Domain-Independent, Automatic Partitioning for Probabilistic Planning

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

Recent progress on external-memory MDP solvers, in particular PEMVI [Dai *et al.*, 2008], has enabled optimal solutions to large probabilistic planning problems. However, PEMVI requires a human to *manually* partition the MDP before the planning algorithm can be applied — putting an added burden on the domain designer and detracting from the vision of automated planning. This paper presents a novel partitioning scheme, which automatically subdivides the state space into blocks that respect the memory constraints. Our algorithm first applies static domain analysis to identify candidates for partitioning, and then uses heuristic search to generate a 'good' partition. We evaluate the usefulness of our method in the context of PEMVI across many benchmark domains, showing that it can successfully solve extremely large problems in each domain. We also compare the performance of automatic partitioning with previously reported results using human-designed partitions. Experiments show that our algorithm generates significantly superior partitions, which speed MDP solving and also yield vast memory savings.

1 Introduction

AI researchers typically formulate planning under uncertainty problems using a Markov Decision Process (MDP) [Bresina *et al.*, 2002; Aberdeen *et al.*, 2004]. Popular algorithms to solve MDPs, like value iteration (VI), RTDP, LAO*, *etc.*, are all based on dynamic programming [Bellman, 1957; Barto *et al.*, 1995; Hansen and Zilberstein, 2001]. While effective in small settings, all these algorithms have memory requirements polynomial in the size of the (reachable) state space, which is exponential in the number of domain features. This greatly constrains the size of the problems that can be solved, making existing techniques practically useless for real planning problems.

Recent progress for scaling up dynamic programming has exploited external memory to alleviate this memory bottleneck. EMVI [Edelkamp *et al.*, 2007] associates values with edges instead of nodes. Using a clever sorting routine it ensures that all the backups of a VI can be accomplished by a scan on two contiguous files from the disk. EMVI is a theoretically clever algorithm, but was outperformed in practice by PEMVI [Dai *et al.*, 2008]. PEMVI divides the states into blocks and performs (one or more) backups on all states in a block before moving onto the next ones. All information required for backing blocks is stored on the disk and is transferred on demand in a piecemeal fashion. This simple idea was very effective in practice and significantly outperformed EMVI, but, with one caveat – PEMVI required a human to specify the partitioning of the state space. Human specified partitioning is a far cry from the vision of automated planning.

In this paper, we develop and implement automatic partitioning on top of PEMVI. We first build a theory of XOR groups [Edelkamp and Helmert, 1999] and construct necessary and sufficient conditions for a set of fluents to be a valid XOR group. We use static domain analysis to identify *XOR groups* present in the domain and search over a partition space created by these XOR groups yielding a suitable partition. We evaluate three criteria (*viz.* coherence, locality and balance) for this search, and conclude that a combination of two yields the best results. We also observe that our automatic partitioning engine constructs better partitions than the manual partitioning constructed in our previous work [Dai *et al.*, 2008], due to which we are able to solve problems larger than ones that were previously reported. We release a fully automated, domain-independent, optimal MDP solver that is able to solve many large problems across domains from the international planning competition 2006 [ipc, 2006].

2 Background

A Markov decision process is defined as a four-tuple $\langle \mathcal{S}, \mathcal{A}, T, C \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ represents the set of all applicable actions, T is the transition matrix describing the domain dynamics, and C denotes the cost of actions. A probabilistic planning problem is typically described by a *factored* MDP, which specifies F — the set of domain features. The state space $\mathcal{S}$ can be calculated as the set of value assignments to the features in F.

The agent executes its actions in discrete time steps called *stages*. At each stage, the system is in one distinct state $\mathbf{s}$. The agent can pick from any action $\mathbf{a} \in \mathcal{A}$, incurring a cost of $C(\mathbf{s}, \mathbf{a})$. The action takes the system to a new state $\mathbf{s}'$ stochastically, with probability $T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})$.

The *horizon* of an MDP is the number of stages for which costs are accumulated. We are interested in a special set of MDPs called *stochastic shortest path* problems. The horizon is indefinite and the costs are accumulated with no discount. There are a set of sink *goal states* $\mathcal{G} \subseteq \mathcal{S}$, reaching which terminates the execution. To solve the MDP we need to find an *optimal policy* ($\mathcal{S} \rightarrow \mathcal{A}$), a probabilistic execution plan that reaches a goal state with the minimum expected cost. We evaluate a policy π by a *value function*. Any optimal policy must satisfy the following system of *Bellman equations*:

$$\begin{aligned} V^*(\mathbf{s}) &= 0 \quad \text{if } \mathbf{s} \in \mathcal{G} \text{ else} \\ V^*(\mathbf{s}) &= \min_{\mathbf{a} \in \mathcal{A}} [C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T(\mathbf{s}'|\mathbf{s}, \mathbf{a}) V^*(\mathbf{s}')] \end{aligned} \tag{1}$$

The optimal policy, thus, is extracted from the value function:

$$\pi^*(\mathbf{s}) = argmin_{\mathbf{a} \in \mathcal{A}} C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V^*(\mathbf{s}').$$

2.1 Dynamic Programming

Most popular MDP algorithms are based on dynamic programming. Its usefulness was first proved by a simple yet powerful algorithm named *value iteration* [Bellman, 1957]. Value iteration first initializes the value function arbitrarily. Then, the values are updated using an operator called *Bellman backup* to create successively better approximations. Value iteration stops updating when the value function converges.

