Efficient Incremental Planning and Learning with Multi-Valued Decision Diagrams

Jean-Christophe Magnan, Pierre-Henri Wuillemin

Efficient Incremental Planning and Learning with Multi-Valued Decision Diagrams

Jean-Christophe Magnan[1], Pierre-Henri Wuillemin[1]

Sorbonne Universités, UPMC, Univ Paris 06, CNRS UMR 7606, LIP6, Paris, France

Abstract

In the domain of decision theoretic planning, the factored framework (Factored Markov Decision Process, FMDP) has produced optimized algorithms using structured representations such as Decision Trees (Structured Value Iteration (SVI), Structured Policy Iteration (SPI)) or Algebraic Decision Diagrams (Stochastic Planning Using Decision Diagrams (SPUDD)). Since it may be difficult to elaborate the factored models used by these algorithms, the architecture SDYNA, which combines learning and planning algorithms using structured representations, was introduced. However, the state-of-the-art algorithms for incremental learning, for structured decision theoretic planning or for reinforcement learning require the problem to be specified only with binary variables and/or use data structures that can be improved in term of compactness. In this paper, we propose to use Multi-Valued Decision Diagrams (MDDs) as a more efficient data structure for the SDYNA architecture and describe a planning algorithm and an incremental learning algorithm dedicated to this new structured representation. For both planning and learning algorithms, we experimentally show that they allow significant improvements in time, in compactness of the computed policy and of the learned model. We then analyzed the combination of these two algorithms in an efficient SDYNA instance for simultaneous learning and planning using MDDs.

Email address: pierre-henri.wuillemin@lip6.fr (Pierre-Henri Wuillemin)

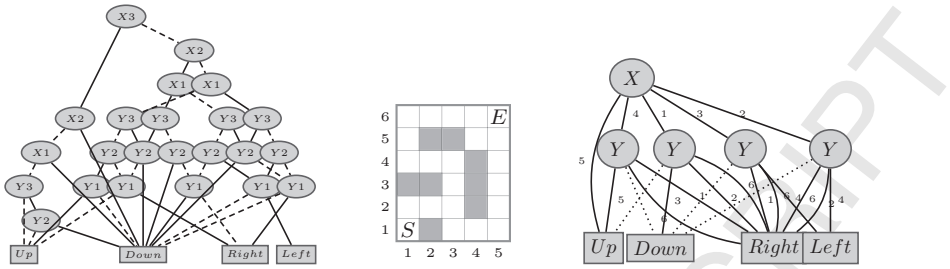
1. Introduction

In decision-theoretic planning, the Markov Decision Process (MDP) is a widely used framework that formalizes the interactions of an agent with a stochastic environment. A MDP is commonly used to find an optimal policy, i.e. the best action for the agent to do in each configuration of the environment (state). Two algorithms named Value Iteration (VI, Bellman, 1957) and Policy Iteration (PI, Howard, 1960) exploit such models to find optimal policies. Each step of these algorithms has a linear time complexity in the size of the state space (Puterman, 2005). However, the size of the state space tends to become very large for real problems. State spaces are indeed often multidimensional and then grow exponentially as the number of variables (dimensions) characterizing these problems increases. VI and PI inevitably fall under the Bellman's "Curse of Dimensionality" (Bellman, 1961). It becomes unfeasible to find the optimal solution.

Many relevant solutions have emerged to handle this growth: for instance by formalizing abstractions in the state space. Through these abstractions, large sets of states in which the agents mostly behave the same are aggregated. As a result, the number of states to visit during VI or PI iterations drastically decreases. Based on this idea, Factored MDPs have proven to be efficient. Structured VI (SVI) and Structured PI (SPI) (Boutilier et al., 1999) proposed to rely on Decision Trees (DTs) as a basis for this abstraction process. More recently, Stochastic Planning using Decision Diagrams (SPUDD, Hoey et al., 1999) achieved better results using Algebraic Decision Diagrams (ADDs). In this article, we describe algorithms that rely on an even more compact data structure: the Multi-Valued Decision Diagrams (MDDs)[1]. Figure 1 shows an ADD and an MDD and their respective compactness.

All of these algorithms need as a prior a known factored model of the problem to be solved. On the contrary, in the Reinforcement Learning framework,

[1] The algorithm SPUMDD has been initially described in (Magnan and Wuillemin, 2013)



(a) (b) (c)

Figure 1: An optimal policy to follow in order to go from S to E in the maze (b) represented as (a) an Arithmetic Decision Diagram (ADD) and as (c) a Multi-Valued Decision Diagram (MDD). The compactness as well as the readability of the solution are clearly improved with MDDs.

the agent has in charge to discover the world and to learn the optimal policy by itself. Model-based approaches such as DYNA and DYNA-Q (Sutton, 1990) propose that the agent iteratively learns a description of the world as a Markov Decision Process and then applies either VI or PI to come up with an optimal policy. Of course, these algorithms face difficulties when scaling up and it becomes necessary to provide the ability to handle factored representations. With SPITI, Degris et al. (2006a) extends the DYNA framework to factored representations. In SPITI, the DTs are used for the abstraction process. To learn the DTs defining a studied MDP as its agent experiences its world, the incremental learning algorithm ITI (Utgoff et al., 1997) is used. Then, optimal policies are planned using SVI or SPI. Yet, DTs are however less efficient than ADDs or MDDs in terms of size and, as a consequence, of handling (see Figure 1).

Learning ADDs or MDDs is difficult, particularly for systems with a huge number of states. It may even be impossible to get the database to execute the learning process on. Generally speaking, there are many cases where an on-line (or incremental) learning process is more appropriate. To the best of our knowledge there is no known algorithm for incremental learning of Multi-Valued Decisions Diagrams. One of the main contributions of this paper is an algorithm addressing this issue.

This article is organized as follows: Section 2 covers the frameworks for planning and incremental learning of compact and factored models. The next two sections describe our algorithms to plan with and to incrementally learn MDDs as well as the experimentations we lead to validate these algorithms. Finally, in Section 5, a new SDYNA instance based on MDDs is proposed.

2. Planning and Learning with Factored Models

This section briefly introduces the global framework for the Markov Decision Process and then describes more precisely the factored domain, the structured representations and the main algorithms that work on them.

2.1. Markov Decision Processes

A Markov Decision Process (MDP) formally describes a system in which an agent interacts with a stochastic environment (Puterman, 2005). At each (discretized) instant, the agent faces several choices that can alter the system. By performing these actions, the agent tries to resolve a task, to achieve a goal inherent to the system.

Each time the agent has chosen and performed a certain action a, the system moves from its current state s to a new one s'[2]. This change of state is called a transition. However, as the environment is stochastic, arrival state s' is uncertain. The MDP formalism makes two assumptions to address this uncertainty. First, there exists a probability to reach any future state s' given s, a and the history of the system. Next, the system follows the Markov property (Markov and Nagorny, 1988). As a consequence, the probability of moving from s to s' by doing a does not depend on the history of the system and can then be denoted $P(s'|a, s)$.

Eventually, the relevance of every transition is evaluated. Indeed, to achieve the inherent goal of the system, some transitions have to be performed, and some have to be avoided. A real number named reward is given to every transition in

[2]We follow the classical notation in the domain: s' stands for s of the future.

order to define its relevance to the task at hand. The function $R : (s, a, s') \to \mathbb{R}$ defines for every transition the associated reward.

In short, a MDP models a system as a tuple $\langle S, A, P, R \rangle$:

- S: the set of states (the state space);
- A: the set of actions;
- P: the transition probabilities function $P : S \times A \times S \to [0, 1]$;
- R: the reward function $R : S \times A \times S \to \mathbb{R}$.

Without loss of generality, we will consider in this article that the reward function only depends on the current state, i.e. $R : S \to \mathbb{R}$.

Given this model, the objective is to plan for the agent a course of action that will maximize its gained rewards. To establish this plan, the classical framework we will use makes these assumptions:

- discrete time: the agent proceeds step-by-step, resolving entirely an action before doing the next one;
- infinite horizon: the agent never stops acting;
- full observability: the agent has a perfect knowledge of the current state;
- discrete state space and actions (yet, these two sets can be very large).

2.1.1. Finding an optimal policy

A policy π explains how the agent must behave at every state. The policy π is then defined as the function[3] $\pi : S \to A$.

Several criteria are used to assess the efficiency of a policy. The criterion used in this study is the maximization of the expected sum of gained reward $\mathbb{E}[\sum_{t=0}^{\infty} R(s_t)]$. As we work at an infinite horizon, this sum may diverge. Moreover, it grants the same importance to rewards obtained far into the future in comparison to immediate rewards.

Rewards should be discounted according to how far into the future they are obtained. A discount factor γ ($\gamma \in [0, 1]$) is defined for this. The value function V^{π} defines then for every state s the *total discounted and expected*

[3] In this article, we will focus on stationary and deterministic policies

reward obtained by following π and starting from s:

$$V^{\pi}(s) \quad = \quad \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right] \tag{1}$$

This function, which seems a priori difficult to tackle with, can, as a matter of fact, be rewritten in the following way (Bellman, 1957):

$$\begin{aligned} V^{\pi}(s) &= R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) \cdot V^{\pi}(s') && (2) \\ &= \Gamma^{\pi} V^{\pi}(s) && (3) \end{aligned}$$

where Γ^{π} is the Bellman Operator:

$$\forall s \in S,\ \Gamma^{\pi} f(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) \cdot f(s') \tag{4}$$

The Bellman Operator satisfies all the prerequisites for Banach's theorem (Banach, 1922) to be applied. Hence it admits a unique fixed point which is, by construction, V^{π}.

This result gives two ways to compute V^{π}:

- either by resolving the system of linear equations (3);
- or by applying the Bellman Operator to a randomly initialized function until convergence.

To find an optimal policy, the Principle of Optimality (Bellman, 1957) can be applied. For every state, the best action to take is the one that maximizes the expected reward, no matter the course of actions that brought us to this state. The Bellman Optimality Operator formalizes this principle:

$$\forall s \in S,\ \Gamma^{\star} V(s) = R(s) + \gamma \max_{a \in A} [\sum_{s' \in S} P(s'|s, a) \cdot V(s')] \tag{5}$$

The Bellman Optimality Operator also verifies Banach's theorem; as a consequence, $V^{\star}$ is its unique and attractive fixed point:

$$\forall s \in S,\ V^{\star}(s) = \Gamma^{\star} V^{\star}(s) \tag{6}$$

To find $V^{\star}$ and any corresponding optimal policy, two strategies are possible, both based on dynamic programming:

- Apply iteratively the Bellman optimality operator (5) to a randomly initialized value function until convergence. Doing a last iteration of the algorithm with an arg max instead of a max gives the optimal policy (Puterman, 2005). This method is known as the value iteration VI (cf. Algorithm 1).
- Starting from a randomly initialized policy, compute its associated value function using the Bellman Operator (4). Then, by applying the Bellman Optimality Operator (5), a more efficient policy is built. These two steps are applied until the policy is stabilized (Howard, 1960). This method is known as the policy iteration PI.

The convergence rate of these two algorithms is difficult to compare: VI iterations are faster but the algorithm converges more slowly. However, PI iterations are much longer because each one requires the computation of a value function. Note that PI has an interesting stopping criterion: its policy is stabilized. In comparison, the VI common stopping criterion only guarantees us to be ϵ-optimal (Puterman, 2005).

