atkeson 1993; Dai and Goldsmith 2007b). Ferguson and Stentz (2004) proposed another prioritization method called focused dynamic programming, where priorities are calculated in a different way than in prioritized sweeping. Dai and Goldsmith (2007a) extended Bhuma and Goldsmith's idea (2003) by using concurrently different starting points. In addition, they also proposed (Dai and Goldsmith 2007b) a topological-value iteration algorithm, which groups together states that are mutually and causally related in a meta state for the case of strongly connected states (or MDPs with cyclic graphs). Likewise, other approaches such as topological sorting (Wingate and Seppi 2005) and shortest path methods (McMahan and Gordon 2005a,b) have been proposed. On the one hand, topological sorting algorithms can be used to find good backup orderings, but their computational cost is usually high (Wingate and Seppi 2005). On the other hand, shortest path methods have been applied to the solution of MDPs with some success (McMahan and Gordon 2005a,b).

We consider the problem of finding an optimal policy in a class of positive MDPs with absorbing terminal states, which are equivalent to stochastic-shortest-path problems (Bertsekas 1995). McMahan and Gordon (2005a) proposed a method called improved prioritized sweeping (IPS) for solving single-goal, stochastic shortest-path problems based on Dijkstra's algorithm. The advantages of this method are the reduction to Dijkstra's algorithm for the case of Markov chains (or acyclic deterministic MDPs), and the improvement in speed when compared with other methods such as prioritized sweeping (Moore and Atkeson 1993) and focused dynamic programming (Dai and Goldsmith 2007b). Unfortunately, IPS has no guaranteed convergence to the optimal policy for the case of stochastic shortest-path MDPs (Dai and Goldsmith 2007a,b; Li, 2009). Thus, in this work, we propose a new, prioritized-value-iteration algorithm based on Dijkstra's algorithm; this new algorithm has guaranteed convergence for the case of stochastic shortest-path problems and can deal with multiple goal and start states.

This article is organized as follows: first, we present a brief introduction to MDPs, as well as solution methods; then, we describe the prioritized sweeping approaches; we next describe our algorithms and present experimental results on sailing strategies (Vanderbei 1996), a stochastic shortest-path problem. Finally, we present the conclusions.

ABOUT MARKOV DECISION PROCESSES

Markov decision processes (MDPs) provide a mathematical framework for modeling sequential decision problems in uncertain dynamic environments (Bellman 1957; Puterman 2005).

Formally, an MDP is a four-tuple (S, A, P, R), where S is a finite set of states $\{s_1, \ldots, s_n\}$, A is a finite set of actions $\{a_1, \ldots, a_n\}$, and $P: S \times A \times S \rightarrow [0, 1]$ is the transition probability function, which associates a set of probable next states to a given action in the current state. The transition probability to reach state s', if one applies action a in state s, is denoted by $P(a, s, s')$. The reward obtained if one applies action a in state s is denoted by $R(s, a)$. A policy (or strategy) is denoted by $\pi : S \rightarrow A$: it yields an action for each state; it is a rule that specifies which action should be taken in each state. The *Markovian property* guarantees that s' depends only on the pair (s, a). The core problem of MDPs is to find the optimal policy to maximize the expected total reward (Puterman 2005). The value function in stage t, which is the expected reward (or utility) when starting at state s and following policy π, is given by:

$$V^{\pi}(s) = E\left[\sum_{t} \gamma^{t} R(s_t, \pi(s_t)) | s_0 = s\right], \tag{1}$$

where $\gamma \in [0, 1]$ is a discount factor, which may be used for decreasing exponentially future rewards. For the case of discounted MDPs $(0 < \gamma < 1)$, the utility of an infinite state sequence is always finite. So, the discount factor expresses that future rewards have less value than current rewards (Russell and Norvig 2004). For the case of additive MDPs $(\gamma = 1)$ and infinite horizon, the expected total reward may be infinite and the agent must be guaranteed to end up in a terminal state.

Let $V^*(s)$ be the optimal value function given by:

$$V^*(s) = \max_{\pi} V^{\pi}(s). \tag{2}$$

The optimal value function satisfies the Bellman equation (Bellman 1954; Puterman 2005) that is given by:

$$V_t^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V_{t-1}^*(s') \right\}. \tag{3}$$

Value iteration, policy iteration and linear programming are three of the most well-known techniques for finding the optimal value function $V^*(s)$ and the optimal policy π^* for infinite horizon problems (Chang and Soo 2007). However, policy iteration and linear programming are computationally expensive techniques when dealing with problems with large state spaces, because they both require solving several linear systems (of equations) of the same size as the state space. In contrast, value iteration avoids this problem by using a recursive approach typically used in dynamic programming (Chang and Soo 2007).