Value iteration (and other modern algorithms like LAO*, labeled RTDP, *etc.*) converges to the optimal value function in time polynomial in the size of the states [Littman *et al.*, 1995], but the model of an MDP (transition and value function) must be present in the memory before computation is applicable. This constraint prohibits these algorithms from solving problems whose models do not fit in the memory. This precludes most real world problems, since they are too large to be optimally solvable.

2.2 External-Memory Dynamic Programming

Recently, researchers have developed external memory algorithms that use disk to store the large MDP models. External memory value iteration (EMVI) [Edelkamp *et al.*, 2007] is the first such MDP algorithm. It associates values with edges instead of nodes. Using a clever sorting routine it ensures that all the backups of a VI can be accomplished by a scan on two contiguous files from the disk. EMVI is able to solve larger problems than classical dynamic programming, albeit, somewhat slowly.

Partitioned external memory value iteration (PEMVI) [Dai *et al.*, 2008] is a much faster external memory algorithm. It first partitions the state space into non-overlapping *partition blocks*, and then iteratively backs up states per block till convergence. PEMVI is able to make efficient use of I/O by backing up a state more than once per I/O iteration, so it converges faster than EMVI by a magnitude. However, PEMVI assumes a given *valid partition*, *i.e.*, one in which each partition block can be backed up in memory. In this work, the partitions are designed with human input by applying the manually chosen XOR constraints in a domain dependent way. Specifying a partition of the state space is added burden on the domain designer and takes away from the vision of AI planning. In this paper, we focus on generating the partitions automatically making the overall planner completely automated and capable of solving large problems using additional storage in the external memory.

2.3 State Abstraction in Deterministic Domains

Edelkamp & Helmert [1999] use automatically-constructed domain invariants to produce a compact *encoding*[1] of the problem. These invariants were used by Zhou & Hansen [2006] as *XOR groups* to define state abstractions for structured duplicate detection in external-memory search algorithms. Zhou & Hansen also defined the *locality* heuristic as a criterion for choosing a suitable state abstraction. This paper generalizes the notion of XOR groups, adapts them to probabilistic domains, and investigates additional partitioning heuristics, which are more relevant to probabilistic domains.

3 A Theory of XOR Groups

While there is an exponential number of ways to partition the state space, we focus on structurally meaningful partitions, which can be represented compactly. We use the key notion of *XOR groups* developed by Zhou and Hansen, but, generalize it as follows: *A set of formulae forms an XOR group if exactly one formula in the set is true in any state reachable from the start state.* For example, for any formula φ, the set $\{\varphi, \neg\varphi\}$ represents an XOR group. For another example consider the Explosive-Blocksworld domain with two block constants, b_1 and b_2. The fact that either a block (say b_1) is *clear* or else it must have another block *on* it can be represented as an XOR group: $\{clear(b_1), on(b_1, b_1), on(b_2, b_1)\}$. Sometimes it is clearer to think of an XOR group as a logical formula. We can write this XOR group logically as $clear(b_1) \oplus on(b_1, b_1) \oplus on(b_2, b_1)$. It is easy to see that an XOR group compactly represents a partition of the state space, where each *partition block* contains all the states satisfying a single formula (exclusive disjunct) in the group.

Our definition of an XOR-group is more general than the original one by Zhou and Hansen — they only allowed positive literals in an XOR group. On the other hand we accept any complex formula in our definition of XOR-groups.

In order to achieve a compact representation, it is convenient to specify XOR groups using first-order logic. This is facilitated by introducing some syntactic sugar, an *exclusive quantifier* $\exists_1$, to the logic. Intuitively, $\exists_1 x\ \varphi(x)$ means that there exists exactly one constant that satisfies $\varphi(x)$, that is, *exclusively quantified* is represented as $\exists_1 x$. With this notation, a large XOR group can be represented by a short logical formula. For example, if there were many blocks in the domain, then the XOR group shown previously would get quite long, but this notation allows a compact specification:

$$\exists_1 b\ [clear(b_1) \oplus on(b, b_1)].$$

Note that this group is specific to the specific block, b_1, but there is a similar group for b_2: $\{clear(b_2), on(b_1, b_2), on(b_2, b_2)\}$. These two XOR

[1]Encoding here refers to converting the PDDL description of the domain to an explicit state representation.

