Proceedings of the Twenty-Sixth International Florida Artificial Intelligence Research Society Conference

Improving Decision Diagrams for Decision Theoretic Planning

Jean-Christophe Magnan and Pierre-Henri Wuillemin
Laboratoire d'Informatique de Paris VI
firstname.lastname@lip6.fr

Abstract

In the domain of decision theoretic planning, the factored framework (FMDP) has produced optimized algorithms using Decision Trees (SVI, SPI) and Algebraic Decision Diagrams (SPUDD). However, the state-of-the-art SPUDD algorithm requires i) the problem to be specified with binary variables and ii) the data structures to share a common order on variables. In this article, we propose a new algorithm within the factored framework that eliminates both these requirements. We compare our approach to the SPUDD algorithm. Experimental results show that our algorithm allows significant gains in time, illustrating a better trade-off between theoretical complexity of algorithms and size of representation.

Introduction

In *Decision Theoretic Planning*, *Markov Decision Process* (MDP) is an important modeling tools. This framework models a decision problem as sets of states and actions, probabilistic transitions from state to state under actions and rewards triggered by specific states on certain transitions. Two main algorithms, *value iteration* (VI) and *policy iteration* (PI), exploit this representation and enable to efficiently solve planning problems. Indeed, these algorithms have a linear complexity in regard of the state space size (Puterman 1994).

Unfortunately, state spaces for realistic problems are often too large to find out optimal solutions in reasonable time. To deal with that issue, some promising solutions have emerged around the concept of *abstraction* : considering and treating similar states together leads to a reduction in the complexity of the state space enumeration. Several variants of MDPs use that approach, for instance hierarchical MDPs (Guestrin 2002) or Factored MDPs (FMDPs: Boutilier, Dean, and Hanks 1999).

FMDPs rely on the description of the system by a set of variables. Each state of the system is represented by a unique instantiation of those variables. The different inputs of the model are then factored: probability distributions with dynamic Bayesian networks (dBNs: Dean and Kanazawa 1989) and rewards with additive decomposition. However, conditional probability tables used in a dBN can still grow exponentially. Furthermore, the whole enumeration of the state space during VI and PI remains an issue.

Boutilier, Dearden, and Goldszmidt proposed a solution to those problems using decision trees to represent conditional probability tables, and reward functions. Such compact representation of each function allows to design efficient algorithms that avoid the exhaustive enumeration of all states: based on VI and PI, *Structured Value Iteration* (SVI) and *Structured Policy Iteration* (SPI) significantly improve the experimental time and space complexity. To further improve these complexities, Hoey et al. proposed another and more compact graphical representation : *Algebraic Decision Diagrams* (ADD). ADDs are a reduced version of decision trees where isomorphic subgraphs are merged together (Bryant 1986). From there, research in this domain has mainly focused on gains obtained by approximations : approximated ADDs (St-aubin, Hoey, and Boutilier 2000), basis functions (Guestrin, Parr, and Venkataraman 2003), etc.

However ADDs hold some drawbacks. First, all the variables in the model have to be binary. Variables with three or more values have to be decomposed in binary variables to be inserted in ADDs. This recasting artificially increases the state space. The second drawback is that ADDs compactness strongly depends on their inherent order on variables. Finding an order on variables that would produce an optimally compacted ADD is NP-hard (Friedman and Supowit 1990). Moreover, the algorithm for operations on two ADDs (for instance addition, multiplication or maximization) requires that orders on those ADDs are the same. As a consequence, a global order on variables is needed for the algorithm and can be largely sub-optimal for certain ADDs, leading to an artificial increase in the complexity for these operations.

The purpose of this paper is to propose new ways to adress those issues. Its main contributions are i) to propose a more efficient data structure to represent functions of multi-valued variables, and ii) to present a new algorithm for operations on two decision diagrams without imposing a common order on both diagrams. This article is organized as follows: Section 2 covers MDPs and the use of compact representation to find optimal policy. ADDs and the limitations they brought will be tackle there, as well as the new model we propose to use instead. Section 3 presents the new algorithm for the operation on two decision diagrams. Finally, the experimentations in Section 4 illustrate the efficiency of these improvements.

