Partitioned External-Memory Value Iteration

Peng Dai **Mausam** **Daniel S. Weld**
Dept of Computer Science and Engineering
University of Washington
Seattle, WA-98195
{daipeng,mausam,weld}@cs.washington.edu

Abstract

Dynamic programming methods (including value iteration, LAO*, RTDP, and derivatives) are popular algorithms for solving Markov decision processes (MDPs). Unfortunately, however, these techniques store the MDP model extensionally in a table and thus are limited by the amount of main memory available. Since the required space is exponential in the number of domain features, these dynamic programming methods are ineffective for large problems. To address this problem, Edelcamp et al. devised the external memory value iteration (EMVI) algorithm, which uses a clever sorting scheme to efficiently move parts of the model between disk and main memory. While EMVI can handle larger problems than previously addressed, the need to repeatedly perform external sorts still limits scalability. This paper proposes a new approach. We partition an MDP into smaller pieces (blocks), keeping just the relevant blocks in memory and performing Bellman backups block by block. Experiments show that our algorithm is able to solve large MDPs an order of magnitude faster than EMVI.

INTRODUCTION

Markov Decision Processes (MDPs) (Bellman 1957) are a powerful and widely-adopted formulation for modeling probabilistic planning problems (Bonet & Geffner 2000). For instance, NASA researchers use MDPs to model the Mars rover decision making problems (Bresina *et al.* 2002). MDPs are also used to formulate the military operations planning (Aberdeen, Thiébaux, & Zhang 2004) and coordinated multi-agent planning (Musliner *et al.* 2007), *etc.*

Unfortunately, the common algorithms for solving MDPs, *e.g.*, value iteration (Bertsekas 2001), LAO* (Hansen & Zilberstein 2001), and RTDP (Barto, Bradtke, & Singh 1995) are based on dynamic programming and thus require the amount of memory that is polynomial in the number of states, $|S|$, *i.e.*, exponential in the number of domain features. This prohibitive use of memory is a major bottleneck in scaling MDP algorithms to real-world problems. Even relatively compact symbolic representations, like algebraic decision diagrams, tend to grow large in size, exhausting memory on moderate sized MDPs (Hoey *et al.* 1999).

The most common approach to sidestep the memory bottleneck is the use of approximations via compact function representations, such as neural networks, linear combination of basis functions, *etc.* (Gordon 1995; Buffet & Aberdeen 2007). However, these approximate techniques are often unable to capture the probabilistically interesting subtleties of symbolic domains (Little & Thiebaux 2007). Even when a policy capable of reaching the goal is returned, its cost is typically much greater than optimal.

Recently, an alternative solution to scaling up MDPs was proposed: the external memory value iteration (EMVI) algorithm (Edelkamp, Jabbar, & Bonet 2007). This algorithm stores values with transition edges (instead of states) and uses a clever sorting scheme to ensure that all necessary values are appropriately loaded from disk into memory so that Bellman backups can be efficiently computed. Because of this scheme, EMVI has been shown to solve much larger problems than any previous optimal algorithm.

While EMVI is a powerful approach, it suffers from some significant drawbacks. First, because EMVI associates values with transition edges, it duplicates information, using substantial additional disk storage. Also, EMVI needs to perform a costly external-memory sort of this $O(|S|^2)$ array between every iteration of backups. Finally, while EMVI requires very little memory to run, it is not able to exploit all the available memory to improve the runtime performance. In the best case when the whole MDP can fit in the memory EMVI performs significantly worse than value iteration.

In this paper we develop a new algorithm, PEMVI, that alleviates these concerns, by exploiting the observation that the order of backups during value iteration do not matter. PEMVI partitions the state space into disjoint blocks, most of which are stored on disk at any time. In order to backup the states in block p, the values corresponding to neighboring blocks are also loaded; several backups are done on the states in p before moving to the next. By choosing a good partition and selecting a good order in which to back them up, PEMVI achieves an efficient flow of information between states with reduced disk activity. Our experiments show that PEMVI solves problems much larger than what internal memory techniques like value iteration, RTDP can handle. Moreover, it outperforms EMVI by an order of magnitude. For example, we solve one problem which would require several years with EMVI (Edelkamp, Jabbar, & Bonet 2007).

BACKGROUND

A Markov decision process (MDP) is a four-tuple $\langle S, A, T, C\rangle$, where S is a finite set of states, A a finite set of actions, T the probability transition function and C the cost function. An MDP system proceeds in a number of discrete time slots called *stages*. At any stage t, the system is at some state s. When an action $a \in A$ is taken, the system proceeds into stage $t+1$ and changes stochastically into one of a number of *successor states*. The transition is based on the transition function, where $T(s'|s,a)$ is the probability of changing to state s'. A cost $C(s,a)$ is charged for taking the action.