For both algorithms, time complexity clearly depends on the size of the state space. Furthermore, memory size complexity is quadratic in the size of the state space due to the necessity to store the transition function. Since the size of the state space increases exponentially as the number of features characterizing the system increases, any realistic model becomes too large to be stored in memory. Furthermore, any iteration of either VI or PI becomes unfeasible because of the large number of states to visit (curse of dimensionality, Bellman, 1961). Many ideas have emerged in the literature in order to deal with this pitfall for VI and PI. We focus here on the concept of abstraction.

2.1.2. Abstraction

In a very large system, many states behave similarly from a mathematical point of view. If we were to aggregate these states together into single more abstract states, we would reduce both the memory consumption and the exponential increase in computational time. Indeed, since aggregated states behave

Algorithm 1: VI: Value Iteration

Data: a MDP $\langle S, A, P, R \rangle$, a discount factor γ

begin

// Initialization

$V \leftarrow R$;

$\Delta \leftarrow 1$;

// Value Iteration

while $\Delta \geq \frac{\epsilon \cdot (1-\gamma)}{2 \cdot \gamma}$ **do**

$V_{old} \leftarrow V$;

foreach $s \in S$ **do**

foreach $a \in A$ **do**

$Q(s,a) \leftarrow R(s) + \gamma \cdot \sum_{s' \in S} P(s' \mid s,a) \cdot V_{old}(s')$;

$V(s) \leftarrow \max_{a \in A} [Q(s,a)]$;

// Stopping criterion reached?

$\Delta \leftarrow \| V - V_{old} \|_{\infty}$;

// Optimal Policy Extraction

foreach $s \in S$ **do**

foreach $a \in A$ **do**

$Q(s,a) \leftarrow R(s) + \gamma \cdot \sum_{s' \in S} P(s' \mid s,a) \cdot V(s')$;

$\pi^{\star}(s) \leftarrow \operatorname{argmax}_{a \in A} [Q(s,a)]$;

Result: a policy $\pi^{\star}$ ϵ-optimal

in the same way from a mathematical point of view, we only need to store the associated abstract states and compute an optimal solution for them. We would then dispose of the optimal solution for each one of the original and concrete states. Several implementations of this idea have been proposed.

In the Hierarchical MDP (HMDP, Guestrin and Gordon, 2002; Hauskrecht et al., 1998; Parr, 1998), the system is decomposed into several elements which

can be again decomposed into sub-elements, each element being organized around a task to resolve. A MDP is then used to resolve each one of these sub tasks. Each task is then abstractly viewed as a macro-action for the parent element. Whenever the system state matches a sub-element state space, the parent considers itself to be in the corresponding abstract state. It then intends to resolve the dedicated task in order to move on to a new abstract state.

In Relational MDPs (RMDP, Boutilier et al., 2001; Wang et al., 2008), the system is modeled by using objects and relations. Each relation is formalized as a predicate and first-order logic is used to handle these relations. Each state is defined by the conjunction of the predicates that are true in that state. As a result, states are now defined by a non constant number of relational atoms. Abstraction can be achieved by replacing the constants in the predicates by variables. These abstract states can also be defined by a conjunction of predicates with variables and can be used in actions definitions. These definitions are based on the STRIPS probability operators (McDermott et al., 1993).

2.1.3. Factored MDP*s*

Both HMDPs and RMDPs require a deep knowledge of the models that may not be achievable for many real-world problems. On the other hand, complex states are often characterized by vectors of features which leads to another kind of abstraction in MDP. The Factored MDPs (FMDPs) evolve around turning the vector of features characterizing the system into a set of variables. Let X_i be a multi-valued variable taking its values over a finite discrete domain D_{X_i}[4].

Definition (Decomposability of state space)**.**

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

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

When S is decomposable, transition probabilities and the reward function

[4]Whenever X_i is instantiated (i.e. set to a given value in D_{X_i}), it will be noted x_i.

can be rewritten as functions of these variables:

$$P(s'|s,a) = P(x'_1,\ldots,x'_n|x_1,\ldots,x_n,a) \quad (7)$$

$$R(s) = R(x_1,\ldots,x_n) \quad (8)$$

Assuming that $\forall i \neq j, X'_i$ and X'_j are conditionally independent to X,

$$P(x'_1,\ldots,x'_n|x_1,\ldots,x_n,a) = \prod_{i=1}^{n} P(x'_i|x_1,\ldots,x_n,a) \quad (9)$$

Conditional independences are exploited to further reduce the number of parameters of each probability distribution using the framework of the dynamic Bayesian Networks (Dean and Kanazawa, 1989; Murphy, 2002):

$$P(x'_1,\ldots,x'_n|x_1,\ldots,x_n,a) = P(x'_i|\textsc{Parents}(x'_i),a) \quad (10)$$

where $\textsc{Parents}(X'_i) \subseteq X$ represents the parents of node X'_i in the directed graph of the Bayesian Network (see Figure 2).

As $|\textsc{Parents}(X'_i)| \leq |X|$, a substantial gain in space complexity is obtained: a large probability distribution over a huge number of states is now represented as a few probability distributions with fewer parameters (see Figure 2). A similar factorization can be applied to the reward function using additive decomposition.

Even if these factorizations are useful, they only concern probability distributions (using conditional independence) or utility (using additive decomposition). Other functions, such as the Value function can not have this kind of decomposition. Indeed, the value function which is composed of an utility function, the reward function, and a joint probability distribution, the transition function, embedded together via the Bellman equation. Furthermore, since $\textsc{Parents}(X_i) \subseteq X$ it may happen that a conditional probability distribution still needs to be stored in a very large array in memory. Finally, the fastidious enumeration of all the states in VI and PI algorithms is not addressed; the complexity in time remains the same.

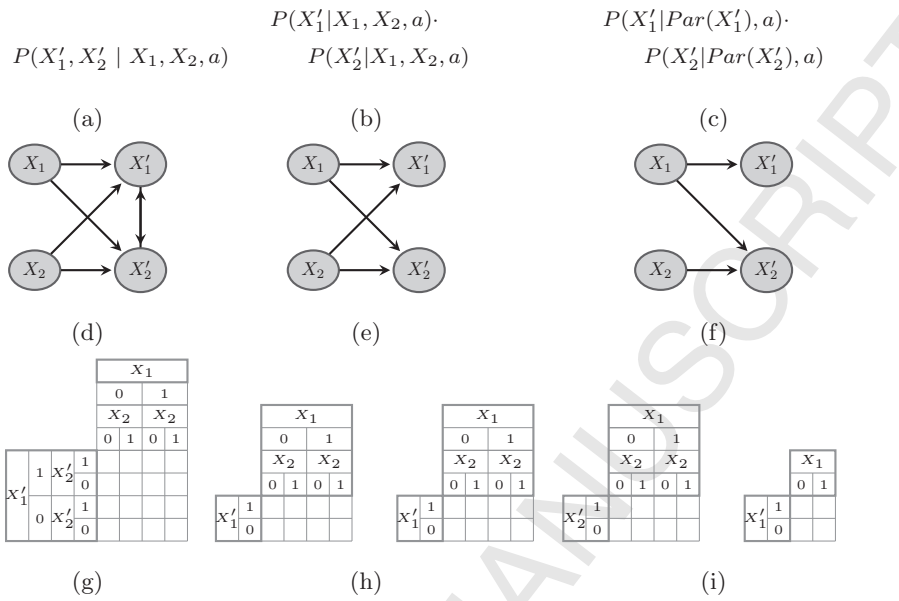


Figure 2: In this example, the same conditional probability distribution over X'_1 and X'_2 given X_1, X_2 and the performed action a is represented: as (a) a conditional joint probability distribution, (b) a product of marginal conditional probability distribution, and (c) exploiting conditional independences.

Figures (d), (e) and (f) show the evolution of the associated DBN as new independences are taken into account.

Finally, figures (g), (h) and (i) show the reductions of the tables used to memorize the whole distribution as new independences are taken into account. Finally, it only requires 12 values to assert the whole distribution instead of 16.

2.1.4. Exploiting contextual independences

A solution, proposed by Boutilier et al. (1995), is to exploit contextual independences. For this purpose, function graphs prove to be efficient data structures.

Let f be a function over the variables $X_1, \ldots, X_n$. 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.,

$$support(f) = \{X_i \,|\, \exists u, v \in D_{X_i} \text{ s.t. } f|_{X_i=u} \neq f|_{X_i=v}\} \tag{11}$$

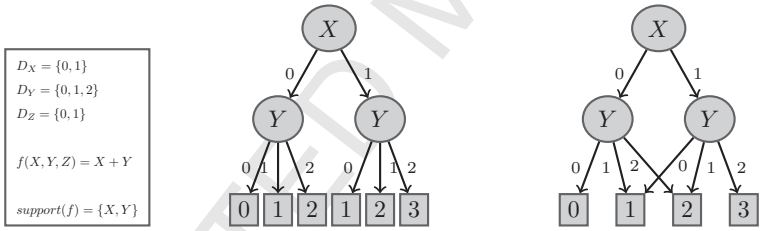
Note that $\forall X_i$, $support(f|_{X_i=b}) \subseteq support(f) \setminus \{X_i\}$: the support of the restric-

tion discards all the variables that become irrelevant and not only the variable X_i.

Definition (Function graph (Recursive Definition))**.** *Let f be a function over $\{X_1, \ldots, X_n\}$.*

A directed acyclic graph (DAG) $G_f(N, A)$ is a function graph of $f \iff$

- *if f is constant then G_f consists of a unique node $N = \{r\}$ s.t. $r.val = f$;*
- *if f is non constant then G_f has a unique node $r \in N$ without a parent. Moreover,*
  - *$\exists X_i \in support(f)$ s.t. $r.var{=}X_i$;*
  - *$\forall u \in D_{r.var}, \exists! n_u \in N$ such that:*
    - *$(r, n_u) \in A$;*
    - *the subgraph from n_u (i.e. $subgraph(n_u)$) is a function graph for $f|_{r.var=u}$.*



(a) Function f (b) DT for f (c) MDD for f

Figure 3: Two different function graphs for the same function.

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 with a variable ($n.var$).

A path from the root to a given node n instantiates every variable visited along that path. As a consequence, upon arriving at node n of G_f, f has been restricted to a function $f|_n$ of which $subgraph(n)$ is the function graph. This restriction is defined by the value assumed by every variable explored on the path from the root to n as shown in Figure 4.

Note that several paths may lead to the same subgraph, characterizing the fact that several restrictions may be equal. For instance, in Figure 3c,

$f|_{X=0,Y=2} = f|_{X=1,Y=1} = 2$. Furthermore, as shown in Figure 3, many different function graphs may describe the same function. However, they all share the same property of compactness: no irrelevant variable can appear on any path of the function graph.

A		0		1	
B		0	1	0	1
C	1	4	5	4	5
	0	7	7	8	8

$$C = 0 \begin{cases} R(0, \_, 0) = 7 \\ R(1, \_, 0) = 8 \end{cases} \qquad C = 1 \begin{cases} R(\_, 0, 1) = 4 \\ R(\_, 1, 1) = 5 \end{cases}$$

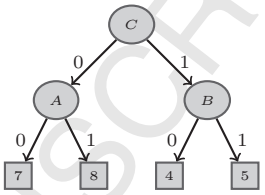


Figure 4: $R(A, B, C)$ has 8 outcomes, one for each state. Yet, the function shows some contextual independences which sum it up to 4 different outcomes, meaning the system has 4 abstract states.