Factored Markov Decision Processes

We assume that our planning problem can be modeled by a fully-observable MDP. Let S and A be respectively the set of states and the set of actions. The conditional probability $P(s|a, s')$ represents the probability of reaching state s' when action a is performed in state s. In order to simplify the presentation, the reward is formalized as a real-value function $R(s)$ depending only on the current state s.

A stationary policy $\pi : S \rightarrow A$ defines the action a to execute when system reaches the state s. Assuming that the objective is to find an optimal policy over an infinite time horizon, we can compare two policies upon the *expected total discounted reward* defined on every state:

$$V_\pi(s) = R(s) + \gamma \sum_{s' \in S} P(s, \pi(s), s') \cdot V_\pi(s') \qquad (1)$$

where γ is used to discount future rewards. V_π is called the value function for policy π. An optimal policy π^* is a policy that verifies: $\forall \pi, \forall s \in S, V_{\pi^*}(s) \geq V_\pi(s)$.

To find the optimal policy, VI algorithm consists in converging toward optimal value function by iterative updating:

$$\begin{aligned} V^0(s) &= R(s) \\ V^{n+1}(s) &= R(s) + \gamma \cdot \max_{a \in A} \sum_{s' \in S} P(s'|a, s) \cdot V^n(s') \end{aligned} \qquad (2)$$

At each iteration of the algorithm, V^n has to be updated for every state $s \in S$. The linear complexity of each iteration is an issue because S happens to be large even for simple problems. A solution is to enhance the compactness of functions $P(.|s, a)$, R and V with graphical and factored representations using discrete variables.

Let X_i be a multi-valued variable taking its values over a finite discrete domain D_{X_i}. Whenever X_i is instantiated (i.e. set to a given value in D_{X_i}), it will be noted x_i.

Definition (Decomposability of state space).

A state space S is decomposable if and only if there exists a set of discrete and finite variables $X = \{X_1, \dots, X_n\}$ that unequivocally characterizes S.

Each state $s \in S$ is then an instantiation of those variables ($s = \{x_1, \dots, x_n\}$).

When S is decomposable, transition probabilities, value function and rewards are expressed as functions of $\{X_1, \dots, X_n\}$ and, as a consequence, can be compactly represented by function graphs.

Let f be a function over the variables $X_1, \dots, X_n$. We denote $f|_{X_i=b}$ the restriction of f on $X_i = b$. The *support* of f is the set of variables that f really depends on, i.e.,

$$\mathrm{support}(f) = \{X_i \mid \exists u, v \in D_{X_i} \text{ s.t. } f|_{X_i=u} \neq f|_{X_i=v}\}$$

Note that $\forall X_i$, $\mathrm{support}(f|_{X_i=b}) \subseteq \mathrm{support}(f) \setminus \{X_i\}$. Indeed the support of the restriction discards all non relevant variables and not only the variable X_i.

Definition (Function Graph).

Let f be a function over $\{X_1, \dots, X_n\}$. A directed acyclic graph (DAG) $G_f(N, A)$ is a function graph of f

- *if f constant then G_f has a unique node $r \in N$, $r.val = f$*
- *if f non constant then G_f has a unique node $r \in N$ without parent. Moreover,*
  - $r.var \in \mathrm{support}(f)$
  - $\forall u \in D_{r.var}, \exists! n_u \in N$ *such that:*
    - $(r, n_u) \in A$
    - $\mathrm{subgraph}(n_u) = G_{f|_{r.var=u}}$

Note that in a function graph, if a node n is terminal (i.e. without children) then n is bound to a constant value ($n.val$) else n is associated to a variable ($n.var$).

For any node n of G_f, $\mathrm{subgraph}(n)$ is a function graph for the restriction of f defined by the instantiation of every variables crossed on the path from the root to n. We note $f|_n$ this restriction. Note that several path may lead to the same subgraph, characterizing the fact that several restrictions may be equal. For instance, in Figure 1(c),

$$f|_{\text{terminal node } 2} = f|_{X=0,Y=2} = f|_{X=1,Y=1} = 2$$

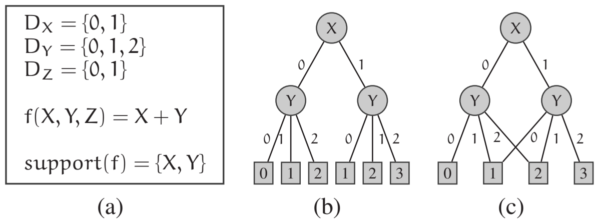


Figure 1: Two function graphs for the same function.