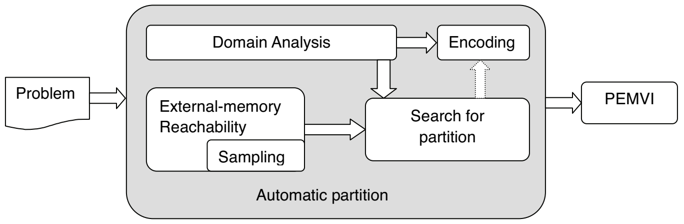

Figure 1: Flow chart of our planning system

groups can be represented, by adding a new, universally quantified, variable b':

$$\forall b' \exists_1 b \, [clear(b') \oplus on(b, b')].$$

If we ground this formula (assuming a universe of two blocks), we get a formula, whose conjuncts corresponds to an XOR group each.

$$(clear(b_1) \oplus on(b_1, b_1) \oplus on(b_2, b_1)) \quad \wedge$$
$$(clear(b_2) \oplus on(b_1, b_2) \oplus on(b_2, b_2))$$

We define a set of formulae $\mathcal{F}$ *balanced under* an action **a** if in each of its probabilistic outcomes, **a**'s effects either (i) do not change the truth value of any formula in $\mathcal{F}$ or (ii) make exactly one of the formulae in $\mathcal{F}$ true, while making another (previously true) formula in $\mathcal{F}$ false.

Necessary and Sufficient Conditions: A set of grounded formulae $\mathcal{F}$ forms a XOR group if and only if $\mathcal{F}$ satisfies the following two conditions:

1. Exactly one $\varphi \in \mathcal{F}$ is satisfied in the initial state.
2. For every ground action **a** that is executable in the reachable part of the state space, $\mathcal{F}$ is balanced under **a**.

These constraints help us efficiently evaluate whether a candidate set of formulae is indeed an XOR group or not.

4 Algorithm

Our planning system contains several pieces, shown in Figure 1. First, it performs static analysis of the domain and identifies the various XOR groups present in it. The external-memory reachability analysis detects irrelevant parts of the state space. Additionally, it samples a subset of the reachable space, which facilitates the next phase (searching for a partition) by estimating the sizes of various partitions. Then it searches in the space of partitions, represented using the XOR groups, to construct a valid partition. Next, the encoding module scans the reachable state space and computes a much more compact string representation for each state based on the XOR groups. It also prepares the data structures necessary for PEMVI, such as distributing all states into disk files, one for each partition block, *etc.* Finally we apply PEMVI over this encoding of states and the chosen partition.

4.1 Domain Analysis

Our original definition of XOR groups is very general, but, searching over such a general space of formulae may not be practical, so we add two constraints in our search for XOR groups: 1) All formulae in an XOR group must be literals (instead of a general Boolean formula). 2) all terms in a first-order XOR formula must either be universally quantified or exclusively quantified (*i.e.*, they can't be constants). In theory, these assumptions may result in missing some XOR groups, but, our experiments demonstrate that our method yields useful XOR groups for all planning benchmarks. Similar results were reported in [Zhou and Hansen, 2006].

We identify XOR groups by performing static domain analysis — first finding *first-order* XOR formulae and later grounding them to construct the final set. In the process we use a new term: We say a set of literals, $\mathcal{P}$, is *fully balanced* (balanced for short) if it is balanced under *all* ground actions. This is faster to check than the necessary XOR conditions which quantify over *executable* actions. A balanced set of literals is *minimal* if no proper subset is also balanced. We denote the set of arguments of a predicate p by *args*(p).

Finding First-Order XOR Groups: At the highest level, our algorithm searches through syntactic XOR formulae in first-order logic and looks for candidates that satisfy the XOR conditions of the previous section. Suppose we want to construct an XOR formula such that each literal corresponds to a predicate from the list $\mathcal{P}$. Note that each predicate $\in \mathcal{P}$ may be in its positive or negated form and two predicates $\in \mathcal{P}$ may be the same[2]. Such a $\mathcal{P}$ is the input to the Algorithm 1.