Starting from an initial value function, value iteration applies successive updates to the value function for each $s \in S$ by using:

$$\hat{V}(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V(s') \right\}. \tag{4}$$

Let $\{V_n | n = 0, 1, \ldots\}$ be the sequence of value functions obtained by value iteration. Then, it can be shown that every value function obtained by value iteration satisfies $|V_n - V^*| \le \gamma^n |V_0 - V^*|$. Thus, from the Banach fixed-point theorem, it can be inferred that value iteration converges to the optimal value function $V^*(s)$. One advantage of value iteration comes from the fact that the value functions obtained can be used as bounds for the optimal value function (Tijms 2003).

The computational complexity of one update of value iteration is $O(|S|^2|A|)$. However, the number of required iterations can be very large. Fortunately, it has been shown in Littman, Dean, and Kaelbling (1995) that an upper bound for the number of iterations n_{it} required by value iteration to reach an ε-optimal solution is given by:

$$n_{it} \le \frac{b + \log\left(\frac{1}{\varepsilon}\right) + \log\left(\frac{1}{1-\gamma}\right) + 1}{1-\gamma}, \tag{5}$$

where $0 < \gamma < 1$, b is the number of bits used to encode rewards and state transition probabilities, and ε is the threshold of the *Bellman error* (Puterman 2005) given by:

$$B_t(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s, a, s') V_t(s') \right\} - V_t(s). \tag{6}$$

The convergence of value iteration may be quite slow for γ close to one. For this reason, several improvements to value iteration have been proposed (Puterman 2005). For instance, common techniques may improve convergence rate, reduce the time taken per iteration, and/or use better stopping criteria.

One of the easiest ways to improve convergence rate is to update the value functions as soon as they become available (also known as asynchronous updates). For instance, Gauss-Seidel value iteration uses the following update equation (Puterman 2005):

$$V^t(s) = \max_{a} \left\{ R(s, a) + \gamma \sum_{s' < s} P(s, a, s') V^t(s') + \gamma \sum_{s' \geq s} P(s, a, s') V^{t-1}(s') \right\}. \tag{7}$$

It is well known that policy iteration converges in fewer number of iterations than value iteration does, but it is more expensive per iteration because it requires solving a system of linear equations at each one of the iterations. In contrast, value iteration does not require the solution of any linear system of equations. A combined approach (modified policy iteration) can exploit the advantages of both. Thus, modified policy iteration uses a partial policy evaluation step based on value iteration (Puterman 2005).

Another way of improving the convergence rate, as well as the iteration time, is using prioritization and partitioning (Wingate and Seppi 2005). Generally, prioritization methods are based on the observation that, at each iteration, the value function usually changes only for a reduced set of states. Thus, by restricting the computation to only those states, a reduction of the iteration time is expected. It has been outlined that for acyclic problems, the ordering of the states, where the transition matrix becomes triangular, may result in a significant reduction in time (Wingate and Seppi 2005).

Another method to reduce the iteration time is to identify and eliminate suboptimal actions (Puterman 2005). For instance, bounds of the optimal value function can be used to eliminate suboptimal actions. The advantage of this approach is that the action set is progressively reduced with the consequent reduction in time.

In another way, the number of iterations can be slightly reduced by using improved stopping criteria based on tighter bounds of the Bellman error (see Equation (6); Puterman 2005). For instance, a stopping criterion would be to stop value iteration when the span of the Bellman error falls below a certain threshold (Puterman 2005).

Finally, for the case of large MDPs with sparse transition matrices, memory savings can be obtained by using a *sparse* representation (Agrawal and Roth 2002) where only nonzero transition probabilities are stored. In this way, it is possible to handle larger problems than those that can be solved otherwise (mainly in highly sparse MDPs). For instance, an adjacency list containing all of the state transitions with nonzero probabilities can be built.

PRIORITY-BASED METHODS FOR SOLVING MDPS

Although value iteration is a powerful algorithm for solving MDPs, it has some potential problems. First, some backups are useless because not all states change in a given iteration (Dai and Goldsmith 2007b). Second, backups are not performed in an optimal order. Priority-based methods such as prioritized sweeping (PS) (Moore and Atkeson 1993) avoid these problems by ordering and performing backups so as to perform the least number of backups (Dai and Goldsmith 2007b). To be more precise, PS maintains a priority queue for ordering backups intelligently. This priority queue is updated as the algorithm sweeps through the state space. PS can begin by inserting the goal state in the priority queue when it is used in an offline dynamic programming algorithm, such as value iteration. 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 of state s is greater than some threshold value ε, or if s is the goal state, then PS inserts its predecessors into the queue according to their priority (Dai and Goldsmith 2007b). Unfortunately, the use of a priority queue for all the states of the model may result in an excessive overhead for real-world problems (Wingate and Seppi 2005), especially for cyclic MDPs.

