External Memory Value Iteration

Stefan Edelkamp[*] and **Shahid Jabbar**[*]
Department of Computer Science
University of Dortmund, Germany
stefan.edelkamp@cs.uni-dortmund.de
shahid.jabbar@cs.uni-dortmund.de

Blai Bonet
Departamento de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Abstract

We propose a unified approach to disk-based search for deterministic, non-deterministic, and probabilistic (MDP) settings. We provide the design of an external Value Iteration algorithm that performs at most $O(l_G \cdot \mathit{scan}(|E|) + t_{\max} \cdot \mathit{sort}(|E|))$ I/Os, where l_G is the length of the largest backedge in the breadth-first search graph G having $|E|$ edges, $t_{\max}$ is the maximum number of iterations, and $\mathit{scan}(n)$ and $\mathit{sort}(n)$ are the I/O complexities for externally scanning and sorting n items. The new algorithm is evaluated over large instances of known benchmark problems. As shown, the proposed algorithm is able to solve very large problems that do not fit into the available RAM and thus out of reach for other exact algorithms.

Introduction

Guided exploration in deterministic state spaces is very effective in domain-dependent (Korf & Felner 2007) and domain-independent search (Bonet & Geffner 2001; Hoffmann 2003). There have been various attempts trying to integrate the success of heuristic search to more general search models. AO*, for example, extends A* over acyclic AND/OR graphs (Nilsson 1980), LAO* (Hansen & Zilberstein 2001) further extends AO* over AND/OR graphs with cycles and is well suited for Markov Decision Processes (MDPs), and Real-Time Dynamic Programming (RTDP) extends the LRTA* search algorithm (Korf 1990) over non-deterministic and probabilistic search spaces (Barto, Bradtke, & Singh 1995). LAO* and RTDP aim at the same class of problems, the difference however is that RTDP relies on trial-based exploration of the search space – a concept adopted from reinforcement learning – to discover the relevant states of the problem and determine the order in which to perform value updates. LAO*, on the other hand, finds a solution by systematically expanding a search graph in a manner akin to A* and AO*. The IDAO* algorithm, developed in the context of optimal temporal planning, performs depth-first iterative-deepening to AND/OR graphs (Haslum 2006). All these algorithms have the interleaving of dynamic updates of cost estimates and the extension of the search frontier in common.

Learning DFS was introduced in (Bonet & Geffner 2005; 2006) for a variety of models including deterministic models, Additive and Max AND/OR graphs, and MDPs. In the experiments, LDFS turned out to be superior to blind dynamic programming approaches like Value Iteration and heuristic search strategies like RTDP over MDPs. LDFS is designed upon a unified view of search spaces, described in terms of an initial state, terminal states and their costs, applicable actions, transition functions, and cost of applying actions on states, which is able to model deterministic problems, AND/OR graphs under additive and max cost criteria, Game trees and MDPs. Interestingly, LDFS instantiates to state-of-the-art algorithms for some of these models, and to novel algorithms on others. It instantiates to IDA* over deterministic problems and bounded-LDFS, LDFS with an explicit bound parameter, instantiates to the MTD$(-\infty)$ over Game trees (Plaat *et al.* 1996). On AND/OR models, it instantiates to novel algorithms (Bonet & Geffner 2005).

Often search spaces are so big that even in compressed form they do not fit into main memory. In these cases, all of the above algorithms are doomed to failure.

In this paper, we extend the Value Iteration algorithm, defined over the unified search model, to use external storage devices. The result is the External Value Iteration algorithm which is able to solve large instances of deterministic problems, AND/OR trees, Game trees and MDPs, and that uses the available RAM and secondary storage.

An orthogonal approach for large MDPs is to use approximation techniques to solve not the original problem, but an approximation of it with the hope that the solution to the latter will be a good approximation to the input problem. There are different approaches to do this. Perhaps the most known one is to use polynomial approximations or linear combinations of basis functions to represent the value function (Bellman, Kalaba, & Kotin 1963; Tsitsiklis & Roy 1996), or more complex approximations such as those based on neural networks (Tesauro 1995; Bertsekas & Tsitsiklis 1996). Recently, there have been methods based on LP and constraint sampling in order to solve the resulting approximations (Farias & Roy 2004; Guestrin, Koller, & Parr 2001). All these approaches do not compute exact solutions and thus are not directly compa-

[*]The authors are supported by the German Research Council (DFG) under projects ED 74/2 and ED 74/4.

rable with our approach. Furthermore, some of them transform the problem into a simpler one which is solved with the Value Iteration algorithm, and thus amenable to be treated with the techniques in this paper.

The paper is structured as follows. First, we review external memory algorithms, the unified search model and the Value Iteration algorithm. Afterwards, we address the external implementation of Value Iteration suited for the unified search model along with a working example. We provide some experimental results of the new algorithm over some known benchmarks, and finalize with conclusions and some future work.

External Memory Algorithms

The need to deal with massive data sets has given rise to the field of external memory algorithms and data structures. Such algorithms utilize hard disk space to compensate for the relatively small RAM. As disks are much slower, and unlike internal memory do not support constant time random accesses, the algorithms have to exploit locality of data in order to be effective. Such algorithms are analyzed on an external memory model as opposed to the traditional RAM model. One of the earliest efforts to formalize such a model is the two level memory model by Aggarwal & Vitter (1988). The model provides the necessary tools to analyze the asymptotic I/O complexity of an algorithm; i.e., the asymptotic number of input/output communication operations as the input size grows. The model assumes a small internal memory of size M with input of size $N \gg M$ residing on the hard disk. Data can be transferred between the RAM and the hard disk in blocks of size $B < M$; typically, $B = \sqrt{M}$. The complexity of external memory algorithms is conveniently expressed in terms of predefined I/O operations such as $\mathit{scan}(N)$ for scanning a file of size N with a complexity of $\Theta(N/B)$ I/Os, and $\mathit{sort}(N)$ for external sorting a file of size N with a complexity of $\Theta(N/B \log_{M/B} N/B)$ I/Os.