Hence, function graphs are a compact and efficient way to store functions by exploiting the contextual independences. Moreover, dedicated algorithms can be designed to perform mathematical functions (addition, multiplication, maximization) directly on these function graphs. These specific operators exploit the structure of the graphs, and, hence, avoid the costly enumeration of every state. A composed function such as the value function can also be stored as a function graph. By designing VI and PI to exploit the function graphs as data structures and to use their dedicated operators, we have now a way to compute optimal policies without involving exhaustive enumerations of all states and too important memory consumptions.

2.1.5. Using Decision Trees

The first data structure used as a function graph for optimal policy search was the *Decision Tree* (DT, as in Figure 3b). Structured Value Iteration (SVI) and Structured Policy Iteration (SPI) rely on this data structure.

Algorithm 2 shows the layout of the algorithm SVI, the new version of the algorithm VI using DTs. The main difference between the two versions is the disappearance of all the *For* statement over the state space S in the SVI version. Indeed, these statements have been replaced by the introduction of two new

Algorithm 2: SVI: Structured Value Iteration

Data: a FMDP where Transition and Reward functions are factored using DTs

begin

`// Building Initial Value function ...`

$T[V] \leftarrow T[R]$;

$T[V_{old}] \leftarrow \{\}$;

`// Structured Value Iteration`

while $\exists$ *a leaf* $l \in Combine(T[V], T[V_{old}], -)$ *s.t.* $l.val \geq \frac{\epsilon \cdot (1-\gamma)}{2 \cdot \gamma}$ **do**

  $T[V_{old}] \leftarrow T[V]$, $T[V] \leftarrow \{\}$;

  foreach $a \in A$ **do**

    $T[Q_a] \leftarrow \{\}$;

    foreach $X_i \in X$ **do**

      $T[Q_a] \leftarrow Combine(T[V_{old}], T[P_{X_i,a}], \times)$; `// where` $T[P_{X_i,a}]$ `is the tree storing the probability distributions` $P(X'_i \mid \textsc{Parents}(X'_i), a)$.

      $T[Q_a] \leftarrow Project(T[Q_a], X'_i)$; `// where Project suppress` X'_i `from` $T[Q_a]$ `by summing over its values.`

    $T[V] \leftarrow Combine(T[V], T[Q_a], \max)$;

`// Greedy extraction of the` ϵ`-optimal policy` $\pi^\star$

$T[\pi^\star] \leftarrow \{\}$;

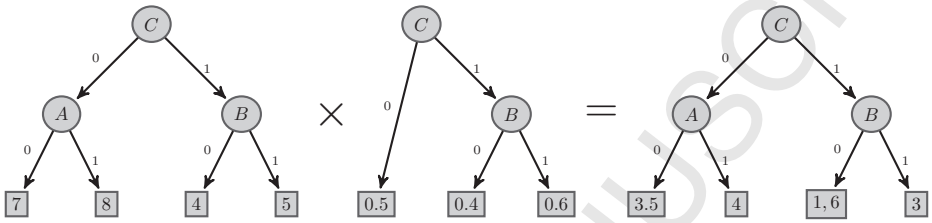
foreach $a \in A$ **do**

  $T[\pi^\star] \leftarrow \acute{e}tendre(T[\pi^\star], T[Q_a], \underset{a \in A}{\text{argmax}})$;

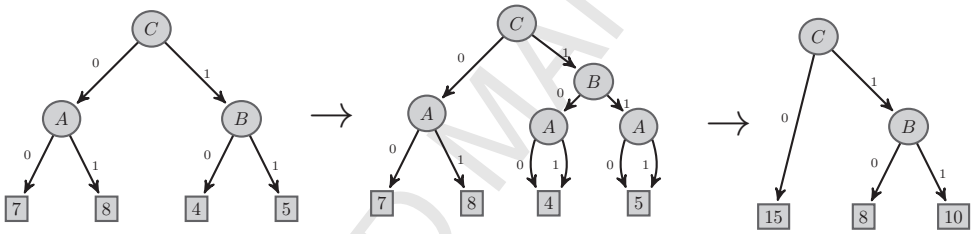
Result: the $T[\pi^\star]$ representing an ϵ-optimal policy for the given FMDP

algorithms: *Combine* and *Project*. In the VI version, the *For* statement was used to compute for each state diverse combinations (addition, subtraction, multiplication, maximization and variable removal) of the functions needed for

the optimal search policy (Transition Function, Reward Function and Value Function). These two algorithms perform these specific combinations while working directly on the DT structures, avoiding then costly enumerations of the whole state space (Boutilier et al., 1999) [5]. Figure 5 shows intuitively how these two algorithms work.



(a) *Combine* algorithm



(b) *Project* algorithm (Suppression of variable A by summing over its values)

Figure 5: Illustrative cases of how *Combine* and *Project* algorithms work.

SVI and SPI were able to overcome, in some measure, the difficulties raised by the exponential increase in the state space. However, as shown in Figures 3b, DTs may contain several isomorphic subgraphs. Indeed, due to the tree structure, these subgraphs can not be merged. Yet, these duplications unnecessarily increase the size of the graphs.

2.1.6. Using algebraic decision diagrams

Hoey et al. (1999) proposed to use *Algebraic Decision Diagrams* (ADD, Bahar

[5] While *Combine* interests us particularly in this article and will be seen in detail in Section 3 (at least its version for ADDs, see below), we will not go through the details of how *Project* works. See (Boutilier et al., 1999) and Hoey et al. (1999) for the details

et al., 1997) as function graphs (as in Figure 3c). ADDs are a generalization of *Binary Decision Diagrams* (BDDs, Bryant, 1986) that represent real functions of boolean variables ($f : \mathbb{B}^n \to \mathbb{R}$). Two important characteristics of these data structures are that they are reduced and ordered.

Definition (Reduced function graph)**.**

$$G_f \text{ is reduced} \iff \forall \text{ nodes } n \neq n',\ f|_n \neq f|_{n'}$$

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 support*(f)*, s.t.* $\forall$ *nodes* n_1, n_2 *non terminal of* G_f*,*

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

When a function graph is ordered, the algorithm to reduce it is polynomial Bryant (1986).

In the same article, Bryant (1986) also gives a version of the algorithm *Combine* dedicated to the BDDs that also works on ADDs. By exploiting the advantages of ADDs over DTs, in terms of compactness, and their specific *Combine* algorithm, Hoey et al. (1999) elaborates another version of VI algorithm, SPUDD, which is still the state-of-the-art exact algorithm for planning with FMDPs.

2.1.7. Improving the data structure

SPUDD is a very efficient algorithm that allows solving very large FMDPs. However, it is confronted by two major drawbacks which limit its use.

First, 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 (see for instance Figure 1). Indeed, to code a variable with n values, one needs $\lceil log_2 n \rceil$ binary variables. These $\lceil log_2 n \rceil$ variables will generate $2^{\lceil log_2 n \rceil}$ possible states, which means that $2^{\lceil log_2 n \rceil} - n$ states are artificially created and have no real

existence. Thanks to the abstraction, these non-existent states will be regrouped in abstract states. However, due to the dedicated operators, an optimal policy will be searched for these irrelevant abstract 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 3c where Y is ternary). The artificial increase in variables is then avoided, reducing therefore any function graph compactness.

The second drawback of ADDs (and of any reduced and ordered function graphs, i.e. MDDs) is that their compactness strongly depends on their inherent variable order. Yet, their specific algorithm *Combine* imposes that they all share the same order. As a consequence, this common order may not be the optimal one for each one of them or for the resulting ADDs. Given that the complexity of the algorithm *Combine* with two ADDs D_1 and D_2 as entries is in $\mathcal{O}(|D_1| \cdot |D_2|)$, this common order tends to arbitrarily increase the computational complexity.

In section 3, we will describe a new algorithm *Combine* on MDDs that will not impose such a common order.

2.2. Planning while learning with factored models

Another subject of concern addressed in this article is the difficulty to dispose of a complete FMDP prior to the optimal policy search. The representation of the world (Reward function and Transition probabilities) may not be known because of a lack of data (the state space has not been explored yet) or because of the nature of the data (typically for on-line processes such as stock market prediction). This remark leads to the implementation of FMDP learning algorithms.

The reinforcement learning framework proposes to learn the FMDP of a situation by trial-and-error (Dyna, Dyna-Q, Sutton, 1990). As the agent experiences the real world problem (acting), an on-line algorithm builds up a representation of transition and reward functions (learning) by taking into account each new transition observation ξ (learning) made, then a planning algorithm computes an

efficient policy based on the created model (planning). In Degris et al. (2006b), the general architecture SDYNA integrates planning, acting and learning using factored representations as described in Algorithm 3.

Algorithm 3: SDYNA global architecture

1 **foreach** *time step* t **do**
2 $s \leftarrow$ current state;
3 $a \leftarrow \pi_t(s)$; // π_t is the current strategy
4 Perform a, observe s' and r and define $\xi = (s, a, s', r)$;
5 Incremental "factored" learning from observation ξ;
6 "Factored" planning new π_{t+1};

In order to implement a SDYNA architecture, one has (i) to choose a structured representation of the problem (DT, ADDs, etc.); (ii) to provide an optimal policy search algorithm (line 6); and (iii) to provide an incremental learning algorithm for the data structure (line 5). The following subsections cover the learning algorithms used to establish factored models.

2.2.1. Incremental learning of decision trees

There are many algorithms to learn a tree from a set of observations: CART (Breiman et al., 1984), C4.5 (Quinlan, 1993), etc. However, many of these approaches require having the complete set prior to the learning. They are not able to modify dynamically the tree as new observations arrive. A windowed approach could be implemented but it would require rebuilding the tree from scratch for each new observation. ITI (Utgoff et al., 1997) is an algorithm that fulfills this need of on-line adaptations to new observations.

In ITI, each node N of the learned DT contains a set of observations Ω_N that are compatible with the instantiation in N. For instance, the sets of observations installed in the leaves of a tree form a partition of the set of all the observations Ω. Adding a new observation ξ means updating any set compatible with ξ and then modifying the structure of the tree where needed. The structure depends itself on these sets of observations: an internal node N contains the "best" (not yet instantiated) variable that separates the set of observations Ω_N. To select

a variable, ITI uses the Information Gain Ratio as a criterion and compares the different distributions that would be created by installing each free variable at the node. We refer the readers to Utgoff et al. (1997) for a much more complete presentation of this algorithm.

2.2.2. Planning while learning with decision trees

SPITI is an instantiation of the SDYNA architecture where DTs are used for the model factorization. Hence, for each $P(X'_i|A,\text{Parent}(X'_i)), X'_i \in \mathbf{X}'$ and for the reward function $R(\mathbf{X})$, a DT has to be incrementally learned. Then, the SPI algorithm (Boutilier et al., 1999) is used on the learned DTs for the planning step.

In Degris et al. (2006a), SPITI is implemented using a version of ITI where the variable selection criterion is the χ^2 test instead of the Information Gain Ratio. Degris et al.(2006a) gives two major reasons to do so. First, White and Liu (1994) shows that the χ^2 criterion does not have any bias toward multi-valued values. Furthermore, the χ^2 test allows SPITI to do pre-pruning: a leaf will not be refined if no variable is a match according to the χ^2 test.