Algorithm 1 first computes all possible variable bindings (Lines 3-8). For example, suppose $\mathcal{P} = \langle clear(\cdot), on(\cdot,\cdot)\rangle$. Then the three arguments in $\mathcal{P}$ could be bound to the same variable name: $clear(x) \oplus on(x, x)$; all different variable names: $clear(x) \oplus on(y, z)$ or the two intermediate cases. All such cases will lead to different final XOR formulae. To compute this we consider all possible *type-sensitive equivalence relations* for the set of arguments and assign the same variable name if two arguments lie in the same equivalence class

Algorithm 1 Xorify($\mathcal{P}$)

```
1: Input: $\mathcal{P} = \langle p_1, p_2, \ldots, p_k \rangle$ (a list of predicates)
2: // compute candidate XORs
3: $XF \leftarrow \emptyset$
4: $A \leftarrow \{a_{ij}\}|_{i=1..k}$ where $a_{ij}$ is $j^{th}$ argument of $p_i$
5: $\mathcal{C} \leftarrow$ set of all possible type-sensitive equivalence relations of $A$ (Algo 2)
6: for all equivalence relations $c \in \mathcal{C}$ do
7:     $\varphi \leftarrow$ XOR of all predicates in $\mathcal{P}$ such that two arguments are represented by the same variable name iff they lie in the same equivalence class $\in c$
8:     add $\varphi$ to $XF$
9:
10: for all $\varphi \in XF$ do
11:     $AA \leftarrow$ set of all arguments in $\varphi$
12:     $open \leftarrow \{emptyset\}$
13:     while $open$ is not empty do
14:         delete the first set, $\mathcal{E}$, from $open$
15:         if $(\forall_{x \in \mathcal{E}} x)(\exists_{1_{y \in AA-\mathcal{E}}} y)[\varphi]$ is an XOR formula then
16:             output $(\forall_{x \in \mathcal{E}} x)(\exists_{1_{y \in AA-\mathcal{E}}} y)[\varphi]$
17:             goto line 10
18:         for all $a \in AA - \mathcal{E}$ do
19:             $\mathcal{E}' \leftarrow \mathcal{E} \cup \{a\}$
20:             if $\mathcal{E}' \notin open$ then
21:                 append $\mathcal{E}'$ at the end of $open$
```

[2]This is necessary in searching for XOR groups of the form $\forall b[clear(b) \oplus \neg clear(b)]$

Algorithm 2 Compute Type-Sensitive Equivalence Relation

```
1: Input: $A$ (a set of arguments) $k$ (the number of types)
2: distribute arguments by type into sets $A_1, \ldots, A_k$.
3: for each type $t_i$ do
4:     $\mathbb{R}_i \leftarrow EquivReln(A_i)$
5: return the Cartesian product of $\mathbb{R}_1, \ldots, \mathbb{R}_k$
6:
7: Function $EquivReln(B)$
8: if $B = emptyset$ then
9:     return a singleton that contains $emptyset$
10: else
11:     $x \leftarrow$ last element in $B$
12:     $\mathbb{R} \leftarrow EquivReln(B - \{x\})$
13:     $\mathbb{R}_{new} \leftarrow emptyset$
14:     for every relation $\mathcal{R} \in \mathbb{R}$ do
15:         for every equivalence class $D \in \mathcal{R}$ do
16:             $D_{new} \leftarrow D \cup \{x\}$
17:             add $((\mathcal{R} \cup \{D_{new}\}) - D)$ to $\mathbb{R}_{new}$
18:         add $\mathcal{R} \cup \{\{x\}\}$ to $\mathbb{R}_{new}$
19:     return $\mathbb{R}_{new}$
```

(line 7). We define a type-sensitive equivalence relations of a set of arguments A to be a set of non-empty, non-overlapping subsets of arguments whose union is A, with the constraint that elements in one equivalence class must have the same type. For example, if $A = \{b_1, b_2, c_1, c_2\}$ (two blocks and two colors) then type sensitive equivalence relation of A is $\{\{b_1, b_2\}, \{c_1, c_2\}\}, \{\{b_1\}, \{b_2\}, \{c_1, c_2\}\}, \{\{b_1, b_2\}, \{c_1\}, \{c_2\}\}, \{\{b_1\}, \{b_2\}, \{c_1\}, \{c_2\}\}$. Algorithm 2 outlines a method to compute this set.

Given this candidate XOR formula, φ, whose variables are not quantified so far, we wish to find a set, $\mathcal{E}$, from all arguments of φ, such that 1) when all arguments in $\mathcal{E}$ are universally quantified and all arguments in *args*$(\varphi) - \mathcal{E}$ are exclusively quantified, a correct XOR group results; and 2) no proper subset of $\mathcal{E}$ meets condition 1. For all $\mathcal{E}$ that meet condition 1 we say $\mathcal{E}$ *xorifies* ϕ. We call a set that meets both condition 1 and 2 a *minimal xorifier*.

To find this set of arguments we perform a breadth-first search over all subsets of *args*(φ), starting from the empty set (lines 12-21 in Algo 1). Expanding a node adds another argument to the current node. The goal is a minimal set $\mathcal{E}$ that xorifies φ. Running this algorithm on different $\mathcal{P}$ yields several first order XOR formulae. Finally, we ground these to get the concrete XOR groups.