Focused dynamic programming (Ferguson and Stentz 2004) is another variant of prioritized sweeping, which exploits the knowledge of the start state to focus its computation on states that are reachable from that state. To do this, focused dynamic programming uses a priority metric that it is defined using two heuristic functions: an admissible estimate of the expected cost for reaching the current state from the start state and an estimate of the expected cost for reaching the goal state from the current state. In contrast to other forms of prioritized sweeping, this approach removes the state with the lowest priority value from the priority queue instead of removing the state with the highest priority value, because it is interested in states through which the shortest path passes.

Dibangoye, Chaib-draa, and Mouaddib (2008) proposed an improved topological value iteration algorithm (iTVI), which uses a static backup order. Instead of minimizing the number of backups per iteration or eliminating useless updates, this algorithm attempts to minimize the number of iterations by using a good backup order (a topological order). First, depth-first search is used to collect all reachable states from the start state. Next, breadth-first search is used to build a metric $d(s)$, which is defined as the distance from the start state to state s. A static backup order is built from the resulting metric in such a way that states that are closer to the start state be updated first. The algorithm is guaranteed to converge to the optimal value function because it updates all states recursively in the same way value iteration does.

Meuleau, Brafman, and Benazera (2006) solved stochastic oversubscription planning problems (SOSPs) by means of a two-level hierarchical model. They exploit this hierarchy by solving a number of smaller factored MDPs. Shani, Brafman, and Shimony (2008) extended the use of prioritization to partially observable Markov decision processes. In this case, backups are prioritized by using the Bellman error as a priority metric, and no priority queue is used.

In contrast with the above methods, it is worth mentioning a prioritization method that does not require a priority queue (Dai and Hansen 2007), instead, it uses a first input, first output (FIFO) queue if the backward traversal of the policy graph is breadth first (forward value iteration), or a last input, first output (LIFO) queue if the backward traversal is depth first (backward value iteration). In both cases, unnecessary backups can be avoided by using a labeling technique (Bonet and Geffner 2003a,b) and the decomposition of the state space into a number of strongly connected components. Unfortunately, it has been shown that the backup order induced by these algorithms is not optimal (Dai and Hansen 2007).

Because the performance of PS depends on the priority metric that is used to order states in the priority queue, several researchers have investigated alternative priority metrics. For instance, IPS (McMahan and Gordon 2005a,b) uses a combination of priority metrics (a value-change metric, and an upper-bound metric). In fact, it has been shown that IPS may outperform other prioritized sweeping algorithms (Dai and Goldsmith 2007a,b).

PROPOSED ALGORITHMS

Dijkstra's algorithm is an efficient greedy algorithm for solving the single-source, shortest-path problem in a weighted acyclic graph. This algorithm is a special case of the A* algorithm, but unlike the A* algorithm, Dijkstra's algorithm is not goal oriented. This is because Dijkstra's algorithm computes all shortest paths from a single source node to all nodes and thus solves the one-to-all shortest-path problem. In fact, it has been shown that Dijkstra's algorithm is a successive approximation method to solve the dynamic programming equation for the shortest-path problem (Sniedovich 2006, 2010), and therefore it is based on the Bellman's optimality principle. The main difference between Dijkstra's algorithm and other dynamic programming methods for the shortest-path problem is the particular order in which it processes states; it processes states according to a greedy-best-first rule. More precisely, Dijkstra's algorithm chooses the state having the smallest value of the dynamic programming functional to be the next state to be processed. One implication for the solution of MDPs is that, instead of a topological order, a more suitable update order may be to choose the state having the highest value function as the next state to be updated. For that reason, in our algorithms we use the current value function as a priority metric.

One of the advantages of value iteration and its variants (in particular PS) is that their convergence to the optimal value function is guaranteed for the case of discounted MDPs and for the case of additive MDPs with absorbing states (Bertsekas 1995; Li 2009). This is because successive applications of the Bellman equation guarantee convergence to the optimal value function. For that reason, in our algorithms we update all predecessors of the best state (having the highest value function) by using the Bellman equation.

Let $V^t(s)$ be the expected cost at time t to reach a goal state starting from a state $s \in S$; $\pi^t(s)$ be the best policy or action at time t and state s; $R(s, a)$ be the reward for action $a \in A$ in state$s$; $L = \{(s_k, s'_k.a_k, p_k) | p_k = P(s'_k | s_k, a_k) \neq 0\}$ be the set of all the possible state transitions; G be the set of goal states; γ be the discount factor; and ε be the maximum error. Basically, our method performs an initialization step followed by successive prioritized removals of each state in the queue with an update of its predecessors by using the Bellman equation until the queue is empty.