As shown in Figure 1, there may be several function graphs for one function. But they all share a same property of compactness: no irrelevant variable can appear in a function graph due to the use of $\mathrm{support}(.)$. Thus, two states that share the same value for represented function but differ on an irrelevant variable will be viewed as a single abstract state by that representation.

Functions graphs give a compact and efficient way to represent the whole state space. Furthermore, dedicated algorithms for operations (addition, multiplication, maximization) on function graphs enable to create composed functions without an explicit enumeration of all states.

Function Graphs in MDPs

The optimal policy search using VI algorithm can be reformulated using function graphs. Indeed, let V, R and P_a be respectively the function graphs for the value function, the reward, and the probabilistic transitions for each action $a \in A$. The update will be done exactly like in equation 2. The only differences are the data structures and the dedicated operations used at each iteration :

$$V^0 = R \quad \text{and} \quad V^{n+1} = R + \gamma \cdot \max_{a \in A} \sum_{s' \in S} P_a \cdot V^n$$

In a factored MDP (FMDP: Boutilier, Dearden, and Goldszmidt 1995), the state space is decomposable. Besides, the assumption of some conditional independences allow to factorize the probability distribution: $P_a(s'|s) = \Pi_i P_a(X'_i|s)$.

The first data structure used as function graph for optimal policy search was the *decision tree* (as in Figure 1(b)). SVI and SPI rely on such data structure and on the specific algorithm for operations on it (Boutilier, Dean, and Hanks 1999).

As shown in Figures 1(b), decision trees may contain several isomorphic subgraphs. Indeed due to the tree structure, those subgraphs can not be merged. That duplication unnecessarily increases the graph size.

Rather than tree, Hoey et al. proposed then to use *Algebraic Decision Diagrams* (Bahar et al. 1993) as function graphs (as in Figure 1(c)). ADDs are a generalization of *Binary Decision Diagrams* (BDDs: Bryant 1986) used to represent real functions of boolean variables ($f : \mathbb{B}^n \rightarrow \mathbb{R}$). Their particularities are that they are reduced and ordered.

Definition (Reduced Function Graph).
G_f *is reduced* $\iff \forall$ *nodes* $n \neq n'$, $f|_n \neq f|_{n'}$

When a function graph is reduced, two isomorphic subgraphs are necessarily merged together.

Definition (Ordered Function Graph).
A function graph G_f *is ordered* $\Leftrightarrow \exists \succ_{G_f}$ *complete order on* $\mathtt{support}(f)$, *s.t.* $\forall$ *nodes* n_1, n_2 *non terminal of* G_f,

$$n_2 \in \mathtt{desc}(n_1) \Rightarrow n_1.var \succ_{G_f} n_2.var$$

When a function graph is ordered, the algorithm for reducing it is polynomial.

Bryant describes the algorithm for the operations on BDDs. The ADDs algorithm relies on the same principle. In SPUDD, Hoey et al. proposes a version of VI algorithm using ADDs as data structure.

However, ADDs represent only functions of boolean variables. As a consequence, all multi-valued variables have to be encoded with binary variables. A first issue is raised by such an encoding ; the state space size is artificially increased. Indeed to code a variable with n values, one needs $\lceil log_2 n \rceil$ binary variables. Those $\lceil log_2 n \rceil$ variables will generates $\lceil 2^{log_2 n} \rceil$ possibles states. Which means that $\lceil 2^{log_2 n} \rceil - n$ states are artificially created and have no real existence. Yet the algorithms (VI, PI, etc.) will search an optimal policy for those states.

In this article, we investigate the use of *Multi-valued Decision Diagrams* (MDDs: Srinivasan et al. 1990). MDDs simply generalize the concept of ADDs to multi-valued variables (see for instance, Figure 1(c) where Y is ternary). The artificial increase in variables is then avoided, reducing therefore both graph compactness and computation time.