As an illustration, consider a modified Explosive-Blocksworld, where blocks are colored. Consider a world with two pigment constants, red and $blue$, and an initial state, which specifies that initially b_1 and b_2 are red and $blue$ respectively. The only action to modify a block's color is paint:

```
(:action paint
  :parameters (?b - block ?c ?nc - pigment)
  :precondition (color ?b ?c)
  :effect (and (color ?b ?nc)
               (not (color ?b ?c))))
```

To find the minimal set using *just* the predicate *color*(b, c) we start the search with $\mathcal{E} = \emptyset$ (line 12). $\emptyset$ does not xorify *color*, since the XOR formula $\exists_1 b \exists_1 c\ [color(b, c)]$ does not hold, because, both blocks have some color in the initial state. The algorithm then looks for sets of size 1. Consider $\mathcal{E} = \{c\}$, or equivalently an XOR group $\forall c \exists_1 b\ [color(b, c)]$. This means there exists exactly one block of each color. Although the initial state satisfies this condition, the $paint$ action may violate it. So it does not xorify $color$. But $\mathcal{E} = \{b\}$ xorifies $color$, since it means every block must have exactly one color. This minimal xorifier leads to two ground xor groups: $color(b_1, red) \oplus color(b_1, blue)$ and $color(b_2, red) \oplus color(b_2, blue)$.

Our algorithm for computing XOR groups has marked differences from the previous version by Edelkamp & Helmert. We call their version *EH algorithm*. In particular, EH algorithm (1) did not handle any XOR formulae with negated predicates, and (2) required exactly one argument in the whole formula to be exclusively quantified, hence they cannot infer properties like $\forall x[loves(x, x) \oplus depressed(x)]$, or $\forall pkg \exists_1 truck \exists_1 city[on(pkg, truck) \oplus in(pkg, city)]$. We implemented an extension of algorithm that additionally finds *simple negated XOR groups* like $\forall b[clear(b) \oplus \neg clear(b)]$.

4.2 External-Memory Reachability Analysis

Often the state space reachable from the start state and the state space expressed by the PPDDL description are of extremely varied sizes. For example, the Blocksworld problems have orders of magnitude more unreachable states than reachable ones, because the PPDDL description allows for several blocks on top of a block, whereas the semantics of the domain given the typical initial states does not. We perform reachability analysis to focus computation only on the relevant subset of the states. Since the state spaces can be huge we implement an external memory version, which is a layered BFS that uses delayed duplicate detection [Korf, 2003]. Our optimizations enable the search algorithm to switch automatically between the in-memory (when a search frontier can fit in the memory) algorithm, and the external-memory version.

Sampling: Additionally, we build a random subset of the reachable state space by sampling each state with a uniform probability x. This subset helps us in estimating the sizes of partition blocks and its transitions in the next phase.

4.3 Search for Partitions

Recall that a *valid partition* is one in which partition blocks can be backed up in memory. In this phase we search for a valid partition by successively applying the XOR groups computed by domain analysis.

Choosing a good partition is vital for PEMVI algorithm, since it can massively save on the computation time, as well as memory requirements. In our previous work, we proposed two desirable heuristics that can be used to assess the quality of a partition — *locality* and *coherence* [Dai *et al.*, 2008]. Coherence favors a partition whose total percentage of intra-block state transitions is the greatest among all transitions, and therefore optimizes the information flow to achieve faster convergence. Locality prefers a partition that has the lower number of successor blocks, so it makes loading successor blocks less I/O costly. In this paper we define a new heuristic, *balance*, which prioritizes a partition that shrinks the memory requirement of PEMVI most significantly, thus it can help reach a valid partition as early as possible.

However, evaluating these heuristics requires computing these criteria, *e.g.*, percentage of intra-block transitions, for the reachable space, an endeavor too ambitious to undertake. We exploit our sampled state space to make this tractable. With the (unknown) partition block size l, the size of the sampled space in that block follows a Binomial distribution(l, x) under some reasonable assumptions. If sampling generates a subspace of size r, then l follows a negative binomial distribution(r, x), with mean $\mu = r/x$ and standard deviation $\sigma = \sqrt{r(1-x)}/x$. This distribution, according to the *central limit theorem* [Rice, 2001], is approximately *Normal*. The size l has a very slight probability ($\approx 0.1\%$) of being greater than $\mu + 3\sigma$. We use this number to be the upper bound while estimating l. We estimate the number of transitions into a block and other criteria in a similar fashion.

Given a heuristic (any of these above), ideally, we will enumerate all partitions and evaluate them based on the heuristic value to get the best partition. Unfortunately, if there are n XOR groups in the domain, the total number of possible partitions is 2^n. Moreover, evaluating a partition, even though made feasible by the use of sampled state space, is still quite time-consuming (see Section 5 for details). For this reason, we trade off optimality for time and use greedy search by picking the XOR group that has the best heuristic estimate at each partitioning step. In the experiments section we empirically compare these three heuristics and develop a strategy to maximize the algorithm performance.