2.2.3. Learning MDD*s*

Several articles address the problem of learning Multi-Valued Decision Diagrams. Oliver(1993) proposes to first build a tree using one of the state-of-the-art algorithms (CART, C4.5). Then, this tree is ordered in order to facilitate the search for isomorphic subtrees. Eventually, these subtrees are merged using the Minimum Description Length principle as a criterion. Kohavi and Li(1995) directly builds an ordered tree and then reduces it using its own set of rules.

These algorithms do not cope with the issue of on-line learning since the trees are learned from fixed databases. Therefore, adding new observations to the database demands learning a new whole tree. Hence, these approaches can not be good candidates for a SDYNA architecture. Being able to review the tree, and if and only if necessary its reduced version the MDD, without having to go through the whole database would be a great asset that has not been proposed

yet to the best of our knowledge. Section 4 covers our results on that matter.

2.2.4. Towards a SDYNA *framework with* MDD*s*

In this section, we aimed to show that the FMDP framework proposes a good solution to solve large MDPs by abstracting the states. The state-of-the-art framework for reinforcement learning in FMDP is SDYNA whose only instance is SPITI which deals with decision trees. However, MDD is a much more compact function graph. In the following sections, we will then present a planning algorithm using MDDs (section 3), an incremental learning algorithm for MDDs (section 4). These two algorithms will allow us to propose a new instance of SDYNA using MDDs (section 5).

3. Planning with Multi-valued Decision Diagrams

In this section, we introduce SPUMDD, a new planning algorithm like SVI or SPUDD that works on MDDs. The main difference of this new algorithm over its predecessors, apart from the fact that it works on MDDs, is that it uses a new version of the *Combine* algorithm that aims at removing the global order constraint. Indeed, the original *Combine* algorithm that was working on ADDs and that also works on MDDs, while having a complexity depending on the sizes of the two combined Decision Diagrams, forces them to be ordered the same way. However, the size of a Decision Diagram strongly depends on its ordering. The resulting problem is that, with the constraint, the two Decision Diagrams may not be ordered optimally, only optimally for the operation. This results in an increase of the complexity of the *Combine* algorithm that may be removed if we were capable of doing this combination without forcing the two Decision Diagrams to be ordered the same way. This section covers our inquiry over that matter.

Let G_1, G_2 and G be three reduced and ordered function graphs (BDDs, ADDs or MDDs) such that $G = G_1 \odot G_2$[6]. Based on our criticism of the existent

[6] $\odot$ is a commutative operation: addition, multiplication, maximization, etc.

dedicated algorithm on ADDs (in SPUDD), this new *Combine* algorithm should produce 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.

At its core, the original *Combine* algorithm calls *OrderedExplore*, a recursive function which explores the two diagrams. This is the behavior of this *OrderedExplore* function that we have to change in order to remove the global order constraint. First, we briefly present the state-of-the-art *OrderedExplore* function that works on two Decision Diagrams ordered similarly (for further details, please refer to (Bryant, 1986)). Then we will go into the details of how to modify this function to obtain the *Explore* function that works without the constraint.

3.1. The OrderedExplore *function*

Let $\succ$ be the common order imposed on G_1, G_2 and G for the operation. The *OrderedExplore* function relies on recursive and simultaneous depth-first explorations of both G_1 and G_2. Each recursive call is made on a pair of nodes $n_1 \in G_1$ and $n_2 \in G_2$. The initial call is made by the *Combine* algorithm on the roots of both graphs; the recursive call on nodes n_1 and n_2 determines how the nodes n_1 and n_2 will be visited:

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 the exploration function only visits this node;
3. if both nodes are non-terminal then
   a. if $n_1.var = n_2.var$, the exploration function visits simultaneously both nodes;
   b. if $n_1.var \succ n_2.var$ (resp. $n_2.var \succ n_1.var$), then the exploration function visits only on n_1 (resp. n_2).

The visit of a node n consists in calling the function again on each one of its children. Since each child is bound to a value that may assume $n.var$, the variable is then instantiated on each call. The called node from the other diagram remains unchanged, unless it is simultaneously explored. In that case,

it is on its child selected by the current value to which $n.var$ is instantiated that the function is called.

At the end of each recursive call, when the exploration of every child is over (for an internal node) or when a value has been computed (terminal node), a new node n_G is inserted in G. If a value was computed, n_G will then be a terminal node and have the computed value attached to it. Otherwise, n_G will have $n.var$ attached to it. The children of n_G are the resulting nodes from the explorations on n children. Of course, redundancy checks are made before inserting n_G so that G is also reduced.

Proceeding this way ensures that every variable will be in the correct order in G. Due to the assumption of a common order, the algorithm is quite simple. When both G_1 and G_2 are trees, each pair of nodes is visited only once. Its complexity is then in $\mathcal{O}(|G_1|.|G_2|)$. For BDDs, ADDs and MDDs, a pruning mechanism (a cache) is needed in order to keep this complexity (see below for more details).

Removing the common order constraint implies that different orders on the variables will coexist: $\succ_1$ for G_1, $\succ_2$ for G_2 and $\succ_G$ for $G = G_1 \odot G_2$.

3.2. The problem of the retrograde variables

With the objective of still performing a depth-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 nodes considered at the beginning of a recursive call. It may now happen that $n_1.var \succ_1 n_2.var$ and $n_2.var \succ_2 n_1.var$. Figure 6 illustrates that case.

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 $X_p \succ_2 X_r$.

X_p is then retrograde in $\succ_1$ w.r.t $\succ_2$ because of X_r (at least). For instance, on Figure 6, we have $A \succ_1 B \succ_1 C$ and $A \succ_2 C \succ_2 B$. As a consequence, B is retrograde in $\succ_2$ w.r.t. $\succ_1$ because of C.

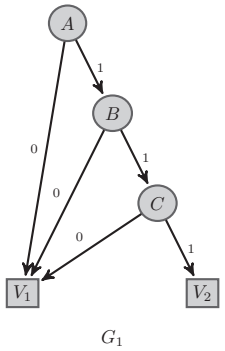


Figure 6: Illustrative case where $\exists n_1$ and n_2 s.t. $n_1.var \succ_1 n_2.var$ and $n_2.var \succ_2 n_1.var$. Here, each time the currently explored nodes are associated to B on one diagram and to C on the other, we are in this new situation.

We note the set of retrograde variables: $\Re_{1,2} = \{X_i \in X, X_i \text{ retrograde in } \succ_2 \text{ w.r.t. } \succ_1\}$ and $D_{\Re_{1,2}}$ the domain of that set. The size of that domain is $|D_{\Re_{1,2}}| = \prod_{X \in \Re_{1,2}} |X|$. As we will see below, the size of both $D_{\Re_{G,1}}$ and $D_{\Re_{G,2}}$ should be minimal.

To address this new case, the function *Explore* has to decide which variable is explored first. This has several consequences that we are going to see now in detail.

3.3. Exploration and construction

A direct consequence of choosing a variable X_r over a variable X_p for the exploration is that X_r will have to precede X_p in the order of the resulting diagram G. Indeed, the recursive call of *Explore* on any node n_r s.t. $n_r.var = X_r$ will end after the recursive call of *Explore* on any node n_p s.t. $n_p.var = X_p$. As a consequence, the resulting node n_r^G will be parent to any resulting n_p^G created during the visits of nodes n_p. Hence, the necessity for X_r to precede X_p in $\succ_G$.

Another difficulty raised by this choice is the situation where a node bound to X_p has among its descendant a node bound to X_r. Indeed, since we have $X_r \succ_1 X_p$ and more importantly $X_p \succ_2 X_r$, we may have in G_2 a node n_r^2 bound to X_r that is among of the descendant of a node n_p^2 bound to X_p. The node

n_r^2 cannot be explored like any other node since it would violate the constraint $X_r \succ_G X_p$ (its recursive call taking place during the recursive call on n_p^2).

To enforce $X_r \succ_G X_p$, a recursive call instantiating variable X_r must have begun before the exploration on n_p^2 begins. Then, whenever during the explorations in G_2 a node bound to X_r is encountered, as X_r has already been instantiated, the algorithm immediately jumps onto the child node determined by the current value of X_r.

A direct consequence is that we may have to possibly anticipate an exploration on X_r whenever an exploration on X_p is required in G_2 (in the case where no exploration has begun on X_r while an exploration on X_p begins). This has a consequence in terms 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. Algorithm 4 presents the core function for the exploration and the different cases it has to deal with.

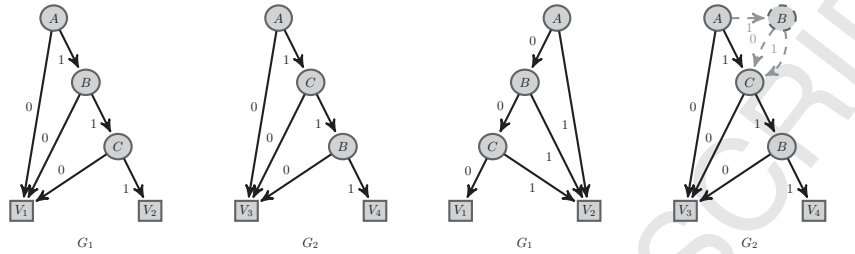
Let (n_1, n_2) be the visited nodes at a given step of our algorithm. If $subgraph(n_2)$ contains X_r[7], the algorithm possibly have to anticipate an exploration on X_r. On the contrary, if $subgraph(n_2)$ does not contain X_r, the algorithm can normally perform the exploration on n_2.

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 explored in G_1. It is then normally explored regardless of its presence in G_2.

Assume that exploration on X_p is required and that X_r possibly has to be anticipated. Two cases can occur: (i) either X_r has been met in the current explored path on G_1; (ii) or X_r has not been met current explored path on G_1. In the first case, X_r has already been instantiated and then no anticipated

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

exploration on X_r is needed. In the second case, the anticipated exploration of X_r is needed before exploring X_p on G_2.



(a) No anticipated exploration is needed. (b) Anticipated exploration is needed.

Figure 7: The two cases. They occur when exploration explore the children of the root nodes for $A = 1$. In case (a), for G_1, the child node is bound to B, and, for G_2, the child node is bound to C. Exploration begins then to explore B, then C. Hence, when C is explored, B is already being explored. In case (b), child for G_1 is a terminal node. Then C is to be explored. The problem is that then no exploration on B has begun. The solution is to insert in G_1 a node bound to B.

For the sake of simplicity of the presentation, we propose to arbitrarily build $\succ_G$ so that $\Re_{G,1} = \emptyset$. The consequence of this choice is that any variable X_r retrograde in $\succ_2$ w.r.t. $\succ_1$ because of any variables X_p will be explored first. The proposed order $\succ_G$ will then have the following properties:

- $\succ_G$ extends $\succ_1$;
- $\succ_G$ extends $\succ_{2\setminus\Re_{1,2}}$.

These two properties are sufficient to build $\succ_G$ from $\succ_1$ and $\succ_2$. By definition, $\succ_G$ is built so that $\Re_{G,1} = \emptyset$. It follows that the set of retrograde variables that can be met is $\Re_{G,2}$.