We also propose to add a default arc mechanism : wherever a majority of arcs are pointing to the same node, those arcs are replaced by a default arc. This had the advantage of simplifying the graph and easing the computations. For instance, in the MDD of Figure 4(a), four default arcs (dashed lines) replace fourteen arcs.

As already evoked, the second drawback of ADDs (and of any reduced and ordered function graphs, i.e. MDDs) is that their compactness strongly depends on the inherent variable order. However, the current algorithm for operations on ADDs impose that they share the same order. As a consequence, that common order may not be the optimal one for each of them or for the result of the operation. Taken that an operation between two ADDs D_1 and D_2 is in $\mathcal{O}(|D_1| \cdot |D_2|)$, this common order tends to arbitrarly increase the computation complexity.

In the next section, we describe a new algorithm of combination on MDDs that will not impose such a common order.

Operations on Decision Diagrams

Let G_1, G_2 and G be three reduced and ordered function graphs (BDDs, ADDs or MDDs) such that $G = G_1 \odot G_2$ ($\odot$ being either addition, or multiplication or maximization). We're looking for an algorithm that build G from G_1 and G_2, without imposing that orders $\succ_1$ from G_1 and $\succ_2$ from G_2 are the same. First, we will shortly present the state-of-the-art algorithm with common order (for further details, please refer to (Bryant 1986)).

Operations with Common Order Constraint

Let $\succ$ be the common order imposed on G_1, G_2 and G for the operation. The algorithm of combination relies on simultaneous depth-first explorations of both G_1 and G_2. Each step of the algorithm is a recursive call to a same function. This function takes two node $n_1 \in G_1$ and $n_2 \in G_2$, starting from the root of both graphs. First, it determines how they should be explored :

1. if n_1 and n_2 are both terminal, then $n_1.val \odot n_2.val$ is computed,
2. if only one node is non-terminal, then exploration is only done on this node,
3. if both nodes are non-terminal then
   a. if $n_1.var = n_2.var$, the exploration is done simultaneously on both nodes,
   b. if $n_1.var \succ n_2.var$ (resp. $n_2.var \succ n_1.var$), then the exploration is done only on n_1 (resp. n_2).

Exploration consists in calling the function again on each child of the visited node n (one for each value of $n.var$, n being either n_1, or n_2). The called node from the other diagram remains unchanged, unless it is simultaneously explored. In which case, it is its child selected by the current value of $n.var$ that is called upon.

When exploration on every children is over, or a value has been computed, a node n_G is inserted in G. If a value was computed, it is bound to that terminal node. Otherwise, variable associated to n is bound to n_G. The children of n_G are the resulting nodes from the explorations on n children. This procedure ensures that every variables will be in the correct order in G.

Due to the assumption of a common order, the algorithm is quite simple. Its complexity is clearly in $\mathcal{O}(|G_1|.|G_2|)$ when both G_1 and G_2 are trees. For BDDs, ADDs and MDDs, a pruning mechanism is needed in order to keep this complexity (see below for more details).

Building an Order for Resulting Decision Diagram

When removing the common order constraint, we have to deal with different orders: $\succ_1$ for G_1, $\succ_2$ for G_2 and $\succ_G$

for $G = G_1 \odot G_2$. The question about how to build the order $\succ_G$ immediately raises.

With the objective of still performing a deep-first recursive and simultaneous exploration on G_1 and G_2, one has to analyze a new case at each step: let $n_1 \in G_1$ and $n_2 \in G_2$ be the considered nodes at any step. It may now happen that $n_1.var \succ_1 n_2.var$ and $n_2.var \succ_2 n_1.var$. In that case, the questions are which variable will precedes the other in $\succ_G$ and in which order do we perform exploration on n_1 and n_2.

Definition (Retrograde Variable).

Let $\succ_1$ and $\succ_2$ be two orders on a set of variables X. A variable $X_r \in X$ is said to be retrograde in $\succ_2$ w.r.t. $\succ_1$ if $\exists X_p \in X$ s.t. $X_r \succ_1 X_p$ and $X_p \succ_2 X_r$.