External graph-search algorithms (Sanders, Meyer, & Sibeyn 2002) exploit secondary storage devices to overcome limited amount of RAM unable to fit the *Open* and *Closed* lists. External breadth-first search has been analyzed by different groups. For undirected *explicit* graphs with $|V|$ nodes and $|E|$ edges, the search can be performed with at most $O(|V| + \mathit{scan}(|V|) + \mathit{sort}(|E|))$ I/Os (Munagala & Ranade 1999), where the first term is due to the explicit representation of the graph (stored on disk). This complexity has been improved by (Mehlhorn & Meyer 2002) through a more organized access to the adjacency list; extensive comparisons of the two algorithms on a number of graphs have been reported in (Ajwani, Dementiev, & Meyer 2006) and (Ajwani, Meyer, & Osipov 2007).

External *implicit* graph search was introduced by Korf (2003) in the context of BFS on $(n \times m)$-Puzzles. Edelkamp, Jabbar, & Schrödl (2004) proposed *External A** for performing heuristic search on large implicit undirected graphs and reduced the complexity to $O(\mathit{scan}(|V|) + \mathit{sort}(|E|))$. For directed graphs, the complexity is bounded by $O(l_G \cdot \mathit{scan}(|V|) + \mathit{sort}(|E|))$ I/Os (Zhou & Hansen 2006), where *locality* l_G is the length of the largest back-edge in the BFS graph and determines the number of previous layers to be looked at to remove duplicate nodes.

Internal-memory algorithms like A* remove duplicate nodes using a hash table. External memory algorithms, on the other hand, have to rely on alternative schemes for duplicate detection since a hash table cannot be implemented efficiently on external storage devices. Duplicate detection based on hash partitions was proposed in the context of a complete BFS for the Fifteen-Puzzle which utilized about 1.4 Terabyes of hard disk space (Korf & Schultze 2005), while Zhou & Hansen (2004) introduced the idea of *structured duplicate detection*. A large-scale search for optimally solving the 30 discs, 4 peg Tower-of-Hanoi problem was recently performed utilizing over 398 Gigabytes of hard disk space (Korf & Felner 2007). In model checking, where one deals with directed and weighted graphs, I/O efficient heuristic search has been proposed by Jabbar & Edelkamp (2005) and later extended to cycle detection (Edelkamp & Jabbar 2006b) and parallel external heuristic search (Jabbar & Edelkamp 2006) with a successful 3 Terabytes exploration. In action planning, Edelkamp & Jabbar (2006a) integrates external search in cost-optimal planning for PDDL3 domains.

The Unified Search Model

The general model for state-space problems considered by Bonet & Geffner (2006) is able to accommodate diverse problems in AI including deterministic, AND/OR graphs, Game trees and MDPs. The model consists of:

M1. a discrete and finite state space $\mathcal{S}$,

M2. a non-empty subset of terminal states $\mathcal{T} \subseteq \mathcal{S}$,

M3. an initial state $\mathcal{I} \in \mathcal{S}$,

M4. subsets of applicable actions $A(u) \subseteq A$ for $u \in \mathcal{S} \setminus \mathcal{T}$,

M5. a transition function $\Gamma(a, u)$ for $u \in \mathcal{S} \setminus \mathcal{T}$, $a \in A(u)$,

M6. terminal costs $c_{\mathcal{T}} : \mathcal{T} \to \mathbb{R}$, and

M7. non-terminal costs $c : A \times \mathcal{S} \setminus \mathcal{T} \to \mathbb{R}$.

For deterministic models, the transition function maps actions and non-terminal states into states, for AND/OR models it maps actions and non-terminal states into *subsets of states*, Game trees are AND/OR graphs with non-zero terminal costs and zero non-terminal costs, and MDPs are non-deterministic models with probabilities $P_a(v|u)$ such that $P_a(v|u) > 0$ if $v \in \Gamma(a, u)$, and $\sum_{v \in \mathcal{S}} P_a(v|u) = 1$.

The solutions to the models can be expressed in terms of Bellman equations. For the deterministic case, we have

$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min_{a \in A(u)} c(a, u) + h(\Gamma(a, u)) & \text{otherwise.} \end{cases}$$

For the non-deterministic, Additive and Max, cases

$$h_{add}(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a, u) + \sum\limits_{v \in \Gamma(a,u)} h(v) & \text{otherwise.} \end{cases}$$

$$h_{max}(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a, u) + \max\limits_{v \in \Gamma(a,u)} h(v) & \text{otherwise.} \end{cases}$$

Algorithm 1 Value Iteration

```
Input: State space model $M$; initial heuristic (estimates) $h$;
    tolerance $\epsilon > 0$; maximum iterations $t_{\max}$.
Output: $\epsilon$-Optimal value function if $t_{\max} = \infty$.
 1: $\mathcal{S} \leftarrow$ Generate-State-Space$(M)$
 2: for all $u \in \mathcal{S}$ do $h_0(u) \leftarrow h(u)$
 3: $t \leftarrow 0$; Residual $\leftarrow +\infty$
 4: while $t < t_{\max} \wedge$ Residual $> \epsilon$ do
 5:     Residual $\leftarrow 0$
 6:     for all $u \in \mathcal{S}$ do
 7:         Apply update rule for $h_{t+1}(u)$ based on the model
 8:         Residual $\leftarrow \max\{|h_{t+1}(u) - h_t(u)|,$ Residual$\}$
 9:     end for
10:     $t \leftarrow t + 1$
11: end while
12: return $h_{t-1}$
```