3.4. Pruning and complexity

In the algorithm *OrderedExplore*, several explorations of a same pair of subgraphs always give to the same resulting structure. The different paths that lead to those subgraphs 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 Algorithm 4, pruning is more difficult: consecutive visits of a same pair of nodes do not necessarily lead to the same result. Suppose that for the current nodes (n_1, n_2), $n_2.var = X_p$ and $X_r \in subgraph(n_2)$. With or without anticipated exploration, X_r will be instantiated to a certain value before exploration starts on n_2. Then, as evoked just above, when any node bound to X_r is encountered in G_2, exploration automatically jumps onto the child selected by the current value of X_r. A direct consequence is that only a part of $subgraph(n_2)$ is then explored. The algorithm will have to explore again $subgraph(n_2)$ for every value that X_r can assume. This has to be done from n_2 in order to build the resulting nodes in the right order (i.e. $succ_G$).

Algorithm 4: Operation between MDDs without common order.

```
1  Procedure Explore(n1,n2,E)
       Input: n1 ∈ G1,n2 ∈ G2, E instantiations of already explored variables
       Output: n1 ⊙ n2
2      if n1.isTerminal and n2.isTerminal then
3          return Terminal(n1.val ⊙ n2.val)
4      if not n2.isTerminal and ∃nr ∈ n2.Descendants, nr.var ∈ ℜ1,2 then
5          ∀m ∈ D_{nr.var}, nm = Explore(n1, n2, E ∪ {nr.var = m})
6          return NonTerminal(nr.var, children = {nm}_{m∈D_{nr.var}})
7      if n1.var ≻G n2.var or n2.isTerminal then
8          ∀m ∈ D_{n1.var}, nm = Explore(n1.child(m), n2, E ∪ {n1.var = m})
9          return NonTerminal(n1.var, children = {nm}_{m∈D_{n1.var}})
10     if n2.var ≻G n1.var or n1.isTerminal then
11         if ∃m, n2.var = m ∈ E then
12             return Explore(n1, n2.child(m), E)
13         else
14             ∀m ∈ D_{n2.var}, nm = Explore(n1, n2.child(m), E ∪ {n2.var = m})
15             return NonTerminal(n2.var, children = {nm}_{m∈D_{n2.var}})
16     if n1.var = n2.var then
17         ∀m ∈ D_{n1.var}, nm = Explore(n1.child(m), n2.child(m), E ∪ {n1.var = m})
18         return NonTerminal(n1.var, children = {nm}_{m∈D_{n1.var}})
```

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 a computed subgraph for $n_1 \odot n_2$ is identified not only by the nodes n_1 and n_2 but also by the values of the explored retrograde variables.

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

But this worst-case complexity has to be pondered. Firstly our algorithm only re-explores subgraphs of G_2 when needed. The worst-case complexity is determined as if all the re-explorations concerned the whole graph. Unfortunately a more accurate upper bound is difficult to obtain; it would demand a topological analysis of the graphs. Secondly G_1 and G_2 have now their (optimal) own order. Then the size of G_1 and G_2 can be smaller in this complexity than in the case of the algorithm with common order. Thirdly $\succ_G$ can be compatible with either $\succ_1$ or $\succ_2$. As a consequence, we have a choice in the set of retrograde variable we have to deal with: either it can be $\Re_{1,2}$ or $\Re_{2,1}$. And as $|D_{\Re_{1,2}}| \neq |D_{\Re_{2,1}}|$, another trade-off is to be found here.

This discussion is confirmed in the experiments described below.

3.5. Experimentations and Observations

We have implemented SPUMDD using the C++ library aGrUM, developed at LIP6 laboratory. Since the standard implementation of SPUDD uses a 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, the 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, the value function decison diagram is used to compute various decision diagrams that are themselves aggregated back into a new value function 5. Therefore, the size of value function is a good indicator of the efficiency of

	SPUMDD			SPUMDD compared to SPUDD		
	State Space	Internal Nodes	Time (s)	State Space	Internal Nodes	Time
Coffee Robot	64	21	0.7	100.0%	100.0%	100.0%
Factory	55 296	377	103	42.2%	46.3%	13.3%
Factory 0	221 184	384	118	42.2%	44.4%	11.3%
Factory 1	884 736	736	213	42.2%	35.2%	6.7%
Factory 2	1 769 472	736	213	42.2%	35.2%	6.7%
Factory 3	10 616 832	814	903	31.6%	42.6%	8.4%
Maze 5x6	30	6	0.86	46.9%	17.7%	34.3%
Maze 8x8	64	9	1.05	100.0%	15.5%	31.6%

Table 1: Results using SPUDD and SPUMDD. The last columns illustrate the improvements in space and time using SPUMDD.

the representations.

In both algorithms, reordering was performed regularly to ensure that the data structures were of minimal size. Of course, for SPUDD, it implies that the minimization occurs simultaneously on every diagram so that the global size is minimized. The heuristic used is the SIFTING algorithm[8] (Rudell, 1993), with proper modifications for multivalued variables in SPUMDD.

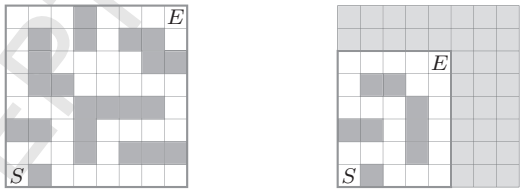


Figure 8: Maze examples. Walls are in dark gray. Impossible s generated by binarization in light gray.

Table 1 shows results of value iteration using SPUDD and SPUMDD on various MDPs. State space size gives the total number of states (including the

[8] St-Aubin et al. (2000) argues that this algorithm is the most efficient one in a planning framework.

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 seconds) 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*. *Coffee Robot* is a problem in which an agent has to seek coffee for its owner while avoiding to get wet by the rain. *Coffee Robot* is a small binary problem. In the *Factory* issue, the goal is to machine and assemble two pieces together in the correct order. *Factory* is mostly a binary problem too (only a few ternary variables), its state space size is larger (5000 states).

The interest in the *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 small purely binary problems ADDs are sufficient.

In *factory*, the interest resides in the mix 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 show clearly that advantage can be taken of by SPUMDD.

Note 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 MDDs can eliminate non relevant variables as ADDs do.

To examine the behavior on problems with multi-valued variables, two mazes have been created. The maze (Figure 8) has 30 cases, 8 of them being blocked. It only requires 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 axis. These variables generate a grid of 64 states where 34 are impossible. The second maze is an 8 by 8 maze, and thus generates no impossible states on translation into a binary problem.

Here again, SPUMDD shows itself better than SPUDD, gaining both on time

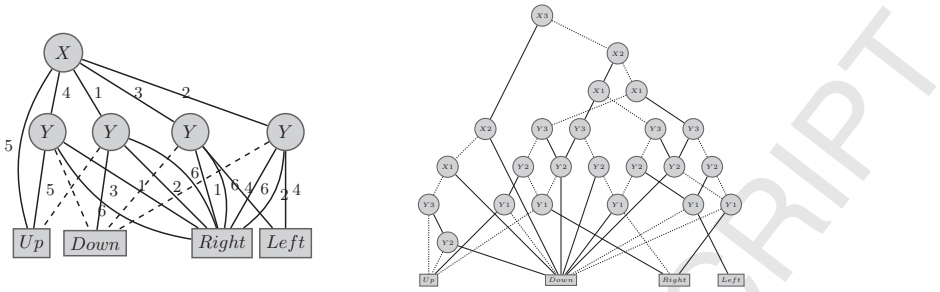


Figure 9: Maze Optimal Policy with SPUMDD (dashed line stands for default arc) and with SPUDD (dashed line from node X stands for $\overline{x}$).

and size representation. Figure 9 shows the simplification of the policy obtained for the first maze with SPUMDD compared to SPUDD.

3.6. Planning with MDD*s*

In this section, we proposed a new algorithm for operations between MDDs. This new algorithm allows us to define SPUMDD, a new planning algorithm using MDDs as compact representation for the different functions. SPUMDD proved to be efficient both in size representation and in time. The next section deals with the second step of our program towards a new SDYNA instance: incremental (or on-line) learning of MDDs.

4. On-line Learning of Multi-valued Decision Diagrams

In this section we describe IMDDI, a novel algorithm for the incremental learning of Multi-valued Decision Diagrams (Magnan and Wuillemin, 2015). Without any loss of generality, this presentation will focus on the estimation of the probability distribution of a multi-valued variable Y according to a set of multi-valued variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ (see Figure 10). Indeed, learning the transition model of a factored MDP consists in learning several conditional probability distributions (Boutilier et al., 1999). Moreover, learning other functions such as the reward function is done using very similar algorithms.

There are two major differences between a DT and an MDD: first, an MDD is structured by a global order on the variables. Variables must appear on each

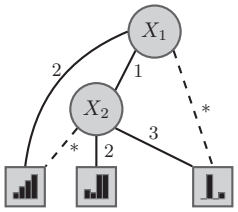


Figure 10: An MDD representing a probability distribution $P(Y|X_1, X_2)$. Each leaf contains a probability distribution $P(Y)$ over the domain of Y. This MDD states that $P(Y|X_1 = 2) = P(Y|X_1 = 1, X_2 \notin \{2, 3\})$ and that $P(Y|X_1 \notin \{1, 2\}) = P(Y|X_1 = 1, X_2 = 3)$. Those equalities represent Context-Specific Independence in $P(Y|X_1, X_2)$ (Boutilier et al., 1996).

branch of the MDD w.r.t this global order. This order has a large impact on the compactness of the MDD. We call a DT with this constraint an Ordered Decision Tree (ODT).

Second, an MDD merges sub-trees together in order to be reduced. The complexity of reducing an ODT T into an MDD is in $O(|T| \cdot \log |T|)$ where $|T|$ is the number of nodes in T (Bryant, 1986). Figure 11 depicts the transformation from a DT into an MDD via an ODT. A first incremental algorithm to learn MDD (named ITI+DD later) could be i) to simply use ITI to learn the DT, ii) then to choose an order and to convert the learned DT into an ODT and finally iii) to build a MDD from this ODT at each step as it has been proposed for non incremental learning of MDDs by Oliver (1993). However the search for an optimal global order is NP-hard. Moreover it is possible to maintain an estimation of an efficient global order. This is a key point in our algorithm IMDDI: we propose a modified version of ITI that handles an ordered decision tree instead of a tree. Then, whenever needed, the MDD will be built from this ODT.

4.1. Incremental Induction of an Ordered Tree

IMDDI is based on a version of ITI but the main features have been revised and are described in the following subsections: how to select the variable to be installed at any node; how to incrementally integrate a new observation ξ and how to update the structure w.r.t. this new observation.

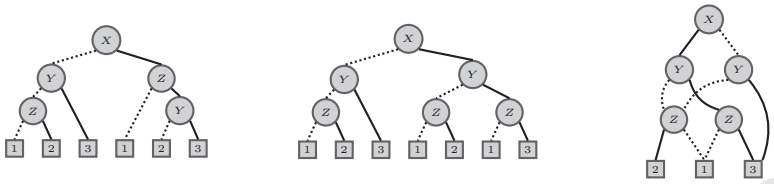


Figure 11: A function represented as (a) a DT, (b) an ODT ($X \succ Y \succ Z$) and (c) an MDD (with the same order).

Algorithm 5: Incremental MDD Induction (IMDDI) for $P(Y|X_1, \ldots, X_n)$

Data: a data stream of observations: $\xi = (X_1, \ldots, X_n, Y)$

```
1 foreach ξ = next observation do
2     AddObs(ξ);                          // see Algorithm 6
3     UpdateODT();                        // see Algorithm 7
4     if change is needed then
5         Reduce();                       // see Algorithm 8
```

Algorithm 5 presents the complete IMDDI algorithm. The three steps of this algorithm are presented in detail below.