The *horizon* of an MDP is the number of stages over which the cost is accumulated. We focus on the *stochastic shortest-path problem* (SSP), a special type of MDP called *indefinite horizon* MDP, in which the horizon is finite but unknown[1]. It has one more component, $G \subseteq S$, a set of goal states. Any state $g \in G$ is an absorbing state. The costs in an SSP problem are always positive. A *policy* of an MDP is a mapping from states to actions. To solve an SSP problem, we want to find an *optimal policy* that directs the system to transition from any state into a goal state with the minimum expected cumulative costs. To evaluate a policy π, the *value function* $V^\pi(\cdot)$ is defined by:

$$V^\pi(s) = C(s,\pi(s)) + \sum_{s' \in S} T(s'|s,\pi(s))V^\pi(s'). \quad (1)$$

The optimal value function $V^*(\cdot)$ satisfies the following *Bellman Equations*.

$$\begin{aligned} V^*(s) &= 0 \quad \text{if } s \in G \text{ else} & (2)\\ V^*(s) &= \min_{a\in A}[C(s,a) + \sum_{s'\in S} T(s'|s,a)V^*(s')] \end{aligned}$$

To solve the SSP problem we need to find the optimal value function. The optimal policy $\pi^*(\cdot)$ can be computed by one step lookahead using the optimal value function:

$$\pi^*(s) = \operatorname*{argmin}_{a\in A}[C(s,a) + \sum_{s'\in S} T(s'|s,a)V^*(s')] \quad (3)$$

Dynamic Programming Dynamic programming is the popular approach to solve MDPs. The basic idea is to continually update the value function, by using the Bellman equations as assignments, until convergence. We call such an update a *Bellman backup*. Value iteration (VI) is a classic dynamic programming algorithm in which the values of states are initialized at random or guided by some heuristic (Bellman 1957). Then in each iteration, the algorithm backs up every state. The state value change between two consecutive iterations is called the *Bellman error* of that state. Value iteration terminates when the maximum Bellman error is less than a threshold value δ.

Dynamic programming converges in time polynomial in the number of states (Littman, Dean, & Kaelbling 1995; Bonet 2007), but it requires storing the whole MDP model and the value function explicitly in the memory. This can be prohibitive when the model is large. Thus, the size of the main memory acts as a bottleneck on the size of the problems solvable.

[1]Our techniques naturally extend to other MDPs also, *e.g.*, infinite horizon discounted MDP.

External Memory Value Iteration To solve the scalability problem, Edelcamp et al. recently proposed a variant of value iteration called external-memory value iteration (EMVI) (Edelkamp, Jabbar, & Bonet 2007). It is the first MDP algorithm that makes use of disk space to store MDP models. When backups are performed, relevant portions of the model are moved from the disk to the main memory. Note from Equation 2 that to back up a state s one needs the transition function of s, and the values of all its successors in memory. If one perform backups in an arbitrary order, one would thrash with inefficient transfers to and from disk. To overcome this problem, Edelcamp et al. used a clever idea: before every iteration, the algorithm sorts the transition functions $T(s'|s,a)$ according to s', and attaches the current value function $V(s')$ to the corresponding edge. This breaks a backup into several incremental additions, and a final minimization. By this means, EMVI loads each part of the model into memory only once per iteration.

The major drawback of EMVI is its requirement of an additional external sort of the entire model at the start of every iteration — huge overhead. To its credit, EMVI only uses a small amount of main memory, but this is also a weakness. If considerable memory is *available* the EMVI algorithm has no way to exploit the memory to reduce the number of required IOs. In the best case, when the whole MDP can fit in main memory, EMVI performs much slower than VI (Edelkamp, Jabbar, & Bonet 2007).

PARTITIONED EXT-MEMORY VI

Similar to EMVI we propose to overcome the memory bottleneck in value iteration using additional storage in the external memory. However, in contrast to the previous approach, we exploit the benefits of partitioning the state space. Partitioning of an MDP has been studied previously (Wingate & Seppi 2005), but not in the context of external memory algorithm. Our algorithm, partitioned external-memory value iteration (PEMVI), demonstrates its applicability in solving large MDPs much faster.