And, for the MDP case, we have

$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a,u) + \sum\limits_{v \in \mathcal{S}} P_a(v|u)h(v) & \text{otherwise.} \end{cases}$$

Solutions h^* to the Bellman equations are value functions of form $h : \mathcal{S} \to \mathbb{R}$. The value $h^*(u)$ expresses the minimum expected cost to reach a terminal state from state u. Policies, on the other hand, are functions $\pi : \mathcal{S} \to A$ that map states into actions and generalize the notion of plan for non-deterministic and probabilistic settings. In deterministic settings, a plan for state u consists of a sequence of actions to be applied at u that is guaranteed to reach a terminal state; it is optimal if its cost is minimum. Policies are greedy if they are best with respect to a given value functions; policies π^* greedy with respect to h^* are optimal.

Value Iteration

Value Iteration is an unguided algorithm for solving the Bellman equations and hence obtaining optimal solutions for models M1–M7.

The algorithms proceed in two phases. In the first phase, the whole state space is generated from the initial state $\mathcal{I}$. In this process, an entry in a hash table (or vector) is allocated in order to store the h-value for each state u; this value is initialized to $c_{\mathcal{T}}(u)$ if $u \in \mathcal{T}$, or to a given heuristic estimate (or zero if no estimate is available) if u is non-terminal.

In the second phase, iterative scans of the state space are performed updating the values of non-terminal states u as:

$$h_{\text{new}}(u) \;:=\; \min_{a \in A(u)} Q(a,u) \qquad (1)$$

where $Q(a,u)$, which depends on the model, is defined as

$$Q(a,u) \;=\; c(a,u) + h(\Gamma(a,u))$$

for deterministic models,

$$Q(a,u) \;=\; c(a,u) + \sum_{v \in \Gamma(a,u)} h(v)\,,$$

$$Q(a,u) \;=\; c(a,u) + \max_{v \in \Gamma(a,u)} h(v)$$

for non-deterministic Additive and Max models, and

$$Q(a,u) \;=\; c(a,u) + \sum_{v \in \mathcal{S}} P_a(v|u)h(v)$$

for MDPs.

Value Iteration converges to the solution h^* provided that $h^*(u) < \infty$ for all u. In the case of MDPs, which may have cyclic solutions, the number of iterations is not bounded and Value Iteration typically only converges in the limit (Bertsekas 1995). For this reason, for MDPs, Value Iteration is often terminated after a predefined bound of $t_{\max}$ iterations are performed, or when the residual falls below a given $\epsilon > 0$; the residual is $\max_{u \in \mathcal{S}} |h_{\text{new}}(u) - h(u)|$. Value Iteration is shown in Alg. 1.

External Value Iteration

We now discuss our approach for extending the Value Iteration procedure to work on large state spaces that cannot fit into the RAM. We call the new algorithm *External Value Iteration*. Instead of working on states, we chose to work on edges for reasons that shall become clear soon. In our case, an edge is a 4-tuple

$$(u, v, a, h(v))$$

where u is called the predecessor state, v the stored state, a the operator that transform u into v, and $h(v)$ is the current value for v. Clearly, v must belong to $\Gamma(a,u)$. In deterministic models, v is determined by u and a and so it can be completely dropped, but for the non-deterministic models, it is a necessity.

Similarly to the internal version of Value Iteration, the external version works in two phases. A forward phase, where the state space is generated, and a backward phase, where the heuristic values are repeatedly updated until an ϵ-optimal policy is computed, or $t_{\max}$ iterations are performed.

We will explain the algorithm using the graph in Fig. 1. The states are numbered from 1 to 10, the initial state is 1, and the terminal states are 8 and 10. The numbers next to the states are the initial heuristic values.

Forward Phase: State Space Generation

Typically, a state space is generated by a depth-first or a breadth-first exploration that uses a hash table to avoid re-expansion of states. We choose an external breadth-first exploration to handle large state spaces. Since in an external setting a hash table is not affordable, we rely on *delayed duplication detection* (DDD). It consists of two phases, first removing duplicates within the newly generated layer, and then removing duplicates with respect to previously generated layers. For undirected graphs, looking at two previous layers is enough to remove all duplicates (Munagala & Ranade 1999), but for directed graphs the *locality* l_G of the graph dictates the number of layers to be looked at. In the example graph of Figure 1, $l_G = 2$ (8 is generated in the third layer through state 3 and again through 7 and 9 in the fifth layer). Note that as we deal with edges, we might need to keep several copies of a state. In our case, an edge $(u, v, a, h(v))$ is a duplicate, if and only if, its predecessor