4.1.1. Variable Selection

Like every DT learning algorithm, IMDDI needs a selection criterion to decide which variable has to be installed on a given node. As selection criteria, both G-statistic (Mingers, 1989) and χ^2 statistic are interesting as they have no bias toward multi-valued variable (White and Liu, 1994). G and χ^2 tests are close when the size of the sample is big enough but Dunning (1993) argues that G-statistic is superior to the χ^2 statistic for dealing with rare events.

Let N be the node on which we want to either install a variable (if N is a leaf) or ensure that the current variable is the most pertinent. Let Ω_N be the set of associated observations on which we rely to perform our selection (see Algorithm 6 below to understand how these databases are extracted). Let $\mathbf{V}_N$ be the set of variables that could be installed at node N. For each variable $X_i \in \mathbf{V}_N$, N keeps a contingency table giving the sample size $n_{x_i,y}$ for each combination of

X_i and Y values. The G-statistic is then computed in the following way:

$$G(X_i) = 2 \cdot \sum_{x_i \in Dom X_i} \sum_{y \in Dom Y} n_{x_i,y} \ln \frac{n_{x_i,y} \cdot |\Omega_N|}{n_{.,y} n_{x_i,.}} \tag{12}$$

where $n_{.,y} = \sum_{x_i} n_{x_i,y}$ and $n_{x_i,.} = \sum_y n_{x_i,y}$.

To decide among every X_i which one should be installed, IMDDI compares the p-values associated to the computed G-statistics: variables with a high number of values tend to have a high G-statistic whereas variables with a low number of G-statistic has a low score. The use of p-values avoids this bias towards multi-valued variables since the p-values "normalize" every G-statistic by integrating degrees of freedom. Furthermore, like the χ^2, the G-statistic has an interesting feature: it can also be used as a pre-pruning criterion in order to prevent the tree from growing unreasonably. To sum up, the variable with the highest p-value will be selected. If this p-value is higher than a fixed threshold, we install that variable at the node. If it is lower than the threshold and node N is not a leaf, N is turned into a leaf.

4.1.2. Adding a new observation

An observation ξ is an instantiation of all the variables $\langle X_1, \cdots, X_n, Y \rangle$. By construction, there exists a unique path from the root to a leaf of the ODT that represents a partial instantiation of $\langle X_1, \cdots, X_n \rangle$ compatible with ξ. Adding ξ to the ODT consists in updating the database Ω_N and the G-statistics of every node N of that path. With X_N the variable installed on an internal node N, $p_G^N(X_i)$ the p-value for a variable $X_i \in \mathbf{V}_N$ at the node N, $c_N(v)$ the child of node N for the value v of X_N and finally $\xi(A)$ the value for the variable A in ξ, Algorithm 6 describes this update of the internal structure of the ODT.

4.1.3. Updating the ODT

Once the statistics have been updated, a last step consists in revising the tree topology. Due to the insertion of the new observation, a revision of the variables previously installed at every node may be necessary. However, this revision has to take into account the global order. To update the ODT, IMDDI

Algorithm 6: AddObs (addition of an observation ξ)

Data: the observation $\xi = \{x_1, \ldots, x_n, y\}$ and the ODT T

```
Node N ← root of T;
repeat
    Add ξ to Ω_N;
    foreach variable X_i ∈ V_N do
        Update p_G^N(X_i);
    if N is not a leaf then
        Node N ← c_N(ξ(X_i));
until N is a leaf;
```

must (i) find a (as good as possible) global order, (ii) ensure that this order is respected on every branch and (iii) ensure that the best possible variable is installed at every node. A last requirement is that, for any observation that will not change the structure, this operation should be as simple as possible. As an incremental algorithm, IMDDI infers a relevant global order while keeping the possibility to revise it. To fulfill these requirements, the strategy we propose is to check variable by variable the relevance of their position in the current global order.

To decide which variable should be the next one in the global order, each remaining variable has to be scored using the G-statistics maintained at any node but later aggregated from specific nodes within a boundary $\mathcal{B}$. This boundary is the set of nodes where a variable check-up must be performed w.r.t. the updated global order. The boundary is initialized as the root node (singleton) and will contain the leaves of the ODT at the end of this structure revision. IMDDI computes an aggregated score by summing up the p-values computed on the nodes $N \in \mathcal{B}$ weighted by the proportion of observations at these nodes ($|\Omega_N|$) compared to the total number of observation added to the tree ($|\Omega|$). The variable with the highest score is then chosen and becomes the next variable in the updated global order.

Then, for every node of the boundary, this chosen variable will be installed if its associated p-value for that node is above a fixed threshold. This installation

is done the same way it is done in ITI (see Utgoff et al. (1997) for further details). If the variable is effectively installed, the node is removed from the boundary, replaced by all its children. Once this step is over, nodes in the boundary will not be authorized to choose to install this variable later on.

Algorithm 7: UpdateODT (updating the structure)

Data: an ODT T after adding ξ (with Algorithm 6)

$\mathcal{B} = \{$ root R of $T\}$; `// boundary`

$\mathcal{F} = \mathbf{X}$; `// Set of variables`

repeat

  foreach *variable* $X_i \in \mathcal{F}$ **do**

    $p_G(X_i) = \sum_{N \in \mathcal{B}} \frac{|\Omega_N|}{|\Omega|} \cdot p_G^N(X_i)$;

  $V \leftarrow \arg\min_{X_i \in \mathcal{F}} p_G(X_i)$;

  $\mathcal{B}' \leftarrow \mathcal{B}$;

  foreach $N \in \mathcal{B}$ **do**

    if $p_G^N(V) \geq \tau_1$ **then**

      Install V in node N;

      $\mathcal{B}' \leftarrow \mathcal{B}' \setminus \{N\} \cup \bigcup_{v \in Dom(V)} c_N(v)$

  $\mathcal{B} \leftarrow \mathcal{B}'$;

  $\mathcal{F} \leftarrow \mathcal{F} \setminus \{V\}$;

until $\mathcal{F} = \emptyset$ *or no variable in* $\mathcal{F}$ *can be installed in* $\mathcal{B}$;

The stopping criterion for Algorithm 7 has to take into account two cases: either all variables of $\mathbf{X}$ have been added to the updated global order or no variables can be installed at any node of the current boundary (i.e. all the p-values are below the threshold). When it stops, the boundary contains all the leaves of the updated ODT. Any node which is internal at this moment is turned into a leaf, its subtree being removed.

If the new observation does not change the structure of the ODT, the only computations performed by Algorithm 7 are weighted sums on each boundary from the root to the leaves of the tree.

4.2. From the ODT to the MDD

Once the ODT is generated, the next step is to reduce it into a MDD. Merging the isomorphic subtrees is done by the bottom-up polynomial (in time) algorithm 8. This algorithm begins by merging all the leaves with similar probability distributions. Then, for every variable X_i, going backward in the global order, two nodes N and N' bound to X_i are merged if they have the same children. Furthermore, if a node has only one child then it is redundant and is then replaced by arcs outgoing from its parents to its unique child.

Algorithm 8: Reduce (merge isomorphic subtrees)

Data: an ODT T

repeat
  $(U^*, V^*) = \arg\min_{(U,V) \text{ leaves}}(\max(p_U^G, p_V^G))$;
  if $\max(p_G^{U^*}, p_G^{V^*}) \leq \tau_2$ **then**
    Merge U^* and V^*;
until $\nexists$ *two leaves that can merge*;
foreach $X_i \in \mathbf{X}$ *backward w.r.t the order* **do**
  foreach N_{X_i}, N'_{X_i} *with* X_i *as installed variable* **do**
    if $\forall x_i \in Dom(X_i), c_{N_{X_i}}(x_i) = c_{N'_{X_i}}(x_i)$ **then**
      N_{X_i} and N'_{X_i} are merged ;
  foreach N_{X_i} *with* X_i *as installed variable* **do**
    if $\forall x_i^k, x_i^l \in Dom(X_i)$, $c_{N_{X_i}}(x_i^k) = c_{N_{X_i}}(x_i^l)$ **then**
      Replace N_{X_i} by arcs outgoing from the parents to the unique child.

The first stage which consists in merging similar leaves together has to be addressed more specifically. If the leaves of the MDD were discrete values, merging these leaves would be very simple. However, in our framework, each leaf is a probability distribution over the variable Y. As a consequence, we need a test to decide whether or not two probability distributions are similar.

Let U and V be the two leaves we want to merge, let $n_{U,y}$ be the sample size for the value $y \in Dom(Y)$ and $N_U = |\Omega_U|$ be the total number of observations on the leaf U. Merging U and V would produce a new node W. It follows that $\forall y \in Dom(Y)$, $n_{W,y} = n_{U,y} + n_{V,y}$ and $N_W = N_U + N_V$. To determine whether or not we should merge U and V into W, we compute for both leaves

a G-statistic:

$$\forall L \in \{U, V\}, G_L = 2 \cdot \sum_y n_{L,y} \ln \frac{n_{L,y}}{e_{L,y}}$$

where $e_{L,y} = n_{W,y} \frac{N_L}{N_W}$. Note that we have to scale down the quantity $n_{W,y}$ in order to compare it to the quantity $n_{L,y}$. We propose a greedy algorithm on the leaves of the ODT (Algorithm 8): a pair of p-values (p_G^U, p_G^V) is computed for every possible pair of leaves (U, V). Then the best candidate for merging is $(U^*, V^*) = \arg\min_{(U,V)} \max\left(p_G^U, p_G^V\right)$. This criterion selects the pair with the lowest high dissimilarity in the probability distributions. If both $p_G^{U^*}$ an $p_G^{V^*}$ are smaller than the threshold then the nodes are merged and the process is repeated. The stopping criterion is the absence of merging during an iteration.

4.3. Some properties about complexity of IMDDI

As a composite incremental learning algorithm, it is difficult to assess a global complexity for IMDDI. However, some properties can be stated about the behavior of parts of this algorithm. This section lists some of them. Let $\mathbf{X}$ be the set of variables, Y the variable of interest. IMDDI tries to learn $P(Y|\mathbf{X})$ as an MDD using an ODT T.

Property 1 (Adding an observation – algorithm 2)**.** *The complexity of adding a new observation ξ to the* ODT *is in $\mathcal{O}(|\mathbf{X}|^2)$.*

Proof. To add a sample to the ODT, the algorithm performs a depth-first search until it reaches a leaf. The height of the tree is in $\mathcal{O}(|\mathbf{X}|)$ since each variable appears at most once on every path from the root to the leaves. During the addition of ξ, at each node of the path, the algorithm updates the G-statistic for every variable of $\mathbf{V}_N$ with $|\mathbf{V}_N| \leq |\mathbf{X}|$. □

The behavior of IMDDI drastically depends on the nature of the observation of ξ: it may happen that adding ξ to the tree does not lead to any structural change in T. From time to time, adding ξ will imply such a change, the complexity will be very different. It has to be noted that even if the value of a variable V in ξ has never been encountered, the complexities given here will not change.

Property 2 (Updating T with no changes – algorithm 3)**.** *The complexity of updating the tree with no structural change is in* $\mathcal{O}(|T||\mathbf{X}|)$.