Corollary. *X_p is retrograde in $\succ_1$ w.r.t $\succ_2$ because of X_r (at least).*

We note the set of retrograde variables:
$\mathfrak{R}_{1,2} = \{X_i \in X, X_i \text{ retrograde in } \succ_2 \text{ w.r.t. } \succ_1\}$.

$D_{\mathfrak{R}_{1,2}}$ is the domain of that set. The size of that domain is $|D_{\mathfrak{R}_{1,2}}| = \prod_{X_r \in \mathfrak{R}_{1,2}} |X_r|$.

Note that in general $\mathfrak{R}_{1,2} \neq \mathfrak{R}_{2,1}$ and $|D_{\mathfrak{R}_{1,2}}| \neq |D_{\mathfrak{R}_{2,1}}|$.

Whenever $\mathfrak{R}_{1,2} \neq \emptyset$, our algorithm will have to select for $\succ_G$ to be compatible with $\succ_1$ or $\succ_2$. We propose to arbitrarily privilege $\succ_1$ (see below for a discussion about that choice).

$\succ_G$ will then have the following properties: (i) $\succ_G$ extends $\succ_1$ and (ii) $\succ_G$ extends $\succ_{2\setminus\mathfrak{R}_{1,2}}$. These two properties are sufficient to build $\succ_G$ from $\succ_1$ and $\succ_2$.

Exploration and construction

Now that G is ordered by $\succ_G$, its construction upon exploration of G_1 and G_2 can be performed. As for the algorithm with common order constraint, exploration will be recursive and simultaneous on both decision diagrams.

Since $\succ_G$ extends $\succ_1$, the set of retrograde variables that may be encountered is $\mathfrak{R}_{1,2}$.

By definition, for all $X_r \in \mathfrak{R}_{1,2}$, there exists at least one variable $X_p \in X$ which verifies that $X_r \succ_G X_p$ but $X_p \succ_2 X_r$. Note that occurrences of X_p preceeding X_r only happen in G_2.

Lemma. *During any recursive algorithm to build G, any exploration on variable X_r must have begun before any exploration on variable X_p.*

Proof. Exploration is performed by recursive calls and a resulting node is created at the end of each call. That node has precedence over all the nodes recursively created during that call.

Then, in order to verify that $X_r \succ_G X_p$, any needed recursive call on X_r must have begun before any recursive call on nodes bound to X_p starts. □

This lemma implies that our algorithm will have to possibly anticipate an exploration on X_r whenever an exploration on X_p is required in G_2. This has a consequence in term of complexity that will be analyzed below.

An anticipated exploration of X_r on a node n_2 consists in performing a normal exploration on an artificially inserted node n_r such that $n_r.var = X_r$ and all its children are n_2.

The end of this section technically characterizes the specific situation where this anticipation is required. Table 1 presents the core function for the exploration and the different cases it has to deal with.

Let (n_1, n_2) be the visited nodes at current step of our algorithm. If $subgraph(n_2)$ contains X_r[1], the algorithm possibly have to anticipate an exploration on X_r. On the contrary, if $subgraph(n_2)$ does not contain X_r, the algorithm can normally perform the exploration on n_2.

An exploration on X_p in G_2 is required if and only if $n_2.var = X_p$ and $n_2.var \succ_G n_1.var$. Before that, exploration goes on normally. In particular, X_r can be crossed in G_1. It is then normally explored regardless its presence in G_2.

Assume that exploration on X_p is required and that X_r possibly have to be anticipated. Two cases can occur :

Case (a). *If $X_r \in$ current explored path on G_1 then X_r has already been instantiated. Hence no anticipated exploration on X_r is needed.*

Case (b). *If $X_r \notin$ current explored path on G_1 then X_r has not yet been explored. Hence the anticipated exploration of X_r is needed before exploring X_p on G_2.*

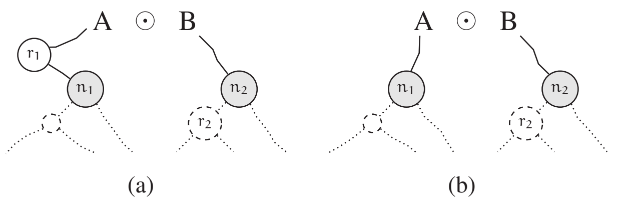


Figure 2: In case (a), no anticipated exploration is needed: X_r is already being explored on r_1. In case (b), anticipated exploration is needed: X_r has not been crossed. Current visited nodes are (n_1, n_2). $r_1.var = r_2.var = X_r$.