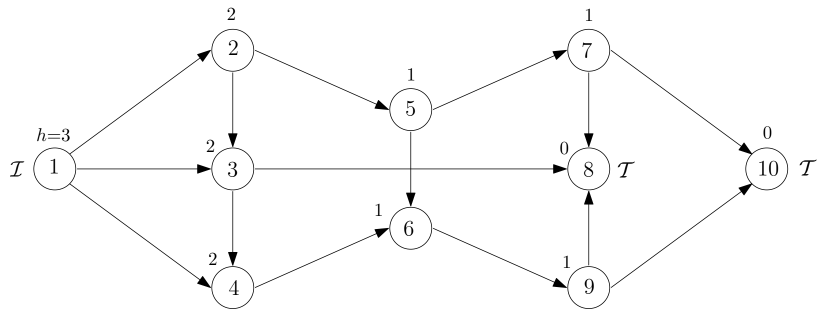
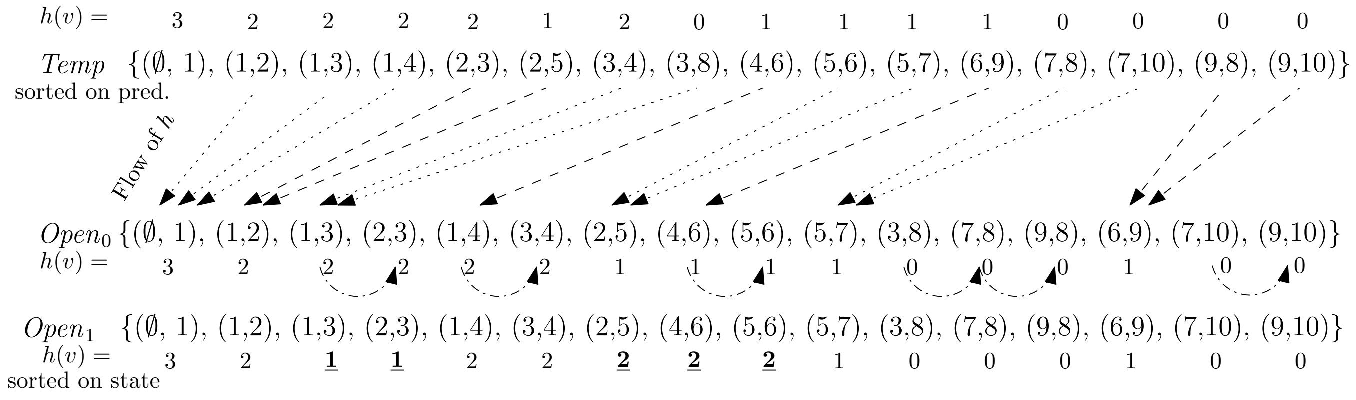


Figure 1: An example graph with initial h-values.



$h(v) =$ 3 2 2 2 2 1 2 0 1 1 1 1 0 0 0 0
$Temp$ {(∅, 1), (1,2), (1,3), (1,4), (2,3), (2,5), (3,4), (3,8), (4,6), (5,6), (5,7), (6,9), (7,8), (7,10), (9,8), (9,10)}
sorted on pred.
Flow of h
$Open_0$ {(∅, 1), (1,2), (1,3), (2,3), (1,4), (3,4), (2,5), (4,6), (5,6), (5,7), (3,8), (7,8), (9,8), (6,9), (7,10), (9,10)}
$h(v) =$ 3 2 2 2 2 2 1 1 1 1 0 0 0 1 0 0
$Open_1$ {(∅, 1), (1,2), (1,3), (2,3), (1,4), (3,4), (2,5), (4,6), (5,6), (5,7), (3,8), (7,8), (9,8), (6,9), (7,10), (9,10)}
$h(v) =$ 3 2 **1** **1** 2 2 **2** **2** **2** 1 0 0 0 1 0 0
sorted on state

Figure 2: Backward phase. the files *Open*$_0$ and *Temp* are stored on disk. A parallel scan of both files is done from left to right. The file *Open*$_1$ is the result of the first update. Values that changed in the first update are shown with bold underline typeface.

u, its state v, and the action a match an existing edge. Thus, in undirected graphs, there are two different edges for each undirected edge.

The procedure for DDD can be borrowed either from sorting-based DDD (Munagala & Ranade 1999), hash-based DDD (Korf & Schultze 2005), or structured duplicate detection (Zhou & Hansen 2004). In our case, sorting-based DDD is the best choice, since it is the most general form of DDD, which makes no further assumptions such as the maximum size of a layer, and also is best suited as the sorting order is further exploited during the backward phase.

Algorithm 2 shows External Value Iteration. The algorithm maintains layers $L(d)$ on disk in the form of files. The first phase ends up by concatenating all layers into one *Open* list that contains all edges reachable from $\mathcal{I}$. The complexity of this phase is $O(l_G \cdot \mathit{scan}(|E|) + \mathit{sort}(|E|))$ I/Os. The first term is obtained by summing up the total I/Os required for successors generation and subtracting l_G many previous layers. The second term *sort*$(|E|)$ is the accumulative I/O complexity for delayed duplicate detection on all the layers.

Backward Phase: Update of Values

This is the most critical part of the approach and deserves more attention. To perform the update (1) on the value of state v, we have to bring together the value of its successor states. As they both are contained in one file, and *there is no arrangement that can bring all successor states close to their predecessor states*, we make a copy of the entire graph (file) and deal with the current state and its successor differently. To establish the adjacencies, the second copy, called *Temp*, is sorted with respect to the node u. Remember that *Open* is sorted with respect to the node v.

A parallel scan of files *Open* and *Temp* gives us access to all the successors and values needed to perform the update on the value of v. This scenario is shown in Fig. 2 for the graph in the example. The contents of *Temp* and *Open*$_t$, for $t = 0$, are shown along with the heuristic values computed so far for each edge (u, v). The arrows show the flow of information (alternation between dotted and dashed arrows is just for clarity). The results of the updates are written to the file *Open*$_{t+1}$ containing the new values for each state after $t + 1$ iterations. Once *Open*$_{t+1}$ is computed, the file *Open*$_t$ can be removed as it is no longer needed.