Sometimes the partitioning process itself exhausts the available memory. The *block transition table* is a table that represents the successor block relations between all partition blocks. In our implementation, we represent the table as an adjacency list, and store it in memory. When the algorithm generates too many partition blocks then the block transition table overflows and we terminate the algorithm with failure.

4.4 Encoding

Based on our set of XOR groups we can encode the state space into a compact representation. For ease of illustration we describe the case when all domain features are Boolean. In such a case a naive encoding will use a bit-string whose length equals the number of the features. To reduce this further we can define a single multi-valued feature for an XOR group of k literals and the total number of bits needed for that feature will be $\log_2 k$ and we will save on $k - \log_2 k$ bits. In our encoding algorithm, we define these features greedily for each XOR group and remove all the participating ground predicates. We continue this process until all XOR groups are exhausted. We are left with a set of multi-valued features that we newly defined and some Boolean features (for the original ground predicates that didn't participate in any XOR group). We use this representation for a compact encoding of a state.

5 Experiments

We address the following questions: (1) How do the different heuristics (*viz.* locality, coherence and balance) compare with each other? (2) Does our general algorithm for finding XOR groups produce better XOR groups compare to EH algorithm? Does our algorithm scale? (3) How does the quality of automatically-generated partitions compare to those that are manually generated? (4) Does automatic partitioning

Algorithm 3 Search for Partition

```
1:  input: $\mathcal{S}$ (state space), $\mathcal{R}$ (the set of XOR groups)
2:  // $Ptn$ means a partition, $bk$ means a partition block
3:  $Ptn \leftarrow \{bk | bk = \mathcal{S}\}$
4:  $backtrack \leftarrow false$
5:  $d \leftarrow 0$
6:  while $Ptn$ is not valid do
7:    if $backtrack = false$ then
8:      pick a greedy $r \in \mathcal{R}$ using $coherence$ heuristic
9:      $d \leftarrow d + 1$
10:   else
11:     pick a greedy $r \in \mathcal{R}$ using $balance$ heuristic
12:   modify $Ptn$ by partitioning $Ptn$ using $r$
13:   if block transition table overflows then
14:     $backtrack \leftarrow true$
15:     if $d \geq 0$ then
16:       $d \leftarrow d - 1$
17:       backtrack to partition depth $d$ AND goto 6
18:     else
19:       return fail
20: return $Ptn$
```

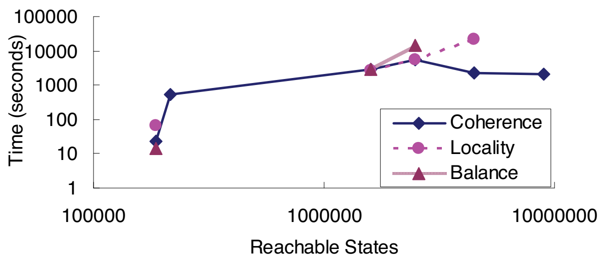

Figure 2: Running time of PEMVI using partitions created by different heuristics (missing points means problem not solved).

scale to large problems in IPC domains? (5) Does the use of simple-negated XOR groups benefit the partitioning process?

We implemented our automatic partitioning algorithms and PEMVI using C++. We evaluated our planner using an Intel(R) Core(TM)2 CPU processor with 2GB RAM and a 60GB SATA II (300MB/s) hard disk.

Comparing Heuristics: We first performed control experiments in order to determine which of locality, coherence and balance is the best heuristic. Using a suite of six problems from three domains, we ran PEMVI with partitions generated by following each heuristic's guidance. We measured overall running time (partition generation + policy construction) with a cut-off of 10 hours. Figure 2 shows the time taken on a log scale. Apart from the smallest problem, coherence outperformed the other heuristics, which often failed to generate partitions that could solve the problem within the time limit.

Moreover, when we tested on even larger Explosive-Blocksworld problems, we observed that none of the heuristics enabled PEMVI to generate a solution within 10 hours. However, balance seemed to get closest in the sense that it was able to generate a valid partition (just not one that could be solved within the limit). In contrast, the other heuristics exhausted memory while just trying to find a valid heuristic.

These and other observations suggest a hybrid strategy. We partition greedily using the coherence heuristic, until either we find a valid partition or we reach a state where further partitioning will cause overflow of the block transition table. In this latter case we backtrack, trying the balance heuristic

Algo	Blocksworld	Drive	Elevator	Schedule
Old	1	2	1	3
New	1	3	2	3
	Ex-Blocksworld	Tireworld	U-Drive	Zeno
Old	3	1	2	3
New	3	1	3	5

Table 1: Number of XOR formulae found by Edelkamp & Helmert (Old) and ours (New) in IPC-06 domains.