As shown in Algorithm 1, for each state $s \in S$, we make $\pi^0(s) = -1$, then, if $s \notin G$, we assign a very large positive constant M to $V^0(s)$; otherwise we set its value to zero. Next, we push each goal state $s \in G$ into the priority queue according to its priority $V^0(s)$. Then, we repeat the following procedure until the priority queue is empty. We pop the state s with the highest priority out from the queue, and then we update all its predecessors $y \in pred(s)$. For every update of a predecessor of state s, we compute the Bellman equation

$$V^{t+1}(y) = \max_a \left\{ R(y, a) + \gamma \sum_{\forall\left(s_k = y, s'_k, a_k = a, p_k\right) \in L} p_k V^t\left(s'_k\right) \right\}, \tag{8}$$

and if $|V^{t+1}(y) - V^{t}(y)| > \varepsilon$, then we push state y into the queue according to its priority $V^{t+1}(y)$, if state y is already in the queue, then its priority is only updated.

As we have previously said, the convergence speed of value iteration depends on a good state ordering, so we have devised several ways of updating the predecessors in our algorithms. In Algorithms 1, Gauss-Seidel Improved Prioritized Value Iteration (IPVI), and 2, Jacobi Improved Prioritized Value Iteration (JIPVI), we update asynchronously all the predecessors of each state, disregarding their ordering. In Algorithm 3, Prioritized Predecessors Improved Prioritized Value Iteration (PPIPVI), we update asynchronously all the predecessors of each state, but updates are prioritized according to the value function of each predecessor.

THE SAILING PROBLEM

For the validation of the proposed algorithm, we chose the sailing strategies problem (Vanderbei 1996, 2008), which is a finite state-action-space, stochastic shortest-path problem, in which a sailboat has to find the shortest path between two points on a lake under fluctuating wind conditions.

The details of the problem are as follows: the sailboat's position is represented as a pair of coordinates on a grid of finite size. The sailor has eight actions that direct toward a neighboring grid position. Each action has a cost (required time) depending on the direction of the boat's heading and the wind. For the action that sets a direction just opposite that of the wind, the respective cost must be high. For example, if the wind is at 45 degrees as measured from the boat's heading (*upwind* tack), it requires four seconds to sail from one waypoint to one of the nearest neighboring points. But, if the wind is at 90 degrees from the boat's heading (*crosswind* tack), the boat moves faster through the water and can reach the next waypoint in only three seconds. If the wind is a quartering tailwind (*downwind* tack), it requires just two seconds. Finally, if the boat is sailing directly downwind (*away* tack), it requires only one second.

The wind can hit the left or right side of the boat (a *port* or a *starboard* tack, respectively). When changing from a port to a starboard tack (or vice versa), three seconds (*delay*) are wasted. To keep our model simple, we assume that the wind velocity is constant but its direction may change at any time. The wind could blow from one of three directions: it could either remain blowing from the same direction or come at 45 degrees from the left or from the right. Table I shows the probabilities of a change in the wind direction as used in all the experiments. When the heading is along one of the diagonal directions, the time is multiplied by $\sqrt{2}$ to account for the somewhat longer distance that must be traveled. Each state s of

TABLE 1 Probability of Wind Direction Change

	N	NE	E	SE	S	SW	W	NW
N	0.4	0.3						0.3
NE	0.4	0.3	0.3					
E		0.4	0.3	0.3				
SE			0.4	0.3	0.3			
S				0.4	0.2	0.4		
SW					0.3	0.3	0.4	
W						0.3	0.3	0.4
NW	0.4						0.3	0.3

First column indicates old wind direction and first row indicates new wind direction.

the MDP corresponds to a position of the boat (x, y), a tack $t \in \{0, 1, 2\}$ and a wind direction $w \in \{0, 1, \ldots, 7\}$.

All of the experiments were performed on a 2.66 GHz Pentium D computer with 2 GB RAM running Windows XP. All of the tested algorithms were implemented using the Java language. The initial and maximum size of the stack of the Java virtual machine was set to 1024 MB and 1536 MB, respectively. For all of the experiments, we set $\varepsilon = 10^{-7}$ and $\gamma = 1$. So, we are dealing with an additive MDP, where convergence is not guaranteed by the Banach fixed-point theorem (Blackwell 1965). Fortunately, the presence of absorbing states (states with zero reward and 100% probability of staying in the same state) may allow the algorithm to converge (Hinderer and Waldmann 2003). The lake size was varied from (50×50) to (260×260), and the resulting number of states varied from 55,296 to 1,597,536, respectively (without the bounding beaches). We repeated each run 10 times, and then we calculated the mean and standard deviation of the solution time.