Algorithm 3 shows the backward update algorithm for the case of MDP models; the other models are similar. It first copies the *Open*$_t$ list in *Temp* using buffered I/O operations, and sorts the new *Temp* list according to the predecessor states u. The algorithm then iterates on all edges from *Open*$_t$ and search for the successors in *Temp*. Since *Open*$_t$ is sorted with respect to states v, *the algorithm never goes back and forth in any of the Open*$_t$ *or Temp files*. Note that all reads and writes are buffered and thus can be carried out

Algorithm 2 External Value Iteration

```
Input: State space model $M$; initial value function $h$; tolerance $\epsilon > 0$; maximum iterations $t_{\max}$.
Output: $\epsilon$-Optimal value function (stored on disk) if $t_{\max} = \infty$.
 1: $L(0) \leftarrow \{(\emptyset, \mathcal{I}, —, h(\mathcal{I}))$
 2: $d \leftarrow 0$
 3: while $(L(d) \neq \emptyset)$ do
 4:     $d \leftarrow d + 1$
 5:     $L(d) \leftarrow \{(u, v, a, h(v)) : u \in L(d-1), a \in A(u), v \in \Gamma(a, u)\}$
 6:     Sort $L(d)$ with respect to edges $(u, v)$
 7:     Remove duplicate edges in $L(d)$
 8:     for $loc \in \{1, \ldots, l_G\}$ do $L(d) \leftarrow L(d) \setminus L(d - loc)$
 9: end while
10: $Open_0 \leftarrow L(0) \cup L(1) \cup \ldots \cup L(d-1)$
11: Sort $Open_0$ with respect to states $v$
12: $t \leftarrow 0$; $Residual \leftarrow +\infty$
13: while $t < t_{\max} \wedge Residual > \epsilon$ do
14:     $Residual \leftarrow$ External-VI-Backward-Update$(M, Open_t)$
15:     $t \leftarrow t + 1$
16: end while
```

very efficiently by always doing I/O operations in blocks.

We now discuss the different cases that might arise when an edge $(u, v, a, h(v))$ is read from *Open*$_t$. States from Fig. 1 that comply with each case are referred in parentheses, while the lines in the algorithm are referred in brackets. The flow of the values in h for the example is shown in Fig. 2.

- *Case I:* v is terminal (states 8 & 10). Since no update is necessary, the edge can be written to *Open*$_{t+1}$ [Line 5].
- *Case II:* v is the same as the last updated state (state 3). Write the edge to *Open*$_{t+1}$ with such last value [Line 7]. (Case shown in Fig. 2 with curved arrows.)
- *Case III:* v has no successors. That means that v is a terminal state and so is handled by case I [Line 11].
- *Case IV:* v has one or more successors (remaining states). For each action $a \in A(v)$, compute the value $Q(a, v)$ by summing the products of the probabilities and the stored values. Such value is kept in the array $Q(a)$ [Line 12].

For edges (x, y, a', h') read from *Temp*, we have:

- *Case A:* y is the initial state, implying $x = \emptyset$. Skip this edge since there is nothing to do. By taking $\emptyset$ as the smallest element, the sorting of *Temp* brings all such edges to the front of the file. (Case not shown for sake of brevity.)
- *Case B:* $x = v$, i.e. the predecessor of this edge matches the current state from *Open*$_t$. This calls for an update in the $Q(a)$ values [Lines 13–17].

The array $Q : A \rightarrow \mathbb{R}$ is initialized to the edge weight $c(a, v)$, for each $a \in A(v)$. Once all the successors are processed, the new value for v is the minimum of the values stored in the Q-array for all applicable actions.

An important point to note here is that the last edge read from *Temp* in Line 16 is not used. The operation **Push-back** in Line 18 puts back this edge into *Temp*. This operation

Algorithm 3 External Value Iteration – Backward Update

```
Input: State space model $M$; state space $Open_t$ stored on disk.
Output: $Open_{t+1}$ generated by this update, stored on disk.
 1: $Residual \leftarrow 0$
 2: Copy $Open_t$ into $Temp$
 3: Sort $Temp$ with respect to states $u$
 4: for all $(u, v, a, h) \in Open_t$ do
 5:     if $v \in \mathcal{T}$ then {CASE I}
 6:         Write $(u, v, a, h)$ to $Open_{t+1}$
 7:     else if $v = v_{last}$ then {CASE II}
 8:         Write $(u, v, a, h_{last})$ to $Open_{t+1}$
 9:     else
10:         Read $(x, y, a', h')$ from $Temp$
11:         Post-condition: $x = v$ {CASE III IS $x \neq v$}
12:         Post-condition: both files are aligned {CASE IV}
13:         for all $a \in A(v)$ do $Q(a) \leftarrow c(a, v)$
14:         while $x = v$ do
15:             $Q(a') \leftarrow Q(a') + P_{a'}(y|x)\, h'$
16:             Read $(x, y, a', h')$ from $Temp$
17:         end while
18:         Push-back $(x, y, a', h')$ in $Temp$
19:         $h_{last} \leftarrow \min_{a \in A(v)} Q(a)$
20:         $v_{last} \leftarrow v$
21:         Write $(u, v, a, h_{last})$ to $Open_{t+1}$
22:         $Residual \leftarrow \max\{|h_{last} - h|, Residual\}$
23: end for
24: return $Residual$
```

incurs in no physical I/O since the *Temp* file is buffered. Finally, to handle case II, a copy of the last updated node and its value are stored in variables v_{last} and h_{last} respectively.