PEMVI runs in two phases: (1) partition generation, and (2) asynchronous dynamic programming that partitions the state space into blocks and then performs backups on a block by block basis by successively loading all the information necessary to back up each block in the internal memory.

In this paper we focus our implementation and empirical evaluations on the second phase. For now, we generate the partitions semi-automatically. The next section describes a sample routine to generate such partitions and we are in the process of implementing the fully automated partition generator. We hope to investigate different variants of partition generator in the future.

Definitions We define a *partition* of an MDP to be the division of the state space, S, into disjoint *partition blocks*

$P = \{p_0, \dots, p_n\}$ such that $S = \cup_i p_i$. Each block in the partition contains at least one state and each state belongs to exactly one partition block[2]. The function $part(s) : S \rightarrow P$, a surjection from the state space to the partition space, specifies which partition block a given state s is in.

We define the *successors* of a state s to be the set of states that can be reached by non-zero probability from s in one step. This leads us to the important notion of *successor block*. A partition block p' is said to be a successor block of another partition block p if there exist two states s and s' s.t. $part(s) = p$, $part(s') = p'$ and s' is a successor of s. Symmetrically, p' is the *predecessor block* of p. The set of all successor blocks of a partition block p is denoted $succpart(p)$. Similarly, the set of all predecessor blocks of a partition block p is denoted $predpart(p)$.

For each partition block p, we are interested in two components: a transition component and a value component. The transition functions and the cost functions of all states s in the partition block, $\{T_p(\cdot|s,a), C_p(s,a)\}$, form its *transition component*; the current value function $\{V_p(s)\}$ form its *value component*.

External Asynchronous Dynamic Programming Asynchronous DP is a version of dynamic programming in which the value updates occur in a random order instead of a systematic pass through the state space. Value iteration through asynchronous DP preserves all desirable properties under the conditions that all states are eventually backed up enough times.

PEMVI can be seen as a version of asynchronous value iteration in which the backups occur in an order dictated by the partition blocks. The use of external memory comes about in storing the information relevant to each partition block.

PEMVI first computes an ordering of partition blocks. In each iteration PEMVI loads the transition components of the partition block (p), one at a time in the order. It additionally loads the value components of all the successor blocks ($succpart(p)$). Thus after this I/O operation the internal memory contains all the information necessary to perform a Bellman backup for all states in p. PEMVI computes one (or more) backups for this partition block. It then writes the value component of p in the external memory and moves on to the next partition block in the order.

In each iteration we track the maximum Bellman error, e, of the system. Because we update the value of each state at least once in each iteration we know that e will decrease. When e is within our threshold limit (δ) we know that the algorithm has converged and we can terminate. Algorithm 1 presents the pseudo-code for PEMVI.

Because our algorithm is a special case of asynchronous value iteration it has properties similar to value iteration.

Theorem 1 PEMVI *terminates in finite time and converges to a δ-optimal value function as long as the optimal value function is finite.*

[2]Note that our algorithm retains its properties even if a state belongs to multiple partition blocks.

Recall that the number of states is $|S|$ and the total number of state transitions is $|T|$. Let $max_p = \max_{p \in P} |predpart(p)|$. We have the following theorem regarding the I/O complexity of PEMVI:

Theorem 2 *Partitioned External Value Iteration performs at most $O(scan(|T|) + max_p \times scan(|S|))$ I/Os per iteration.*

Proof: In each iteration, PEMVI loads the transition component of each partition block at most once, which requires $O(scan|T|)$ I/Os. To back up a partition block, PEMVI needs to load all its successor block blocks. Thus a particular value component $V_p(s)$, is loaded up to $predpart(p)$ times. This incurs a total of $\sum_p |predpart(p)| \times scan(|V_p|)$ I/Os, which is upper-bounded by $max_p \times scan(|S|)$. □

$|T|$ is usually significantly larger than $|S|$. Moreover, for structured problems, such as ours, the number of successors and predecessors is small, hence an effective partitioning can keep max_p down. Thus, the dominating I/O cost of PEMVI is in the scan of $|T|$. EMVI, on the other hand, requires an external sort of $|T|$ at every iteration. Therefore, PEMVI improves the I/O complexity of EMVI by a logarithmic factor.