	CPU (s)	I/O (s)	Memory (MB)
Manual partition	2,894	1,489	1,681
Automatic partition	2,414	1,729	1,574

Table 2: PEMVI Running time and Memory usage.

at the previous decision points. We use this hybrid scheme in all further experiments (Algorithm 3).

Comparing domain analysis algorithms: We now compare the performance of our new algorithm for finding XOR formulae (Algorithm 1) with the EH algorithm. For all the problems that we tried, both algorithm ran very fast (finished in less than 1 second). As expected, the new algorithm consistently found a superset of XOR formulae. Table 1 shows that our algorithm finds additional XOR formulae in 50% of the domains. With the new set of XOR formulae, PEMVI managed to solve Zeno p5 in 23% of the time, with 53% of memory compared to using the smaller set of XOR formulae found by the EH algorithm.

Quality of Automatic Partitioning: We now compare the partition generated by our algorithm with the Explosive Blocksworld partition devised manually in our previous work [Dai *et al.*, 2008]. That partition recursively applied a grounding of the $\exists_1 b\ [clear(b') \oplus on(b, b')]$ XOR group, using a new block b' at each level.

In contrast, automatic partitioning starts by picking the same XOR group, but at the next level it picks a group that differentiates different locations of b', namely $\exists_1 b\ [holding(b') \oplus on(b', b)]$ (whether b' is in hand, or on top of another block). By considering these two related XOR groups as a pair, the resulting partition gets a better performance, as we show below.

We compare the two partition schemes by averaging over four 7-block Explosive Blocksworld problems, whose average number of reachable states is 9,649,979. Table 2 compares the performance of the two partitioning approaches, which each found a valid partition at level 2 on every problem. We observe that on average automatic partitioning solves problems slightly faster (5.5%) and with slight less memory (6.4%). When tested on even larger (8- and 9-block) problems, manual partitioning failed completely, overflowing its block transition table. In contrast, automatic partitioning solved these problems handily (Table 3).

Scalability: We evaluated our system on problems from 8 domains from IPC-06 [ipc, 2006]. Since problems in Drive and Unrolled-Drive are extremely small, we do not report those results. Table 3 lists some of the large problems solved by our algorithm.[3] For each problem, we also report the size of its *reachable* state space, time spent performing reachability and partitioning, I/O time and CPU time consumed by

[3]The ones with * are not original IPC problems

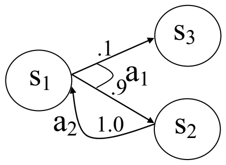

Figure 3: Example of an MDP which would be expensive to solve if s_1 is mistakenly placed in a partition which doesn't contain s_2.

PEMVI, as well as the planner's peak memory usage. As an external memory algorithm, the convergence of PEMVI on these problems is relatively fast (usually within a couple of hours). This shows the great power of our automated techniques in solving large probabilistic planning problems.

Simple-Negated XOR Groups: We also find that very few XOR groups are present in some domains. For them we decided to use *simple-negated XOR groups*, of the form $\{l, \neg l\}$, where l is a grounded literal. This modification especially helps in Blocksworld and Tireworld, since they have very few XOR groups with complex structure.

Results show that, for the four Tireworld problems we tried, using simple-negated XOR groups lead to faster convergence (average 50.0% speedup, but most useful on small problems). For the Blocksworld domain, failure to consider simple-negated groups does not slow convergence on small problems (e.g., 5-blocks). However, on larger problems (e.g. 6-blocks and larger), we were unable to find a solution without using simple-negated groups. We conclude that simple-negated groups is very important in generating a partition, especially when the non-trivial XOR groups are too sparse. So, overall their use makes the algorithm more robust to a wide variety of domains.

6 Related Work

Automatic partitioning in logically-specified planning domains was introduced by Zhou and Hansen [2006]. They use a partition to reduce the size of the states that are checked for finding duplicate states during systematic, external-memory searches. Static partitioning over non-logical probabilistic domains was used by Wingate and Seppi [2005]. They manually form a partition and solve each partition block iteratively, in order to expedite classical dynamic programming.

Our idea of automatic partitioning is related to variable resolution [Moore and Atkeson, 1995; Dean *et al.*, 1997; Munos and Moore, 2000; 2002]. Variable resolution tries to automatically and dynamically group similar states as an abstract state, solves the abstract MDP, and uses the solution to the abstract MDP as an approximation. However, we aim to solve the original MDP optimally, and thus our approach differs from them in a fundamental way. Our emphasis is on creating a partition that minimizes the total time (I/O + backup) for optimal policy construction whereas they aim at partitions that achieves better approximation. However, dynamic partitioning is not I/O efficient, and does not guarantee that the abstract MDP fits in the memory.