EXPERIMENTAL RESULTS

We tested our algorithms (IPVI, JIPVI, and PPIPVI) and several variants of value iteration including different acceleration techniques (Puterman 2005): Gauss-Seidel Value Iteration (GSVI); Gauss-Seidel Value Iteration with updates of only those states (as well as those of their neighbors) whose value function changed in the previous iteration (GSVI2); and Gauss-Seidel Value Iteration with the same acceleration procedures as GSVI2 plus static reordering of the states in decreasing order of maximum reward (GSVI3). Also two other algorithms were tested: the Vanderbei's dynamic-programming (VDP) approach (Vanderbei 1996, 2008) and the improved topological value iteration (iTVI) (Dibangoye, Chaib-draa, and Mouaddib 2008). We also tested IPVI with different priority metrics, but it converged to the optimal policy only for the priority metric suggested by Dijkstra's algorithm.

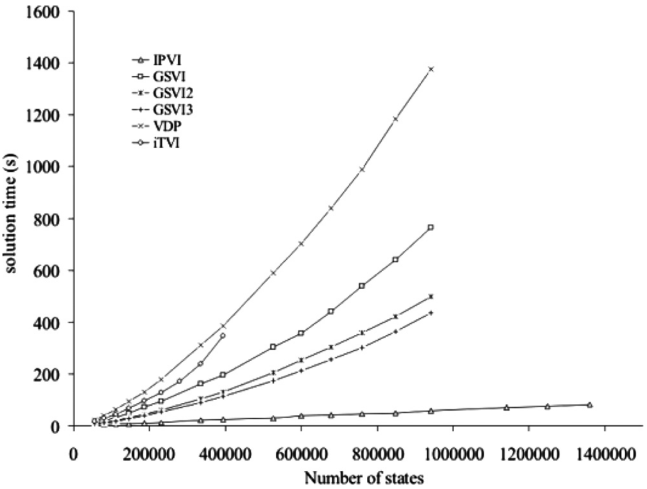

FIGURE 1 Solution time as a function of the number of states for IPVI, three accelerated variants of VI, VDP, and iTVI.

Figure 1 shows the solution time for all tested algorithms as a function of the number of states. As we can see, IPVI yielded the lowest solution time, whereas VDP yielded the highest solution time. For instance, for 525,696 states, our algorithm took 29.9 seconds, whereas GSVI3 took 173.6 seconds, GSVI2 took 203.3 seconds, GSVI took 303.6 seconds and VDP took 589.5 seconds. In this case, our algorithm was 5.8 times faster than GSVI3, 6.8 times faster than GSVI2, 10.2 times faster than GSVI, and 19.7 times faster than VDP. For 940,896 states, our algorithm took 57.4 seconds, whereas GSVI3 took 412 seconds, GSVI2 took 486.5 seconds, GSVI took 755.1 seconds and VDP took 1376.6 seconds. In this case, our algorithm was 7.2 times faster than GSVI3, 8.5 times faster than GSVI2, 13.2 times faster than GSVI, and 24 times faster than VDP. For 1,359,456 states, our algorithm took 81.5 seconds. As we can see, iTVI (Dibangoye, Chaib-draa, and Mouaddib 2008) was not tested for more than 400,000 states because it exhausted the memory resources.

Figure 2 shows a closer look at the solution time as a function of the number of states for IPVI, VI with asynchronous updates (GSVI) and iTVI. For instance, for 393,216 states, our algorithm took 22.2 seconds, whereas GSVI took 195.4 seconds and iTVI took 347.6 seconds. In this case, our algorithm was 8.8 times faster than GSVI and 15.7 times faster than iTVI.

Figure 3 shows the solution time for different strategies of performing predecessor updates. The best strategies were based on IPVI, whereas the worst strategies were based on PPIPVI. In this case, it is clear that prioritizing predecessor updates did not result in a better solution time because of

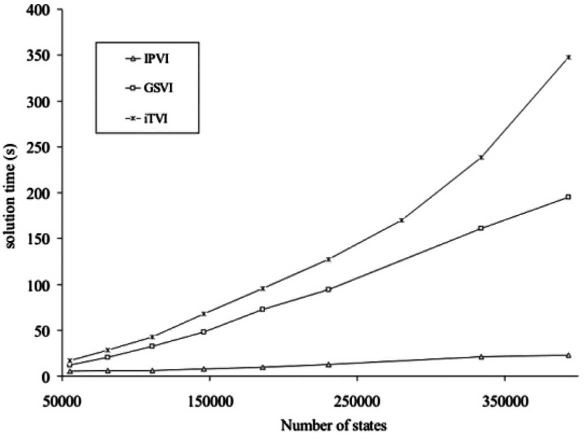

FIGURE 2 Closer look at the solution time as a function of the number of states for IPVI, GSVI, and iTVI.

the overhead involved. In fact, the number of updates for both algorithms was almost the same.