Theorem 1 *The algorithm* External Value Iteration *performs at most* $O(l_G \cdot scan(|E|) + t_{\max} \cdot sort(|E|))$ *I/Os.*

Proof: The forward phase requires $l_G \cdot scan(|E|) + sort(|E|)$ I/Os. The backward phase performs at most $t_{\max}$ iterations. Each such iteration consists of one sorting and two scanning operations for a total of $O(t_{\max} \cdot sort(|E|))$ I/Os. ∎

Experiments

We implemented External Value Iteration (Ext-VI) and compared it with Value Iteration (VI), and in some cases to the LDFS and LRTDP algorithms, on several problem instances from 4 different domains.

The first domain is the racetrack benchmark used in a number of works. An instance in this domain is characterized by a racetrack divided into cells such that the task is to find the control for driving a car from a set of initial states into a set of goal states minimizing the number of time steps. Each applied control achieves its intended effect with probability 0.7 and no effect with probability 0.3. We report results on two instances from (Barto, Bradtke, & Singh 1995), the larger instance from (Hansen & Zilberstein 2001) and one new largest instance. We also extended the ring and square instances from (Bonet & Geffner 2006). The results, obtained on an Opteron 2.4 GHz Linux machine with

Algorithm	$\|\mathcal{S}\|/\|E\|$	RAM	Time	$\|\mathcal{S}\|/\|E\|$	RAM	Time	$\|\mathcal{S}\|/\|E\|$	RAM	Time	$\|\mathcal{S}\|/\|E\|$	RAM	Time
	barto-small			barto-big			hansen-bigger			largest		
VI	9,398	**7.0M**	1.2	22,543	16M	4.9	51,952	37M	22.1	150,910	77M	256.8
Ext-VI	139,857	7.6M	19.4	337,429	16M	82.5	780,533	**34M**	325.2	2,314,967	**67M**	1,365.3
	ring-4			ring-5			square-5			square-6		
VI	33,238	34M	6.5	94,395	86M	28.9	1,328,820	218M	2,899	out-of-memory		
Ext-VI	497,135	**33M**	92.2	1,435,048	**80M**	193.1	21,383,804	**160M**	3,975	62,072,828	**230M**	19,378

Table 1: Performance of External Value Iteration on the racetrack domain with parameters $p = 0.7$ and $\epsilon = 0.0001$.

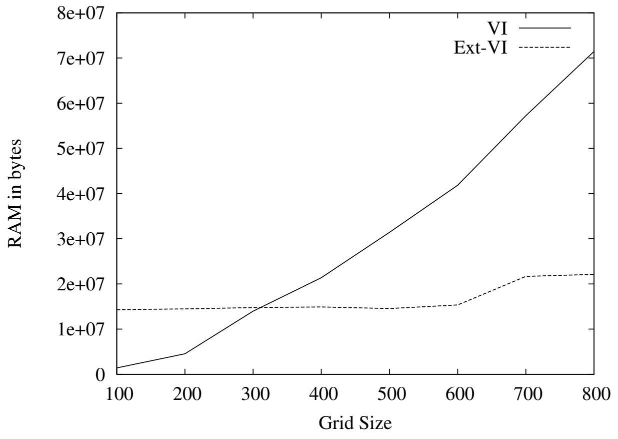


Figure 3: Memory consumption in the domain wet-floor

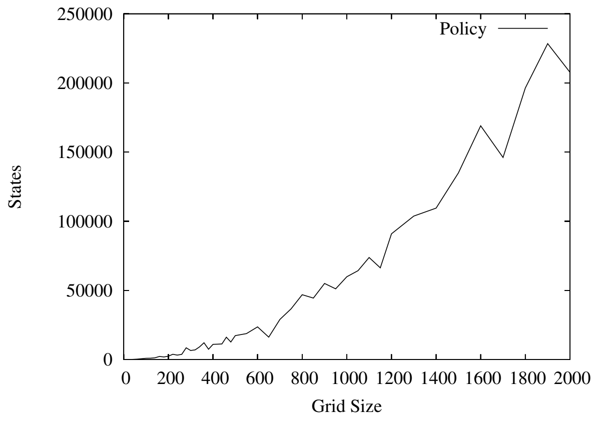


Figure 4: Growth of policy size in wet-floor domain.

a memory bound of 512MB, are depicted in Table 1. The table shows the size of the problem in states for VI and edges for Ext-VI, the amount of RAM used (in MB), and the total time spent by the algorithm in seconds.

For *square-6*, the state space could not be generated by VI within the memory bound. For Ext-VI, the edges took about 2GB of hard disk and another 6GB for the backward phase. We also started a much larger instance of the square model on an Athlon X2 with 2GB RAM. The instance consists of a grid of size 150×300 with the three start states at the top left corner and three goal states at bottom right corner. Internal memory VI consumed the whole RAM and could not finalize. Ext-VI generated the whole state space with 518,843,406 edges consuming 16GB on the disk while just 1.6GB on RAM. After 91 hours, the algorithm finished in 56 iterations with a value of $h^*(s_0) = 29.233$ and residual $< 10^{-4}$. We also evaluated the algorithms LDFS and LRTDP on the same instance. LRTDP consumed 2GB while running for 12 hours and was canceled. LDFS consumed 1.5GB while running for 118 hours and also was canceled.

The difference in RAM for Ext-VI observed in the table is due to the internal memory consumption for loading the instances. Moreover, external sorting of large files require opening of multiple file pointers each allocating a small internal memory buffer.