Proof. Since no structural change will take place during the iteration over the boundaries, every node N of the ODT T will participate only once to the computation of the aggregated score over a boundary. N will do this computation for each variable of $\mathbf{V}_N$ with (as above) $|\mathbf{V}_N| \leq |\mathbf{X}|$. □

Property 3 (Installing a new variable – algorithm 3)**.** *The complexity of installing a new variable at a given node* N *is* $\mathcal{O}(|\Omega_N| + |\mathbf{X}|)$.

Proof. The installation of a new variable V at a node N requires creating a whole new set of $(Dom(V))$ children for the node N. Let $c_N(i)$ be the child for the ith value of the new variable V installed at N. This child $c_N(i)$ will now include all the observations of Ω_N where V is equal to its ith value. To find these observations (and those for the others children), it is necessary to go through Ω_N.

Then the p-values ($\leq |\mathbf{X}|$) have to be computed for every new node. □

The next important step of IMDDI is the reduction of the ODT into a MDD.

Property 4 (Merging T into an MDD – algorithm 4)**.** *The complexity of reducing the* ODT *is in* $\mathcal{O}(|T|^2)$.

Proof. As Bryant (1986) demonstrates, the merging of an ODT where the leaves are exact value is in $\mathcal{O}(|TlogT|)$. However, IMDDI has to merge similar probability distributions on the leaves and not exact values. This merging hence has to be cut in two parts. One part is the merging of internal nodes. This parts remains in $\mathcal{O}(|TlogT|)$. The merging of the leaves is different. Indeed, initially, two p-values are needed for each pair of leaves ($2|leaves|^2$ values are then computed). Each merging computes again the p-values for the newly created leaf with all the other remaining leaf ($|leaves|$ values are then computed). However, there will be at most $|leaves|$ aggregations (there can not be more merging than the number of existing leaves). Hence, this stage computes at

most $2|leaves|^2 + |leaves| * |leaves|$ values. Altogether, the complexity for this stage is in $|leaves|^2$. Since $|T|/2 \leq |leaves| \leq |T|$, the complexity of the whole algorithm is $\mathcal{O}(|T|log|T|) + \mathcal{O}(|T|^2) = \mathcal{O}(|T|^2)$. □

These complexities demonstrate that IMDDI is an algorithm dedicated to online learning: the size of the base of observations Ω is not a parameter for those complexities. Only one sub-algorithm complexity depends on the sub-part Ω_N installed at node N. Note also that the two most complex algorithms (installing and merging) will not occur each time a new observation ξ is taken into account. As the next section will demonstrate, the re-evaluation of the MDD does not occur often.

4.4. Experimental Validation of IMDDI

In order to analyze the behavior of our algorithm IMDDI, we compared it to ITI since it is the online learning algorithm for DTs. However, ITI is an algorithm that produces unordered DTs. Hence, we also compared IMDDI to the extended version ITI+DD which reorders the tree learned by ITI with the heuristic used in IMDDI and reduces it into an MDD. These three algorithms were written in Python. Oliver(1993) and Kohavi and Li(1995) propose batch MDD learning algorithms that would also be interesting to compare with. Unfortunately, to the best of our knowledge, no working implementation for these algorithms is accessible.

We tested our algorithms on several databases of observations $\xi = \langle \mathbf{x}, y \rangle$. Each database was associated to a different set of variables $\mathbf{X}$ and to a different distribution $P(Y|\mathbf{X})$. To generate the distributions $P(Y|\mathbf{X})$, we retained three different settings for each original model: i) MDD, ii) DT, and iii) Bayesian Networks. In the first setting, IMDDI can find the original model. In the two last settings, the representation of $P(Y|\mathbf{X})$ as an MDD necessarily leads to approximation. For the second setting, the challenge is to see if a learned MDD is able to represent a DT efficiently, despite the absence of explicit isomorphic subgraphs and global order. Finally, a Bayesian Network implies conditional

dependencies between Y and the set of variables $\mathbf{X}$ but does not compel the existence of context-specific independences that are existing in MDDs or a DTs.

For each setting, 20 random instances were generated. For each instance, the size of $\mathbf{X}$ was 10; the domain size of each variable was randomized (up to 5 values); then the structure and the distribution of the instance were randomly chosen. The randomized structure of the model may imply that only a subset of $\mathbf{X}$ is needed for the estimation of Y. From each instance, databases of 20 000 observations and of 10 000 observations were generated. The first database was used to learn the model whereas the second one was used to compare the log likelihood of the learned models.

4.5. Quality of the learned models

For a qualitative comparison , we propose to use two criteria: the log likelihood of the second database according to the learned model and the size (in term of nodes) of the learned model (MDDs or DTs). The log likelihood is observed to ensure that IMDDI does not degrade too much the quality of the learned probability distribution in comparison to both ITI and ITI+DD. And, as we need a model as compact as possible, the size criterion ensures that there is a gain in the size of the learned models.

	IMDDI vs ITI		IMDDI vs ITI+DD	
	Size	Log Likelihood	Size	Log Likelihood
MDDs	63.34% $\pm 9,51\%$	100.96% $\pm 1,53\%$	101.15% $\pm 3,46\%$	100.01% $\pm 0.04\%$
DTs	69.44% $\pm 15.44\%$	101.93% $\pm 1.89\%$	101.52% $\pm 15.22\%$	100.03% $\pm 0.09\%$
BNs	60.51% $\pm 8.73\%$	100.69% $\pm 1.25\%$	105.53% $\pm 5.83\%$	99.94% $\pm 0.39\%$

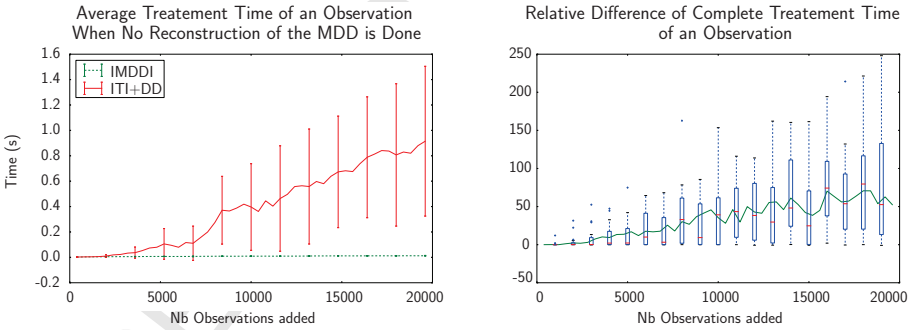
Table 2: Log likelihood and size comparison (average $\pm$ standard deviation) between IMDDI, ITI and ITI+DD

Table 2 shows the results obtained by averaging over the twenty instances for each setting. The numbers are the relative difference between IMDDI and ITI or ITI+DD. For instance, the first number 63.34% means that on average, IMDDI gives a decrease of 36.66% for the size of the learned model compared

to ITI with the first setting (MDD). According to this table, the IMDDI strategy is as interesting as ITI or ITI+DD from a quality of the solution point of view. Moreover, the learned models are clearly more compact than the ones obtained with ITI. Compared to ITI+DD, the results both in terms of likelihood and in terms of size do not lead to an improvement from our algorithm. However, it has to be remembered that ITI+DD learned the MDD from scratch at each iteration.

4.6. Computation time

The other criterion on which we challenged IMDDI and ITI+DD is the computational time. We compared the time spent to take into account a newly arrived observation. Figure 12a shows the average time spent to take into account a new observation when no reduction is applied to the decision diagram. It clearly demonstrates how the time spent to reorder the tree in the ITI+DD strategy completely renders it inefficient: the reordering part becomes much slower for ITI+DD as the tree grows.



(a) (b)

Figure 12: Two comparisons between IMDDI and ITI+DD: 12a time spent to take into account a new observation with no revision of the MDD and 12b evolution of the relative change of the total time to take a new observation into account.

The steps with reconstruction of the MDDs experimentally represent from 15% to 7% of the observations. In that case, on all our experiments, the IMDDI step is still shorter than the ITI+DD step. Indeed, in Figure 12b, we show the relative differences between the total times taken by IMDDI and ITI+DD to handle

a new observation: the central curve demonstrates that on average, in spite of the reduction to the MDDs, the IMDDI algorithm remains fifty times faster than the ITI+DD algorithm. These experiments illustrate that our algorithm outperforms ITI in terms of quality (same likelihood, better compactness) and outperforms a straightforward ITI+DD in terms of computation time.

4.7. Online learning of MDD*s*

This section presented IMDDI, the first online learning algorithm for MDDs. It describes the different phases: addition of an observation, update of the structure, and reduction into a decision diagram and shows their complexities proving that IMDDI is adequate for on-line learning. It also experimentally verifies the compactness and the accuracy of the learned models in comparison with algorithms ITI and ITI+DD. Finally, it illustrates that IMDDI is much faster than a straightforward ITI+DD learning strategy.

It is noteworthy to mention that the IMDDI algorithm needs no prior on the variables: no knowledge about the number of variables or even their domains are required and can be dynamically discovered during the learning.

5. SPIMDDI: a SDYNA instance with Multi-valued Decision Diagrams

IMDDI proves to be an efficient online MDD learning algorithm. Because a motivation for IMDDI is to substitute the DTs with the MDDs in the SDYNA framework, the next step is naturally to integrate it in a SDYNA instance, jointly with the SPUMDD algorithm for the planning phase. Hence, the obtained SPIMDDI instance can perform reinforcement learning tasks in a large environment by only manipulating MDDs.

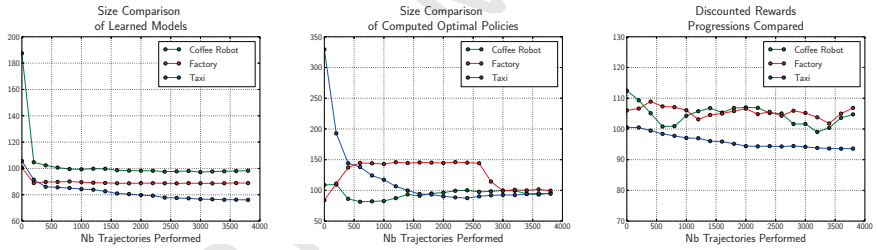
5.1. Validation of the SPIMDDI instance

This section covers experiments made on SPIMDDI on three classical problems from the literature: *Coffee Robot*, *Factory* (Dearden and Boutilier, 1997) et *Taxi* (Dietterich, 1998). In the *Taxi*, a driver has to pick up his client at a point A

and bring him to a point B while navigating in a city modeled as a graph and managing his gas tank. *Taxi* is a large multivalued issue; the smallest domain size is 4 while the biggest is 14.

In order to be fair in our comparisons, both SPIMDDI and SPITI were coded in C++ in the same software library. Figure 13 shows three comparison criteria between SPIMDDI and SPITI on the three problems mentioned above. The curves were obtained by averaging over 20 experiments of 4000 trajectories each. A trajectory was composed of 25 decision makings before the system current state was randomly reset. Between experiments, the learned model was also reset.

Figure 13a compares the size of the learned models; this size is the total number of nodes in every learned function representation (every probability distribution and every reward function) as DTs or MDDs. We test here if the "better compactness" objective is achieved. Results show that on the three examples, IMDDI learns a more compact description of the problems.



(a) (b) (c)

Figure 13: Ratio $\frac{SPIMDDI}{SPITI}$ (in %) in terms of (a) the size of the learned models (transitions and rewards), (b) the size of the optimal planned policy , and (c) the obtained discounted rewards during the experiments.