Figure 4 shows the solution time as a function of the number of states for IPVI and JIPVI. Both algorithms achieved a good solution time, but JIPVI was slightly slower.

Figure 5 shows the number of updates as a function of the number of states for IPVI. As we can see, the number of updates is nearly a linear function of the number of states. As a matter of fact, the number of updates performed by IPVI was almost equal to the number of transitions of the MDP.

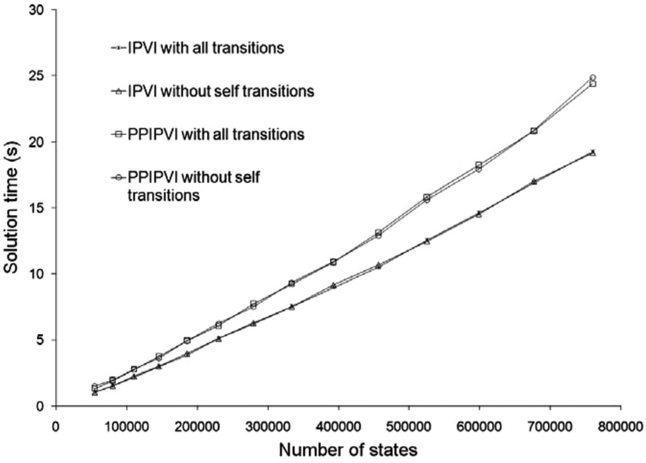

FIGURE 3 Comparison of different strategies for predecessor updating.

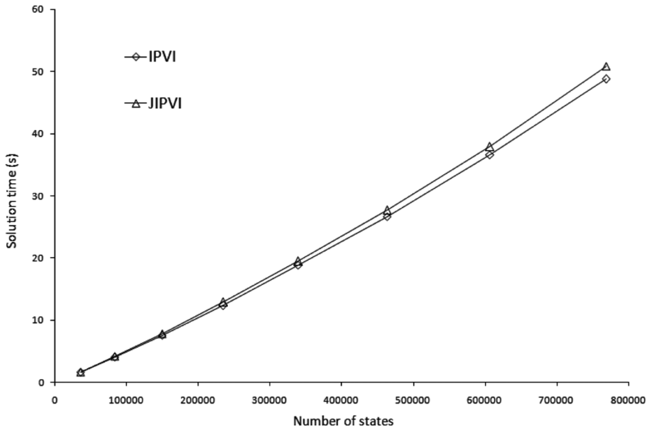

FIGURE 4 Solution time versus number of states for IPVI and JIPVI.

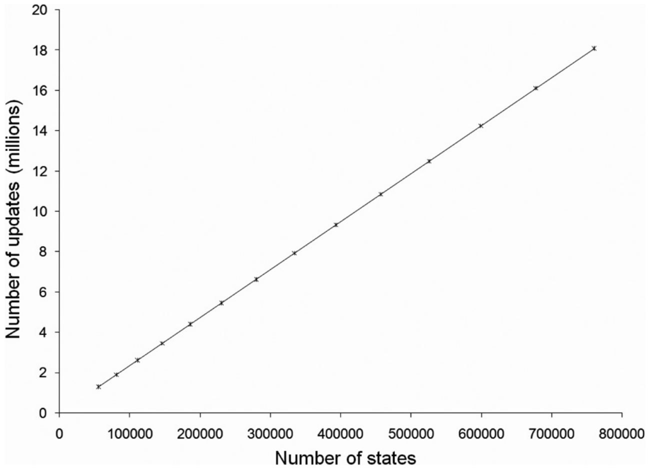

FIGURE 5 Number of predecessor updates versus number of states for IPVI.

CONCLUSIONS

In this article we have proposed and tested new prioritized value iteration algorithms, based on Dijkstra's algorithm, for solving the sailing strategies problem, a stochastic shortest-path MDP. Unlike other prioritized approaches such as IPS, our approach can deal with multiple start and goal states, and because it successively updates each state by using the Bellman equation, it has guaranteed convergence to the optimal solution. In addition, our algorithms use the current value function as a priority metric because Dijkstra's algorithm suggests that a more suitable update order is given by the value of the dynamic programming functional.