Algorithm 1 Partitioned External-Memory Value Iteration

```
1: Input: $S$, $A$, $\lambda$, $\delta$
2: partition $S$ into $P$ using Algorithm 4
3: generate a partition order for $P$
4: repeat
5:   for every partition block $p \in P$ in the order do
6:     load the transition component of $p$
7:     load the value components of every $p' \in succpart(p)$
8:     repeat
9:       numruns$\leftarrow 0$
10:      for every state $s$ in $p$ do
11:        Backup($s$)
12:        compute BellmanError($s$)
13:      numruns$\leftarrow$ numruns+1
14:      BellmanError($p$) $\leftarrow \max_{s \in p}$ BellmanError($s$)
15:    until numruns$> \lambda$ or BellmanError($p$)$< \delta$
16:    write back the value component of $p$
17:    release used memory
18: until ($\max_{p \in P}$BellmanError($p$)$< \delta$)
19:
20: Backup($s$)
21: $V(s) \leftarrow min_a[C(s,a) + \sum_{s'} T(s'|s,a)V(s')]$
```

Single v.s. Multiple Backups Our initial implementation of PEMVI computed a single backup per state in each iteration. However, early experiments confirmed that PEMVI spent most of its time on I/O operations. As a natural extension we started performing multiple backups per iteration — thus increasing message passing, improving amortization of I/O, and speeding convergence in most cases. Note that external memory value iteration is unable to perform multiple backups without I/O, since it has to sort the complete, tabular value function before it is ready to perform each iteration of dynamic programming.

Our mechanism of doing multiple backups is the following: We set an upper bound, λ, on the number of backup runs. In each run, we back up every state in the partition block, and measure the Bellman error. We finish backing up a partition block either when the maximum Bellman error is sufficiently small, *i.e.*, the states have converged for this iteration, or the maximum number of runs are completed. Line 15 of Algorithm 1 describes this in the pseudo-code.

Backup Order In each iteration of PEMVI every partition block is backed up once. For some problems, the order of backing up partition blocks is important, since a good order might make the algorithm converge faster. We explore different heuristics for the backup order.

Our first heuristic (Algorithm 2) attempts to maximize information flow by choosing a back-up order which processes states which are close to the goal before those which are far. We call the partition block which contains the goal state the *goal block*. We compute a "best flow" order that places the partition blocks in the increasing order of their distance from the goal block. We name the reverse of this order as the "worst flow" order. A random order chooses the partition order randomly. For example, consider the partitioning scheme illustrated in Figure 1. If the goal state is in the partition block p_0 then a "best flow" order could be $p_0, p_1, p_4, p_2, p_5, p_8, p_3, p_6, p_9, p_{12}, p_7, p_{10}, p_{13}, p_{11}, p_{14}, p_{15}$. We conjecture the performance of the random order to be between the "best flow" and the "worst flow" order.

Algorithm 2 Backup Order: Maximize Information Flow

```
1: Input: $P$ (set of partition blocks), $p_g$ (goal block)
2: $O \leftarrow \langle p_g \rangle$
3: $P \leftarrow P - \{p_g\}$
4: while $P \neq \emptyset$ do
5:     $p \leftarrow$ first partition block in $O$ where $predpart(p) \nsubseteq O$
6:     $N \leftarrow$ partition blocks in $predpart(p)$ not in $O$
7:     remove each $p' \in N$ from $P$
8:     append each $p' \in N$ at the end of $O$
9: return $O$ as the backup order
```

Our second heuristic for computing the backup order attempts to minimize the total disk read/writes. The next partition block in the order is chosen greedily to be the one that minimizes the I/O at the current step. We start from the goal block and then simulate the I/O operations and the memory contents. Whenever we have to choose the next partition block, we look at all the value components and transition components already in memory. We choose the next partition to be the one that requires minimum I/O. In effect, we will favor partition blocks that have high overlap with the part of the MDP model already in memory, in essence minimizing the I/O required in the current step. Algorithm 3 describes the details of this computation.

To get the benefits of the I/O efficient order we modify line 17 of Algorithm 1 by not releasing a value component when it is needed by the next partition block. Similarly, in line 6, we load only the value components that are not in the memory already.

Algorithm 3 Backup Order: Minimize I/O

```
1: Input: $P$ (set of partition blocks), $p_g$ (goal block)
2: $O \leftarrow \langle p_g \rangle$
3: $P \leftarrow P - \{p_g\}$
4: while $P \neq \emptyset$ do
5:     $p \leftarrow$ the last element in $O$
6:     $L \leftarrow \cup_{\bar{p} \in succpart(p)} \{V_{\bar{p}}\}$
7:     $\hat{p} \leftarrow argmin_{q \in P}(|T_q + C_q| + | \cup_{p' \in succpart(q) - L} \; \{V_{p'}\}|)$
8:     $P \leftarrow P - \{\hat{p}\}$
9:     append $\hat{p}$ at the end of $O$
10: return $O$ as the backup order
```