Note that both cases imply that during the explorations in G_2, a node bound to X_r will be crossed. The algorithm will then immediately skips onto the child selected by the current value of X_r.

Pruning and complexity

In the algorithm with common order constraint, several explorations of a same pair of subgraphs always lead to the same resulting structure. Paths that brought to their root do not alter the result. As a consequence, once a pair of nodes has been visited, the resulting node is stored in a table along with this pair as a key. Pruning consists then in looking for the pair of nodes in the table, and taking the result. This guarantees a complexity in $\mathcal{O}(|G_1| \cdot |G_2|)$, since every pair of nodes are visited only once.

In the new algorithm, pruning can't be done so easily. Suppose that for the current nodes (n_1, n_2), $n_2.var = X_p$ and $X_r \in subgraph(n_2)$. As evoked just above, when a node bound to X_r is encountered in G_2, exploration will automatically jump onto the child selected by the current value

[1]More precisely, $X_r \in support(subgraph(n_2))$

```
PROCEDURE $Explore(n_1, n_2, \text{FixedVars})$ :
  CASE: $n_1$.isTerminal and $n_2$.isTerminal
    RETURN $Terminal(n_1.val \odot n_2.val)$
  CASE: $n_2$ non terminal and $\exists n_r, n_r \in n_2.\mathtt{Descendants}$
                                   and $n_r.\text{var} \in \mathfrak{R}_{1,2}$
    $\forall$ modality $m \in D_{n_r.var}$ :
      $n_m$ = $Explore(n_1, n_2, \text{FixedVars} \cup \{n_r.var = m\})$;
    RETURN $NonTerminal(n_r.var, \mathtt{children} = \{n_m\}_{m \in D_{n_r.var}})$;
  CASE: $n_1.var \succ_G n_2.var$ or $n_2$.isTerminal
    $\forall$ modality $m \in D_{n_1.var}$ :
      $n_m$ = $Explore(n_1.\mathtt{child}(m), n_2, \text{FixedVars} \cup \{n_r.var = m\})$;
    RETURN $NonTerminal(n_1.var, \mathtt{children} = \{n_m\}_{m \in D_{n_1.var}})$;
  CASE: $n_2.var \succ_{1 \odot 2} n_1.var$ or $n_1$.isTerminal
    If $\exists m, n_2.var = m \in \text{FixedVars}$
      RETURN $Explore(n_1, n_2.\mathtt{child}(m), \text{FixedVars})$;
    Else
      $\forall$ modality $m \in D_{n_2.var}$ :
        $n_m$ = $Explore(n_1, n_2.\mathtt{child}(m), \text{FixedVars})$;
      RETURN $NonTerminal(n_2.var, \mathtt{children} = \{n_m\}_{m \in D_{n_2.var}})$;
  CASE: $n_1.var = n_2.var$
    $\forall$ modality $m \in D_{n_1.var}$ :
      $n_m$ = $Explore(n_1.\mathtt{child}(m), n_2.\mathtt{child}(m), \text{FixedVars})$;
    RETURN $NonTerminal(n_1.var, \mathtt{children} = \{n_m\}_{m \in D_{n_1.var}})$;
END PROCEDURE
```

Table 1: Operation between MDD without common order.

of X_r: only a part of $\mathtt{subgraph}(n_2)$ is then explored. The consequence is that $\mathtt{subgraph}(n_2)$ has to be re-explored for each value of X_r.

Unnecessary explorations can still be pruned : once an exploration is done for a value of the retrograde variable, there's no need to repeat this exploration. So the key used to know if a subgraph has already been visited simply has to be extended. It needs to indicate which value the retrograde variable had when the exploration was performed.

The multiple explorations due to retrograde variables affect the complexity of the algorithm. The increase in complexity is by the size of the domain of the retrograde variables $D_{\mathfrak{R}_{1,2}}$. Then complexity is now in $\mathcal{O}(|G_1| \cdot |G_2| \cdot |D_{\mathfrak{R}_{1,2}}|)$.