In the second experiment, we used the wet-floor domain from (Bonet & Geffner 2006) which consists of a navigation grid in which cells are wet with probability $p = 0.4$ and thus slippery. The memory consumption in bytes is shown in Fig. 3 for grids ranging from 100×100 to 800×800. As shown, the memory consumption of VI grows without control, whereas for Ext-VI the memory consumption can be adjusted to fit the available RAM. Indeed, we also tried a large problem of 10,000×10,000 on an Athlon X2 machine with 2GB RAM. The internal version of VI quickly went out of memory since the state space contains 100M nodes. Ext-VI, on the other hand, was able to generate the whole state space with diameter 19,586 and 879,930,415 edges taking 16GB of space to store the BFS layers and about 45GB of space for the backward updates. Each backward-update iteration takes about half an hour, and thus we have calculated that Ext-VI will take about 2 years to finish; we stopped the algorithm after 14 iterations. Fig. 4 shows the increase in size of the optimal policy for the wet-floor domain in instances from 100×100 to 2000×2000. As it can be seen, the policy size grows quadratically and since any internal-memory algorithm must store at least all such states, one can predict that no such algorithm will be able to solve the 10,000×10,000 instance. Fig. 5 shows the number of iterations needed by internal Value Iteration on the wet-floor domain over the same instances.

The third domain considered is the $n \times m$ sliding tile puzzle. We performed two experiments: one with deterministic moves, and the other with noisy operators that achieve their intended effects with probability $p = 0.9$ and no effect with probability $1 - p$. Table 2 shows the results for random instances of the 8-puzzle for both experiments. Note the differences in the total iterations performed by the two

Algorithm	p	$\lvert\mathcal{S}\rvert/\lvert E\rvert$	Iter.	Updates	$h(\mathcal{I})$	$h^*(\mathcal{I})$	RAM	Time
VI($h = 0$)	1.0	181,440	27	4,898,880	0	14.00	21M	6.3
Ext-VI($h = 0$)	1.0	483,839	32	5,806,048	0	14.00	**11M**	71.5
VI(h_{manh})	1.0	181,440	20	3,628,800	10	14.00	21M	4.4
Ext-VI(h_{manh})	1.0	483,839	28	5,080,292	10	14.00	**11M**	65.2
VI($h = 0$)	0.9	181,440	37	6,713,280	0	15.55	21M	8.7
Ext-VI($h = 0$)	0.9	967,677	45	8,164,755	0	15.55	**12M**	247.4
VI(h_{manh})	0.9	181,440	35	6,350,400	10	15.55	21M	8.3
Ext-VI(h_{manh})	0.9	967,677	43	7,801,877	10	15.55	**12M**	237.4

Table 2: Performance of External Value Iteration on deterministic and probabilistic variants of 8-puzzle.

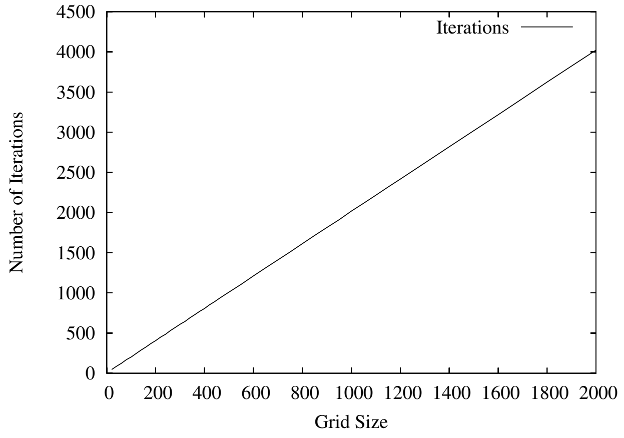


Figure 5: Iterations in wet-floor domain by Internal VI.

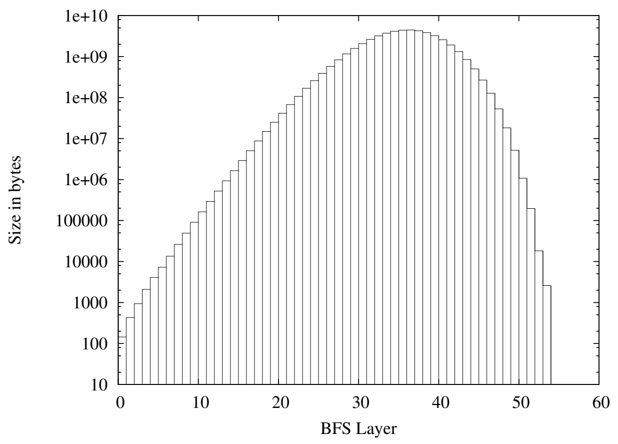


Figure 6: Edge space of 11-puzzle with $p = 0.9$

algorithms. This is due to the fact that during an iteration, VI can also use the new values of the successors updated in the same iteration. On the other hand, Ext-VI does not have constant time accesses to these new values, as they might have already been flushed to the disk. We also tried our algorithm on the 3×4-puzzle with $p = 0.9$. The problem cannot be solved with our internal VI because the state space does not fit in RAM, there are $12!/2 \approx 239 \times 10^6$ states. Ext-VI generated a total of 1,357,171,197 edges taking 45GB of disk space. Figure 6 shows the memory consumption for the edges in each layer of the BFS exploration. The backward update took 437 hours and *finished in 72 iterations* until the residual became less than $\epsilon = 10^{-4}$. The internal memory consumed is 1.4G on a Pentium-4 3.2GHz machine. The h-value of initial state converged to 28.8889.