Algorithm 4 Partition Generation

```
1: input: $S$, $A$
2: $P \leftarrow \{p | p = S\}$
3: while some $p$ in $P$ is invalid do
4:     for all invalid $p$ in $P$ do
5:         for all XOR constraints $R$ applicable in $p$ do
6:             compute $P_R$ by partitioning $p$ according to $R$
7:             compute heuristic($P_R$)
8:         pick $R^*$ with minimum heuristic value
9:         remove $p$ from $P$, add all blocks in $P_{R^*}$ to $P$
10: return $P$
```

GENERATING PARTITIONS

We now describe a mechanical procedure (Algorithm 4) to create effective partitions for PEMVI. While we haven't yet implemented this part of our system, the partitions used for the domains in our experiments can be easily computed using this algorithm.

We are interested in partitions which are refined enough that every block in P can be backed up in main memory. Formally, let P be a partition and let M denote the size of main memory. We say that a partition block, $p \in P$, is *valid* if the its transition component plus the value components of every block in $succpart(p)$ fit in the main memory:

$$|T_p(\cdot|s,a)| + |C_p(s,a)| + \sum_{q \in succpart(p)} |V_q(s)| \leq M$$

We say that a partition, P, is *valid* if $\forall p \in P$, p is valid.

Zhou and Hansen (2006) have studied partitioning in the case of large, deterministic, search problems. They advocate partitioning on the basis of an XOR constraint over a set of mutually exclusive grounded functions. For example, if X is a proposition, then (X XOR $\neg X$) is obviously true. Using this constraint for partitioning will result in two partition blocks, one for each value of X. We adapt Zhou and Hansen's approach to construct more complex constraints by static domain analysis.

In addition, we also deal with domains with numeric variables. For finite-sized problems, a numeric variable (Y) has a well defined lower and upper bound. We split this range at its midpoint (y_{mid}), thus creating an inequality based XOR constraint: ($Y \leq y_{mid}$ XOR $Y > y_{mid}$).

Any partitioning algorithm deals with two opposing forces. One must split the state space into small enough blocks that validity is maintained, but on the other hand one wishes to keep the blocks large so that information flow among the member states can be maximized via multiple backups. Algorithm 4 confronts this tension by recursively choosing the best XOR constraint and splitting along that dimension until reaching a valid partition (Lines 5-9).

This greedy search can be informed by a variety of heuristics (Line 7). For example, we can adapt Zhou and Hansen's notion of *locality* (the maximum number of successors in a partition), since a small value indicates memory efficiency. A complement is *coherence* (the percentage of transitions to states within the partition block); a high value indicates that partition blocks are relatively independent and may converge after only a small number of iterations. An empirical study is needed to determine the best heuristic as well as to evaluate the relative efficacy of greedy vs. systematic search for partitions.

Example: Wet-Floor Domain To illustrate Algorithm 4, we first look at the Wet-Floor domain (Bonet & Geffner 2006). Each problem represents a navigation grid whose cells (states) are slippery with probability 0.4. The only successors of a cell are its neighboring cells, but one doesn't always go in the intended direction. One possible XOR constaint stems from whether or not a cell is slippery, but this has low coherence. Recursively splitting on the value of the x and y coordinates, on the other hand, yields high coherence and low locality.

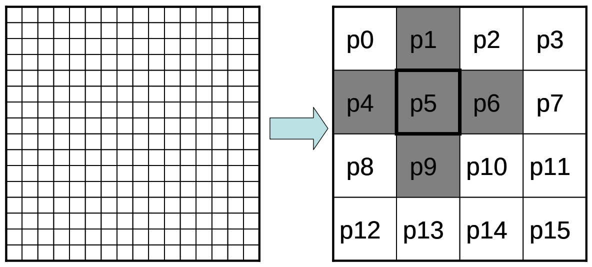

Figure 1: Partitioned MDP for the Wet-Floor domain. The greyed partition blocks on the right side need to be stored in memory in order to back up partition block p_5: the transition component of p_5 and the value components of its successors.