But this worst case complexity has to be pondered. Firstly our algorithm only re-explore subgraphs of G_2 when needed. Yet the given complexity is determined as if all the re-explorations concerned the whole graph. Unfortunately, a more accurate upper bound is difficult to obtain because it would demand a topological analysis of the graphs. Secondly, G_1 and G_2 have now their own order. Then the size of G_1 and G_2 can be smaller in this complexity than in the complexity of the algorithm with common order. Thirdly, $\succ_G$ is compatible either with $\succ_1$ or with $\succ_2$ (we arbitrarily chose $\succ_1$ for presentation purposes). As a consequence, we have to deal either with $D_{\mathfrak{R}_{1,2}}$ or with $D_{\mathfrak{R}_{2,1}}$. But since $|D_{\mathfrak{R}_{1,2}}| \neq |D_{\mathfrak{R}_{2,1}}|$, another tradeoff can be found here.

This discussion is confirmed in the experiments described in the next section.

Experiments and Observation

We have implemented MDD and SPUmDD (the implementation of our algorithm) using the aGrUM C++ library developed at LIP6[2]. Since the standard implementation of SPUDD uses an highly optimized library for ADD (but not for MDD), we have coded our own version of SPUDD in order to compare the algorithms on a time basis.

Meanwhile, size of computed diagrams is the most interesting measure. Indeed, as seen before, the complexity of operations depends on the size of the diagrams. The size of the computed value function is particularly relevant: on each iteration, value function is used to compute various other decision diagrams that are themselves aggregated using Equation 2. Therefore, the size of value function is a good indicator of the efficiency of representations.

No reordering is performed during execution in both algorithms. For SPUDD, the common order on variables is fixed and specified at the beginning. For SPUmDD, each MDD has its own order. That order is chosen so that the overall complexity to obtained the MDD is minimal (see previous section).

Table 2 shows result of value iteration using SPUDD and SPUmDD on various MDPs. State space size gives the total number of states (including the ones induced by binarization of multi-valued variables). Internal nodes gives the number of non-terminal nodes inside the computed value functions at last iteration (the number of terminal nodes is the same for both methods). And time (in second) is the average time to reach stopping criterion (ε was set to 10^{-5}) over 30 runs.

We examine the efficiency of our algorithm on two standard problems: *coffee robot* and *factory*. The interest of *coffee robot* planning problem is that it contains only binary variables. It allows to see if SPUmDD remains efficient on such cases. Results show that SPUmDD got same behavior than SPUDD. Yet it is slightly slower, showing that on purely and small binary problems ADDs are sufficient.

In *factory*, the interest resides in the mixed of binary and ternary variables. The conversion of ternary variables in binary variables generates an increase in the number of variables as much as an increase in state space size. Results shows clearly that advantage can be taken of by SPUmDD.

Notice that *factory1* and *factory2* only differ on one variable that is not relevant for value function (it has no incidence on other variable). Both *factories* got eventually the same structure, showing clearly that MDD can eliminate non relevant variables as ADDs does.

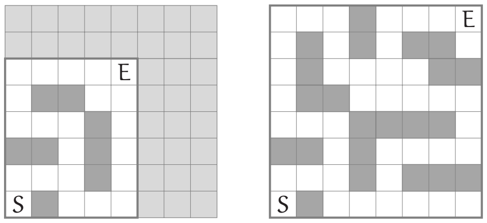


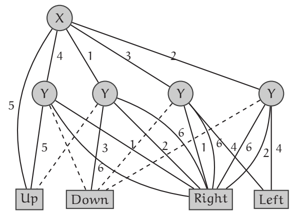
Figure 3: Maze examples: blocked cases are in dark gray. Impossible states generated by binarization in light gray.