We compared the performance of our algorithms with other state-of-the-art algorithms including different acceleration techniques. At least in the sailing problem, our approach was the fastest, and it has guaranteed convergence to the optimal value function. We compared different ways of performing predecessor updates. IPVI and JIPVI were the fastest algorithms. At least in our experiments, prioritization of predecessor updates (as in PPIPVI) did not result in a better solution time. In fact, prioritization of predecessor updates yielded a very small reduction in the number of updates. The number of updates performed by our algorithms was nearly a linear function of the number of transitions of the MDP.

REFERENCES

Agrawal, S., and D. Roth. 2002. Learning a sparse representation for object detection. In *Proceedings of the 7th European conference on computer vision*, 1–15. Copenhagen, Denmark: Springer.

Bellman, R. E. 1954. The theory of dynamic programming. *Bulletin of the American Mathematical Society* 60:503–516.

Bellman, R. E. 1957. *Dynamic programming*. Princeton, NJ, USA: Princeton University Press.

Bertsekas, D. P. 1995. *Dynamic programming and optimal control*. Belmont, MA, USA: Athena Scientific.

Bhuma, K., and J. Goldsmith. 2003. Bidirectional LAO* algorithm. In *Proceedings of the 1st Indian International conference on artificial intelligence*, 980–992. Hyderabad, India: IICAI.

Blackwell, D. 1965. Discounted dynamic programming. *Annals of Mathematical Statistics* 36:226–235.

Bonet, B. and H. Geffner. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proceedings of the 18th international joint conference on artificial intelligence*, 1233–1238. Acapulco, México: Morgan Kaufmann.

Bonet, B., and H. Geffner. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the international conference on automated planning and scheduling*, 12–21. Trento, Italy: AAAI Press.

Bonet, B., and H. Geffner. 2006. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings and its application to MDP. In *Proceedings of the 16th international conference on automated planning and scheduling*, 142–151. Cumbria, UK: AAAI Press.

Boutilier, C., T. Dean, and S. Hanks. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Chang, I., and H. Soo. 2007. *Simulation-based algorithms for Markov decision processes. Communications and Control Engineering*. London, UK: Springer Verlag London Limited.

Dai, P., and J. Goldsmith. 2007a. *Faster dynamic programming for Markov decision processes* (Technical Report, Doctoral Consortium, Department of Computer Science and Engineering, University of Washington).

Dai, P., and J. Goldsmith. 2007b. Topological value iteration algorithm for markov decision processes. In *Proceedings of the 20th international joint conference on artificial intelligence*, 1860–1865. Hyderabad, India: Morgan Kaufmann Publishers.

Dai, P., and E. A. Hansen. 2007. Prioritizing Bellman backups without a priority queue. In *Proceedings of the 17th international conference on automated planning and scheduling*, 113–119. Providence, RI, USA: Association for the Advancement of Artificial Intelligence.

Dibangoye, J. S., B. Chaib-draa, and A.-I. Mouaddib, 2008. A novel prioritization technique for solving Markov decision processes. In *Proceedings of the 21st international FLAIRS (The Florida artificial intelligence research society) conference*, 537–542. Coconut Grove, FL, USA: AAAI Press.

Ferguson, D., and A. Stentz. 2004. Focused propagation of MDPs for path planning. In *Proceedings of the 16th IEEE international conference on tools with artificial intelligence*, 310–317. Boca Raton, FL, USA: IEEE Computer Society.

Hansen, E. A., and S. Zilberstein. 2001. LAO: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.

Hinderer, K., and K. H. Waldmann. 2003. The critical discount factor for finite Markovian decision processes with an absorbing set. *Mathematical Methods of Operations Research*, 57:1–19. Heidelberg, Germany: Springer–Verlag.

Li, L., 2009. *A unifying framework for computational reinforcement learning theory* (PhD Thesis, The State University of New Jersey, New Brunswick, NJ, USA).

Littman, M. L., T. L. Dean, and L. P. Kaelbling. 1995. On the complexity of solving Markov decision problems. In *Proceedings of the 11th international conference on uncertainty in artificial intelligence*, 394–402. Montreal, Quebec: Morgan Kaufmann Publishers.

Meuleau, N., R. Brafman, and E. Benazera. 2006. Stochastic over-subscription planning using hierarchies of MDPs. In *Proceedings of the 16th international conference on automated planning and scheduling*, 121–130. Cumbria, UK: AAAI Press.

McMahan, H. B., and G. Gordon. 2005a. Fast exact planning in Markov decision processes. In *Proceedings of the 15th international conference on automated planning and scheduling*, 151–160. Monterey, CA, USA: AAAI Press.

McMahan, H. B., and G. Gordon. 2005b. *Generalizing Dijkstra's algorithm and Gaussian elimination for solving MDPs* (Technical Report, Carnegie Mellon University, Pittsburgh, PA, USA).

Moore, A., and C. Atkeson. 1993. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning* 13:103–130.