Figure 1 illustrates our partition schema. It shows a 16×16 Wet-Floor problem, which has been grouped into 16 partition blocks, based on x and y coordinates. Each partition block has the same size and thus, the same number of states. Under this partition, the cardinality of $succpart(p)$ for a particular partition block p equals the number of neighboring partition blocks plus one (since p is a successor of itself). So the locality of this partition schema is 5. For example, in order to back up partition block $p5$, we need to load T_{p5}, as well as C_{p1}, C_{p4}, C_{p5}, C_{p6} and C_{p9}. While for value iteration, all 16 transition components and value components must be in memory at all times. Using PEMVI, 15/16 of the space required to store transition components and 11/16 of the space required to store value components can be saved. If memory were so tight that this partition was still invalid, one could achieve additional savings by further increasing the number of partition blocks — the locality of this partitioning scheme does not change with granularity. Of course, for efficiency one generally seeks a valid partition with the smallest number of blocks.

Example: Racetrack Domain The Racetrack domain (Barto, Bradtke, & Singh 1995) is another grid world, which is a popular testbed in reinforcement learning. The car starts stochastically from one of a fixed set of $\langle x, y \rangle$ points with speed 0. At each state, the car can accelerate or decelerate, changing its speed by at most one unit. Clearly, we may partition this domain using the same XOR constraint (x and y coordinates) as in Wet-Floor, but this constraint yields bad locality and bad coherence, because the x and y coordinates can change greatly when the car is moving fast. Instead, we partition on the instantaneous x and y components of velocity. The locality of this partition method is 9 since both v_x and v_y can change by at most one unit per action. By imposing a speed limit of 4 in every directional component, we bound the number of partition blocks by $(4 * 2 + 1)^2 = 81$.

Example: Explosive Blocksworld Domain We choose the first two domains so we could compare with EMVI, but for our third domain, we wanted something unlike a grid-world to illustrate the generality of partitioning. The Explosive Blocksworld from the probabilistic track of IPC-5 (IPC 2006) is a variant of Probabilistic Blocksworld where blocks have a chance of detonation when placed on another object.

The first XOR constraint we choose reflects the constraint that any block (arbitrarily choose b_0) can support at most one other block: $(XOR\ (clear\ b_0)\ (on\ b_1\ b_0)\ \ldots\ (on\ b_n\ b_0))$. Unfortunately, if we use just this constraint, we will end up with a very unbalanced partition, since the size of the partition block matching $(clear\ b)$ has many more states than the others. For problems with many blocks, this partition will not be valid. To solve this problem, we refine the *clear* partition block with a second XOR constraint reflecting the fact that the robot can hold at most one block: $(XOR\ (clearhand)\ (inhand\ b_1)\ \ldots\ (inhand\ b_n))$.[3]

EXPERIMENTS

In our empirical evaluation we wish to answer the following questions: Does PEMVI scale up to problems unsolvable by value iteration? Is PEMVI more efficient than EMVI? What are the best settings for PEMVI? For example, do multiple backups per iteration outperform single backup, how important is choosing an optimal backup order?

We implemented PEMVI and VI in C and evaluated their performance on three domains: Racetrack, Wet-Floor and Ex-Blocksworld as described in the previous section. Similar to Edelkamp et al. the threshold value δ we used for our experiments is 10^{-4}. All experiments were run on a dual-core AMD 2.4GHz with 4GB RAM.

As expected, we verified that VI quickly exhausts available memory on our suite of problems. In all domains PEMVI easily solved problems too big for VI. We also tried labeled RTDP, another optimal algorithm, yet one which is

[3]A block remains clear when it is picked up.

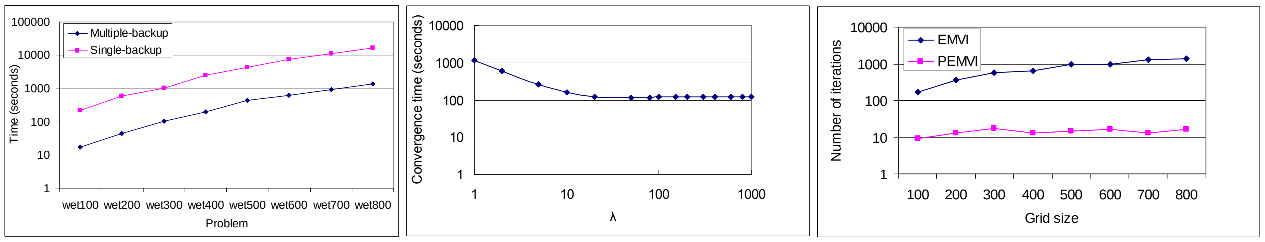

Figure 2: (a) Convergence time for single v.s. multiple backup (b) Convergence time of different λ values on a Wet-Floor 300×300 problem (c) Comparison of number of iterations required by EMVI and PEMVI for problems in Wet-Floor domain.

more scalable due to reachability methods (Bonet & Geffner 2003). However, LRTDP also failed to solve the large problems. For the largest problems, an external memory algorithm like PEMVI is a must. Table 1 reports some of the large problems that PEMVI could solve. We also solved a Wet-Floor problem with 100 million states.