7 Limitations and Future Work

PEMVI can be very sensitive to the quality of partition. For instance, consider Figure 3, in which $\mathbf{s_1}$ and $\mathbf{s_2}$ form a cycle with probability 0.9. Dynamic programming may need

Problem	Reachable States	Reachability	Partition	PEMVI (I/O)	PEMVI (CPU)	Memory (M)
Blocksworld p5	103,096	47	40	120	872	148
Elevator p15	538,316	990	893	220	898	877
Schedule p5	5,062,359	1,984	1,893	2,413	1,161	1,980
Zeno p5	5,223,057	3,456	3,187	1,340	13,918	448
Ex-Blocksworld α*	21,492,777	8,215	8,574	5,267	806	1,212
Tireworld p15	29,664,846	9,908	8,534	27,423	6,197	1,359
Ex-Blocksworld β*	42,899,585	16,656	16,059	10,926	1,527	580

Table 3: PEMVI running time (in seconds) and memory usage on some large problems in IPC-06 domains.

many iterations of backups for computing the correct values for both states. If $\mathbf{s_1}$ and $\mathbf{s_2}$ end up in two different partition blocks, this translates into a large number of I/O iterations, which can be very costly! Note that if the two states are in the same block it is not a problem, since PEMVI already performs multiple backups within a block in each I/O iteration. In our experiments, we found that cycles of this type are pervasive especially in Explosive-Blocksworld problems. In practice, our automatic partitioning algorithm sometimes falls into traps and finds a partition that cannot be solved. In the future, we plan to resolve this issue by exploring dynamic repartitioning. If one partition fails, *i.e.*, is taking a long time to converge, we can detect the cycles responsible for slow convergence and generate a new valid partition in which the states on those cycles are assigned to the same partition block.

8 Conclusions

This paper makes several contributions. First, we demonstrate the practicality of automatically partitioning probabilistic planning problems by implementing a completely autonomous, domain-independent MDP solver, which can optimally handle as yet unsolved problems in several IPC-06 domains.

Second, our algorithm uses a novel approach for generating XOR groups, which generalizes those of [Edelkamp and Helmert, 1999; Zhou and Hansen, 2006].

Third, we propose a new heuristic, *balance*, for guiding the search for partitions and introduce a sampling method for its efficient computation.

Finally, we present several experiments: comparing three partitioning heuristics, comparing our algorithm to compute XOR groups with previous work, showing that automatic partitioning can beat manual methods, and demonstrating the utility of simple-negated groups.

Acknowledgments

This work was supported by Office of Naval Research grant N00014-06-1-0147 and the WRF / TJ Cable Professorship. We thank the three anonymous reviewers and Andrey Kolobov for their suggestions on improving the paper.

References

[Aberdeen *et al.*, 2004] Douglas Aberdeen, Sylvie Thiébaux, and Lin Zhang. Decision-theoretic military operations planning. In *ICAPS*, pages 402–412, 2004.

[Barto *et al.*, 1995] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *AI J.*, 72:81–138, 1995.

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

[Bresina *et al.*, 2002] John L. Bresina, Richard Dearden, Nicolas Meuleau, Sailesh Ramkrishnan, David E. Smith, and Richard Washington. Planning under continuous time and resource uncertainty: A challenge for AI. In *UAI*, pages 77–84, 2002.

[Dai *et al.*, 2008] Peng Dai, Mausam, and Daniel S. Weld. Partitioned external memory value iteration. In *AAAI*, 2008.

[Dean *et al.*, 1997] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *UAI*, 1997.

[Edelkamp and Helmert, 1999] Stefan Edelkamp and Malte Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*, pages 135–147, 1999.

[Edelkamp *et al.*, 2007] Stefan Edelkamp, Shahid Jabbar, and Blai Bonet. External memory value iteration. In *ICAPS*, 2007.

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

[ipc, 2006] 2006. http://www.ldc.usb.ve/ bonet/ipc5/.

[Korf, 2003] Richard E Korf. Delayed duplicate detection: Extended abstract. In *IJCAI*, pages 1539–1541, 2003.

[Littman *et al.*, 1995] Michael L. Littman, Thomas Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *UAI*, pages 394–402, 1995.

[Moore and Atkeson, 1995] Andrew W. Moore and Christopher G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21:199–233, 1995.

[Munos and Moore, 2000] Rémi Munos and Andrew Moore. Rates of convergence for variable resolution schemes in optimal control. In *ICML*, pages 647–654, 2000.

[Munos and Moore, 2002] Rémi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49(2-3):291–323, 2002.

[Rice, 2001] John A. Rice. *Mathematical Statistics and Data Analysis*. 2001.

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

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