Puterman, M. L. 1994. *Markov decision processes —Discrete stochastic dynamic programming*. New York, USA: John Wiley & Sons.

Puterman, M. L. 2005. *Markov decision processes: Discrete stochastic dynamic programming*. Wiley Series in Probability and Statistics. New York, USA: John Wiley & Sons.

Russell, S., and P. Norvig. 2004. "Making Complex Decisions" (chap.17). In *Artificial intelligence: A modern approach*. 2nd ed. Berkeley, CA, USA: Pearson Prentice Hall.

Shani, G., R. Brafman, and S. Shimony. 2008. Prioritizing point-based POMDP solvers. *IEEE Transactions on Systems, Man. and Cybernetics* 38 (6): 1592–1605.

Sniedovich, M. 2006. Dijkstra's algorithm revisited: The dynamic programming connexion. *Control and Cybernetics* 35:599–620.

Sniedovich, M. 2010. *Dynamic programming: Foundations and principles*. 2nd ed. In *Pure and Applied Mathematics Series*. Melbourne, Australia: Taylor & Francis.

Tijms, H. C. 2003. "Discrete-time Markov decision processes," chap. 6. In *A first course in stochastic models*. Chichester, UK: John Wiley & Sons.

Vanderbei, R. J. 1996. *Optimal sailing strategies. Statistics and Operations Research Program*, University of Princeton, USA. http://orfe.princeton.edu/~rvdb/sail/sail.html (accessed June 6, 2010).

Vanderbei, R. J. 2008. *Linear programming: Foundations and extensions*. 3rd ed. Princeton, NJ, USA: Springer Verlag.

Wingate, D., and K. D. Seppi. 2005. Prioritization methods for accelerating MDP solvers. *Journal of Machine Learning Research* 6:851–881.

Algorithm 1

Gauss-Seidel Improved Prioritized Value Iteration (IPVI)

```
IPVI($R$, $L$, $S$, $G$, $\gamma$, $\varepsilon$)
($\forall s \in S$) $V(s) \leftarrow M$
($\forall s \in G$) $V(s) \leftarrow 0$
($\forall s \in G$)queue.enqueue($s$, $V(s)$)
while ($\neg$queue.isempty())
  $s \leftarrow$ queue.pop()
  for all $y \in$ pred($s$)
    $V'(y) \leftarrow V(y)$
    $V(y) = \max_a \left\{ R(y, a) + \gamma \sum_{\forall (s_k = y, s'_k, a_k = a, p_k) \in L} p_k V'(s'_k) \right\}$
    if $|V(y) - V'(y)| > \varepsilon$ then
      queue.decreasepriority($y$, $V(y)$)
    end
  end
end
return
```

Algorithm 2

Jacobi Improved Prioritized Value Iteration (JIPVI)

```
JIPVI($R$, $L$, $S$, $G$, $\gamma$, $\varepsilon$)
($\forall \mathrm{s} \in \mathrm{S}$) $V(s) \leftarrow M$
($\forall \mathrm{s} \in \mathrm{G}$) $V(s) \leftarrow 0$
($\forall \mathrm{s} \in \mathrm{G}$) $queue.enqueue(s, V(s))$
while ($\neg queue.isempty()$)
  $s \leftarrow queue.pop()$
  for all $y \in pred(s)$
    $V'(y) \leftarrow V(y)$
    $V(y) = \max_a \left\{ (1 - \gamma p_{yya})^{-1} \left[ R(y, a) + \gamma \sum_{\forall (s_k = y, s'_k \neq y, a_k = a, p_k) \in L} p_k V'(s'_k) \right] \right\}$
    if $|V(y) - V'(y)| > \varepsilon$ then
      queue.decreasepriority($y$, $V(y)$)
    end
  end
end
return
```

Algorithm 3

Prioritized Predecessors Improved Prioritized Value Iteration (PPIPVI)

```
PIPVI($R, L, S, G, \gamma, \varepsilon$)
($\forall s \in S$) $V(s) \leftarrow M$
($\forall s \in G$) $V(s) \leftarrow 0$
($\forall s \in G$)queue.enqueue($s, V(s)$)
while ($\neg$queue.isempty())
  $s \leftarrow$ queue.pop()
  sort predecessors of state s
  for all $y \in$ pred($s$)
    $V'(y) \leftarrow V(y)$
    $V(y) = \max_a \left\{ R(y, a) + \gamma \sum_{\forall (s_k = y, s'_k, a_k = a, p_k) \in L} p_k V'(s'_k) \right\}$
    if $|V(y) - V'(y)| > \varepsilon$ then
      queue.decreasepriority($y, V(y)$)
    end
  end
end
return
```