Problem	States (M)	I/O time	CPU time
Wet-Floor	1	36.37	8.46
Racetrack	3.41	314.13	56.17
Racetrack	6.98	523.59	88.90
Ex-Blocksworld	13.14	55.33	13.24
Ex-Blocksworld	13.14	54.23	13.53

Table 1: PEMVI Running time (in minutes) on some large problems in several domains. The number of states are in millions.

Single v.s. Multiple Backups To find the best parameters for PEMVI we compared the single-backup versus multiple-backup settings. We tested eight Wet-Floor problems, whose grid sizes range from 100×100 to 800×800. For small problems we constrained the memory available to the algorithms. The convergence time of the two versions of PEMVI on these problems is plotted in Figure 2(a). We noticed that the multiple-backup version converged an order of magnitude faster than the single-backup version. On average, the single-backup version was 11.83 times slower than multiple-backup version. We conclude that the ability to perform multiple backups in the same iteration without performing additional I/Os is an important feature of this algorithm that can result in significant time savings.

To better characterize this behavior, we looked at the running time spent on each iteration for the case of multiple-backups. The first iteration took over 25% of the computation time with the first five iterations using 60% of the cumulative CPU time. In subsequent iterations, partition blocks converged after very few backups. I/O costs, naturally, were constant across iterations. We also performed a control experiment varying λ, the maximum number of backups per state in an iteration. The convergence times on a Wet-Floor problem with grid size 300×300 are plotted in Figure 2(b). We noticed that when λ changed from 1 to 20, the convergence time dropped significantly, but when λ was greater than 20, running times were not that distinct. So PEMVI is not overly sensitive to λ as long as it is not too small.

While multiple backups got us substantial savings in Wet-Floor problems, the racetrack domain did not benefit from it, since its partitioning is one of low coherence, *i.e.*, partition blocks contain many more external transitions than internal transitions. Often, multiple-backups converged about a factor of 2 slower than single-backup on racetrack problems, which mirrors the findings in (Wingate & Seppi 2005), where some partitions are more effective than the others. This also suggests that different heuristics for the automated partitioning might have significantly different results, and we plan to explore this further.

In all subsequent experiments with multiple backups, we set $\lambda = 100$.

Backup Order We also investigated the importance of the order in which the various partition blocks are backed up. We first evaluated the order produced by Algorithm 2, which aims to maximize information flow. For the eight Wet-Floor problems we computed the ratio of running time for "worst flow" ordering divided by that of "best flow" ordering. For single backups the ratio was only 1.021 with standard deviation of 0.007 — almost no difference. For multiple-backup version, the ratio was 1.320 with s.d. of 0.155 — a larger difference. For both versions, the running time for random ordering lay consistently between the two extreme orderings.

We next evaluated the order generated by Algorithm 3, which tries to minimize I/O. This order on a multiple-backup version gave us a maximum speedup of 10% on the eight Wet-Floor problems, but a larger 15% on the single backup one. Similarly, for the single backup version in a mid-sized racetrack problem we got a speedup of 10% compared to 20% in the single-backups.

Not surprisingly we find that results of backup order are correlated with the ratio between the I/O and computational time for the algorithm. Typically, the single-backup algorithm's I/O time is significantly higher than the computation time, and thus backup order maximizing the information flow is less significant; minimizing I/Os yields better savings. On the other hand, the multiple-backup algorithm is more positively affected by the order which maximizes information flow.

For consistency we use the "best flow" order on all subsequent experiments.

Comparison with EMVI We compared PEMVI and EMVI on two of the probabilistic domains used by Edelkamp et al. in their evaluations — Wet-Floor and Racetrack. We first compared the two algorithms on relatively small problems (under 1 M states). Figure 2(c) compares the number of iterations (and thus, implicitly the amount of I/O required) for both algorithms. In the Wet-Floor domain, EMVI's number of iterations increased linearly (R^2 value 0.986) with the grid perimeter of the problems. In contrast, the number of iterations taken by PEMVI remained stable, well under 20; a vast difference between the algorithms, which we credit to the power of multiple backups.