To examine the behavior on problems with multi-valued variables, two mazes have been created. First maze (Figure 3) has 30 cases, 8 of them being blocked. It only requires

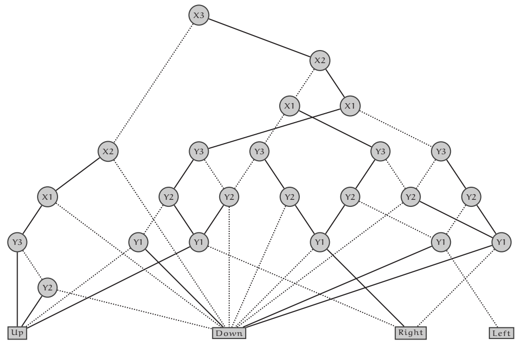
[2]Laboratoire d'Informatique de Paris VI

	SPUDD			SPUmDD			SPUmDD in comparison with SPUDD		
	States Space	Internal Nodes	Time (in s)	States Space	Internal Nodes	Time (in s)	States Space	Internal Nodes	Time
Coffee Robot	64	21	2.06	64	21	2.19	100.0%	100.0%	106.3%
Factory	131 072	1 269	1 751.45	55 296	473	661.93	42.2%	37.3%	37.8%
Factory 0	524 288	2 131	2 447.92	221 184	733	1 596.51	42.2%	34.4%	65.2%
Factory 1	2 097 132	2 889	8 796.10	884 736	1 283	5 181.56	42.2%	44.4%	58.9%
Factory 2	4 194 304	2 889	8 916.83	1 769 472	1 283	5 986.89	42.2%	44.4%	67.1%
Factory 3	33 554 432	3 371	27 240.00	10 616 832	2 001	17 439.90	31.6%	59.3%	64.0%
Maze 5x6	64	34	2.51	30	6	0.86	46.9%	17.7%	34.3%
Maze 8x8	64	58	3.32	64	9	1.05	100.0%	15.5%	31.6%

Table 2: Results using SPUDD and SPUmDD. The last columns illustrate the improvements in space and time using SPUmDD.



(a)



(b)

Figure 4: Maze Optimal Policy (a) with SPUmDD (dashed line stands for default arc) and (b) with SPUDD (dashed line from node X stands for $\overline{x}$).

two multi-valued variables (X and Y) of 5 and 6 modalities to represent its 30 possible states. However, its translation in binary variables demands 3 variables on each axes. Those variables generate a grid of 64 states where 34 are impossible. Second maze is a 8 by 8, and thus generates no impossible states on translation into a binary problem.

Here again, SPUmDD shows itself better than SPUDD, gaining both on time and size representation. Figure 4 shows the simplification of the policy obtained for the first maze with SPUmDD compared to SPUDD.

Conclusion

In this paper, we propose new improvements in the factored framework for decision theoretic planning.

First, we investigate the use of multi-valued decision diagrams which avoids the transformations of multi-valued variables into sets of binary variables. Such transformations increase the diagram size and constrain the algorithm to deal with states that are impossible for the modeled system. Our extension to multi-valued decision diagrams allows computations on simpler structures.

We then propose a new algorithm for operations on such decision diagrams that eliminates the second main drawback of the state-of-the-art. Our algorithm does not force the diagrams to share a common order of variables. In spite of an increase in worst case complexity, SPUmDD shows a significant gain in time and in size when compared to SPUDD.

With these improvements, the optimal policy search operates directly on the inputs of the problem (no binarization, no reordering) and produces more readable solutions. This is an opening for future works on incremental algorithms. Particularly, we will look further in dynamical reordering for the policies and incremental learning of the model.

References

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR* 11:1–94.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *IJCAI-95, pp.11041111*.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35:677–691.

Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Comput. Intell.* 5(3):142–150.

Friedman, S. J., and Supowit, K. J. 1990. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput.* 39(5):710–713.

Guestrin, C.; Parr, R.; and Venkataraman, S. 2003. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research* 19:399–468.

Guestrin, C. 2002. Distributed planning in hierarchical factored mdps. In *In Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, 197–206.

Hoey, J.; St-aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 279–288. Morgan Kaufmann.

Puterman, M. L. 1994. Markov decision processes: Discrete stochastic dynamic programming.

Srinivasan, A.; Kam, T.; Malik, S.; and Brayton, R. K. 1990. Algorithms for discrete function manipulation. In *ICCAD'90*, 92–95.

St-aubin, R.; Hoey, J.; and Boutilier, C. 2000. Apricodd: Approximate policy construction using decision diagrams. In *In Proceedings of Conference on Neural Information Processing Systems*, 1089–1095.