Finally, we implemented Ext-VI for deterministic settings within the planner MIPS-XXL (Edelkamp, Jabbar, & Nazih 2006), which is based on the state-of-the-art planner *MetricFF* (Hoffmann 2003). As an example, we choose the propositional domain TPP from the recent planning competition. For instance 6 of TPP, 3,706,936 edges were generated. In 30 iterations the change in the average h-value was less than 0.001. It took 5 hours on a 3.2GHz machine taking a total of 3.2GB on the disk, while 167MB on RAM.

Conclusions and Future Work

Wingate & Seppi (2004) proposed to use a disk-based cache mechanism into Value Iteration and Prioritized Value Iteration. The paper provides empirical evaluation on number of cache hits and misses for both the algorithms and compare it against different cache sizes. However, External Value Iteration provides the first implementation of a well-established AI algorithm, able to solve different state space models, that exploits secondary storage in an *I/O efficient* manner. Contrary to internal Value Iteration, the external algorithm works on edges rather than on states. We provided an I/O complexity analysis that is bounded by the number of iterations times the sorting complexity. This is itself a non-trivial result, as backward induction has to connect the predecessor states' values with the current state's value by connecting two differently sorted files in order to apply the Bellman update. We provided empirical results on known benchmark problems showing that the disk space can overcome limitations in main memory. The largest state space we have been able to generate in the probabilistic setting took 45.5GB.

The obtained I/O complexity bound is remarkable: we cannot expect a constant number of iterations, since, in difference to the deterministic case, there is currently no (internal) algorithm known to solve non-deterministic and MDP problems in a linear number of node visits.

Having a working externalization, it is worthwhile to try parallelizing the approach, e.g., on a cluster of workstations or even on a multi-core machine. Since the update can be done in parallel (different processors working on different parts of the edge list), a full parallel implementation would require the development of external parallel sorting procedures. It would also be interesting to extend the approach to heuristic search planning in non-deterministic and probabilistic models.

References

Aggarwal, A., and Vitter, J. S. 1988. The I/O complexity of sorting and related problems. *JACM* 31(9):1116–1127.

Ajwani, D.; Dementiev, R.; and Meyer, U. 2006. A computational study of external memory BFS algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 601–610.

Ajwani, D.; Meyer, U.; and Osipov, V. 2007. Improved external memory BFS implementations. In *Workshop on Algorithm engineering and experiments (ALENEX)*, 3–12.

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72(1):81–138.

Bellman, R.; Kalaba, R.; and Kotin, B. 1963. Polynomial approximation – a new computational technique in dynamic programming. *Math. Comp.* 17(8):155–161.

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, (2 Vols)*. Athena Scientific.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bonet, B., and Geffner, H. 2005. An algorithm better than AO*. In *AAAI*, 1343–1348.

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

Edelkamp, S., and Jabbar, S. 2006a. Cost-optimal external planning. In *AAAI*, 821–826.

Edelkamp, S., and Jabbar, S. 2006b. Large-scale directed model checking LTL. In *Model Checking Software, SPIN*, volume 3925 of *LNCS*, 1–18.

Edelkamp, S.; Jabbar, S.; and Nazih, M. 2006. Cost-optimal planning with constraints and preferences in large state spaces. In *ICAPS Workshop on Preferences and Soft Constraints in Planning*.

Edelkamp, S.; Jabbar, S.; and Schrödl, S. 2004. External A*. In *German Conf. on AI*, volume 3238 of *LNAI*, 226–240.

Farias, D. P. D., and Roy, B. V. 2004. On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of Operations Research* 29(3):462–478.

Guestrin, C.; Koller, D.; and Parr, R. 2001. Max-norm projections for factored MDPs. In Nebel, B., ed., *Proc. 17th International Joint Conf. on Artificial Intelligence*, 673–680. Seattle, WA: Morgan Kaufmann.

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

Haslum, P. 2006. Improving heuristics through relaxed search. In *Journal of Artificial Research*, volume 25, 233–267.

Hoffmann, J. 2003. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Jabbar, S., and Edelkamp, S. 2005. I/O efficient directed model checking. In *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, 313–329.

Jabbar, S., and Edelkamp, S. 2006. Parallel external directed model checking with linear I/O. In *Verification, Model checking, and Abstract Interpretation*, volume 3855 of *LNCS*, 237–251.

Korf, R. E., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *IJCAI*, 2324–2329.

Korf, R. E., and Schultze, T. 2005. Large-scale parallel breadth-first search. In *AAAI*, 1380–1385.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Korf, R. E. 2003. Breadth-first frontier search with delayed duplicate detection. In *MOCHART workshop*, 87–92.

Mehlhorn, K., and Meyer, U. 2002. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, 723–735.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, 687–694.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Company.

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence* 87(1-2):255–293.

Sanders, P.; Meyer, U.; and Sibeyn, J. F. 2002. *Algorithms for Memory Hierarchies*. Springer.

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38:58–68.

Tsitsiklis, J., and Roy, B. V. 1996. Feature-based methods for large scale dynamic programming. *Machine Learning* 22:59–94.

Wingate, D., and Seppi, K. D. 2004. Cache performance of priority metrics for MDP solvers. In *AAAI Workshop on Learning and Planning in Markov Processes*, 103–106.

Zhou, R., and Hansen, E. 2004. Structured duplicate detection in external-memory graph search. In *AAAI*, 683–689.

Zhou, R., and Hansen, E. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170:385–408.