The results on these relatively small problems were very encouraging so we attempted larger problems. We ran EMVI and PEMVI on two middle-sized Racetrack problems with grid sizes of 75×75 and 50×50. EMVI took 2.5 times as long as PEMVI using single backups and 1.28 times as long as PEMVI, when it used multiple backups ($\lambda = 100$). At least with the current partitioning scheme, PEMVI is not able to derive great benefit from locality in this domain.

We also ran PEMVI on the largest Wet-Floor problem (grid size $10,000 \times 10,000$) mentioned in (Edelkamp, Jabbar, & Bonet 2007). EMVI did not actually solve the problem, but their paper reported an expected running time to convergence of 2 years. PEMVI managed to solve this problem after 62 iterations, taking just under 2 months — an order of magnitude faster than EMVI.

RELATED WORK

Wingate and Seppi (Wingate & Seppi 2005) proposed an algorithm, called prioritized partitioned value iteration (PPVI), to speed up the convergence of value iteration. They also partitioned the state space into a number of partitions, and backed up states on a partition basis. They showed that, by using some clever prioritization metrics, the process of value iteration can be greatly sped up. The focus of PPVI was faster convergence on problems small enough to fit in the main memory. They also share the scalability bottleneck due to limited memory similar to other dynamic programming approaches. We have already explained how our work extends the use of partitioning in structured duplicate detection (Zhou & Hansen 2006) and compared our results with those of external memory value iteration (Edelkamp, Jabbar, & Bonet 2007).

CONCLUSIONS

Most MDP solvers are limited by the size of the main memory and thus only solve small to medium problems. A notable exception is EMVI which uses disk to store the MDP model. However, EMVI is slow, since it requires a complete external memory re-sort in each iteration. This paper presents a novel algorithm, PEMVI. By partitioning a problem's state space, PEMVI may load the MDP model piecemeal into the memory, and perform the backups in an I/O-efficient manner. Our experiments demonstrate that PEMVI can solve problems much too large for internal-memory algorithms. Moreover, PEMVI converges an order of magnitude faster than EMVI, since it has the ability to perform several backups in a single I/O scan and does not require continual sorting.

In the future we plan to complete the implementation of automatic partitioning and evaluate several partitioning heuristics. Lessons from database query optimizers suggest that automated techniques will yield better partitions than the ones we have generated by hand.

ACKNOWLEDGMENTS

We thank Blai Bonet, Stefan Edelkamp, and Shahid Jabber for sharing their program and results with us. We thank Sungwook Yoon for answering the questions on his planner. We also thank Andrey Kolobov and three anonymous reviewers for suggestions on improving the paper. This work is supported by NSF grant IIS-0307906, ONR grant N00014-06-1-0147, and the WRF/TJ Cable Professorship.

References

Aberdeen, D.; Thiébaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *ICAPS*, 402–412.

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *AI J.* 72:81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P. 2001. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 2 edition.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *ICAPS*, 52–61.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, 12–21.

Bonet, B., and Geffner, H. 2006. Learning in depth-first search: A unified approach to heuristic search in deterministic non-deterministic settings, and its applications to MDPs. In *ICAPS*, 142–151.

Bonet, B. 2007. On the speed of convergence of value iteration on stochastic shortest-path problems. *Mathematics of Operations Research* 32(2):365–373.

Bresina, J. L.; Dearden, R.; Meuleau, N.; Ramkrishnan, S.; Smith, D. E.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for AI. In *UAI*, 77–84.

Buffet, O., and Aberdeen, D. 2007. FF+FPG: Guiding a policy-gradient planner. In *ICAPS*, 42–48.

Edelkamp, S.; Jabbar, S.; and Bonet, B. 2007. External memory value iteration. In *ICAPS*, 128–135.

Gordon, G. 1995. Stable function approximation in dynamic programming. In *ICML*, 261–268.

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

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *UAI*, 279–288.

IPC 2006. http://www.ldc.usb.ve/ bonet/ipc5/.

Little, I., and Thiebaux, S. 2007. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*.

Littman, M. L.; Dean, T.; and Kaelbling, L. P. 1995. On the complexity of solving Markov decision problems. In *UAI*, 394–402.

Musliner, D. J.; Carciofini, J.; Goldman, R. P.; E. H. Durfee, J. W.; and Boddy, M. S. 2007. Flexibly integrating deliberation and execution in decision-theoretic agents. In *ICAPS Workshop on Planning and Plan-Execution for Real-World Systems*.

Wingate, D., and Seppi, K. D. 2005. Prioritization methods for accelerating MDP solvers. *JMLR* 6:851–881.

Zhou, R., and Hansen, E. A. 2006. Domain-independent structured duplicate detection. In *AAAI*.