Figure 13b compares the size of the obtained optimal policies (in number of nodes). On every problem, final policies are of comparable size. Moreover, in Figure 13c, the similarity between discounted rewards[9] for both instances tends

[9]Upon each observation, discounted rewards for both algorithms are updated according to the following formula: $R_{n+1} = r_n + \gamma \cdot R_n$ where r_n is the reward obtained with the observation.

to indicate that SPIMDDI learns a policy that is as efficient as the SPITI one. However, the number of possible actions in these problems may be too low to make a meaningful difference between the optimal policy expressed as a DT and the optimal policy expressed as an MDD.

Figure 14 shows that IMDDI learns a near optimal representation of structured transition (sub-figure a,b,c) while computing a policy (sub-figure d,e,f) based upon this model. The quality of the learned models and of the optimal policy tends to validate our approach and suggests that IMDDI is a good candidate for a SDYNA algorithm using MDD.

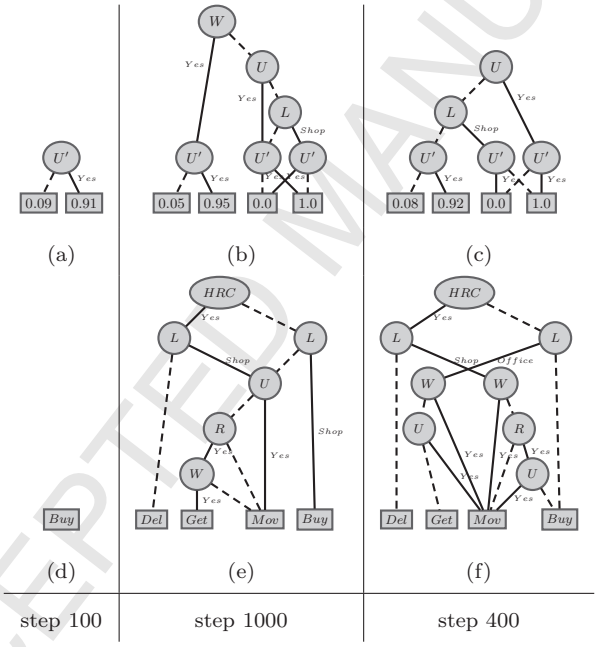


Figure 14: Evolutions of the transition model $P(U'|X, Get)$ [a,b,c] and of the optimal policy [d,e,f]. In step 400 the transition distribution is close to the original and the optimal policy is equivalent to the policy for the original MDP.

5.2. Application to a real problem

While the three problems on which we tested SPIMDDI are challenging in several aspects, they remain artificial. As a consequence, we looked at a less

established problem: the management of units in the StarCraft™[10] video game. The objective is to see if SPIMDDI is able to learn a factored representation of the environment in which the given units evolved.

Several sensors (i.e. variables) are used to characterize the situation of the unit: unit's number of life points, unit's localization on the map (abstract one like near enemies or allies, close to unexplored areas, ...), is the unit under attack?, number of allies and enemies visible, has the unit a target or not, etc. These sensors lead to a virtual state space of size 16 000 states. The unit can performed high level actions: explore an unknown area of the map, attack a visible enemy, go on patrol, support allies or go to the last place where an enemy unit was seen. The rewards are based on whether the game is won or lost, if the unit has died and if the unit has explored new (geographical) areas. Each unit is an agent that learns and applies the same FMDP. There is no centralized decision but the experiences are shared and the process of decision making is the same among the units. Figure 15 shows the setting of the experiment with the game in the upper right corner, and the different programs that control the interactions between SPIMDDI and the game.

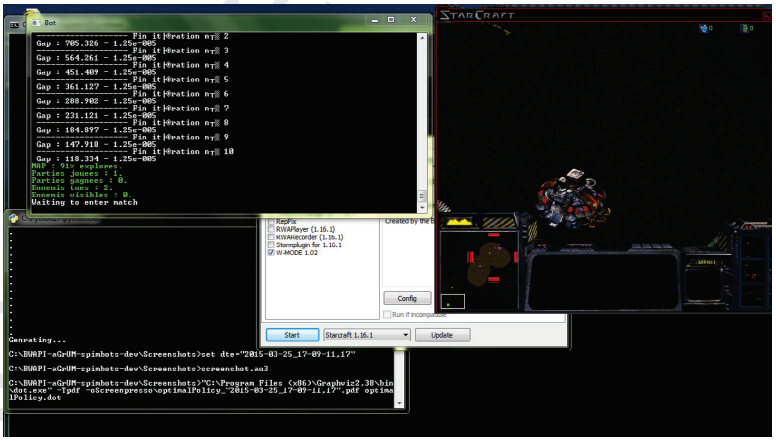


Figure 15: A look at SPIMDDI used in the video game StarCraft™.

[10] http://eu.battle.net/sc2/fr/

While this is still a work in progress, preliminary results tend to validate the efficiency of SPIMDDI. Indeed, the diverse MDDs produced by SPIMDDI during the experiments are all compact representations of the state space; meaning there are far fewer paths in these representations from the root to any leaf than there are states in the problem: the average size is of 200 internal nodes (proving the ability of abstraction of the model). Moreover, the MDDs learned are often much more compact than trees. For instance, Figure 16 shows the optimal policy that SPIMDDI managed to compute after 50 games. SPIMDDI clearly shows its ability to extract compact structured representations from an unknown problem.

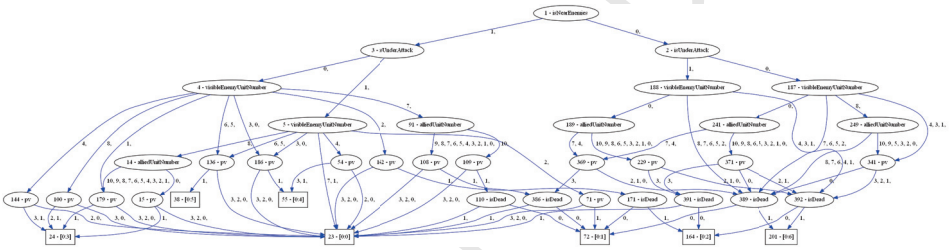


Figure 16: The optimal policy computed after 50 games.

6. Conclusion

Surprisingly, even if the advantages of MDDs over DTs are evident in terms of size and compactness, their use in Reinforcement Learning is minimal (to the best of our knowledge). This article aims to fix this lack. It first shows how building a planning algorithm that takes into account the characteristics of MDDs mainly by dropping the constraint of a same global order for all the function graphs that describes the behaviour of the systems. It then describes an on-line learning algorithm for MDDs that allows building a first incremental scheme for FMDPs using MDDs as function graphs for conditional probability distributions and rewards. Theses two contributions can be combined in an new instance of SDYNA, the Reinforcement Learning framework for factored domains. Finally,

all these algorithms are written in C++, are available in the aGrUM[11] library and could be exploited in an application dedicated to find optimal policies for instantiations of the StarCraft™game.

References

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., Somenzi, F., 1997. Algebric decision diagrams and their applications. Formal Methods in System Design 10 (2-3), 171–206.

Banach, S., 1922. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. Fundamenta Mathematicae 3 (1), 133–181.

Bellman, R., 1957. Dynamic Programming, 1st Edition. Princeton University Press, Princeton, NJ, USA.

Bellman, R., 1961. Adaptive Control Processes. Princeton University Press.

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

Boutilier, C., Dearden, R., Goldszmidt, M., et al., 1995. Exploiting structure in policy construction. In: IJCAI. Vol. 14. pp. 1104–1113.

Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D., 1996. Context-specific independence in Bayesian Networks. In: Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., pp. 115–123.

Boutilier, C., Reiter, R., Price, B., 2001. Symbolic dynamic programming for first-order mdps. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1. IJCAI'01. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 690–697.

[11] http://agrum.lip6.fr

Breiman, L., Friedman, J., Olshen, R., Stone, C., 1984. Classification and Regression Trees. Wadsworth & Brooks.

Bryant, R. E., 1986. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 100 (8), 677–691.

Dean, T., Kanazawa, K., 1989. A model for reasoning about persistence and causation. Computational Intelligence 5 (2), 142–150.

Dearden, R., Boutilier, C., 1997. Abstraction and approximate decision-theoretic planning. Artificial Intelligence 89 (1–2), 219 – 283.

Degris, T., Sigaud, O., Wuillemin, P.-H., 2006a. Chi-square tests driven method for learning the structure of factored MDPs. In: UAI-22. AUAI Press, pp. 122–129.

Degris, T., Sigaud, O., Wuillemin, P.-H., 2006b. Learning the structure of factored Markov decision processes in reinforcement learning problems. In: Proceedings of the 23rd International Conference on Machine learning. ACM, pp. 257–264.

Dietterich, T. G., 1998. The MAXQ method for hierarchical reinforcement learning. In: Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998). Morgan Kaufmann, pp. 118–126.

Dunning, T., Mar. 1993. Accurate methods for the statistics of surprise and coincidence. Computational Linguistic 19 (1), 61–74.

Guestrin, C., Gordon, G., 2002. Distributed planning in hierarchical factored MDPs. In: Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., pp. 197–206.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., Boutilier, C., 1998. Hierarchical solution of Markov Decision Processes using macro-actions. In: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., pp. 220–229.

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

Howard, R. A., 1960. Dynamic Programming and Markov Processes. MIT Press.

Kohavi, R., Li, C., 1995. Oblivious decision trees, graphs, and top-down pruning. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann, pp. 1071–1077.

Magnan, J.-C., Wuillemin, P.-H., 2013. Improving decision diagrams for decision theoretic planning. In: Proceedings of the Twenty-Sixth International Florida Artificial Intelligence Research Society Conference. pp. 621–626.

Magnan, J.-C., Wuillemin, P.-H., 2015. On-line learning of multi-valued decision diagrams. In: Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference. pp. 576–580.

Markov, A., Nagorny, N., 1988. The Theory of Algorithms. Mathematics and its Applications (Kluwer Academic).: Soviet Series. Springer.

McDermott, S. H. D., Hanks, S., McDermott, D., 1993. Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. Artificial Intelligence 66, 1–55.

Mingers, J., 1989. An empirical comparison of selection measures for decision-tree induction. Machine Learning 3 (4), 319–342.

Murphy, K. P., 2002. Dynamic Bayesian Networks: Representation, inference and learning. Ph.D. thesis, University of California, Berkeley.

Oliver, J. J., 1993. Decision graphs - an extension of decision trees. In: Proceedings of the Fourth International Workshop on Artificial Intelligence and Statistics. pp. 343–350.

Parr, R. E., 1998. Hierarchical control and learning for Markov decision processes. Ph.D. thesis.

Puterman, M., 2005. Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley series in probability and statistics. Wiley-Interscience.

Quinlan, J. R., 1993. C4.5: Programs for Machine Learning. San Francisco, CA, USA.

Rudell, R., 1993. Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design. IEEE Computer Society Press, pp. 42–47.

Srinivasan, A., Ham, T., Malik, S., Brayton, R. K., 1990. Algorithms for discrete function manipulation. In: ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on Computer-Aided Design. IEEE, pp. 92–95.

St-Aubin, R., Hoey, J., Boutilier, C., 2000. APRICODD: Approximate policy construction using decision diagrams. In: Proceedings of Conference on Neural Information Processing Systems. pp. 1089–1095.

Sutton, R. S., 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Proceedings of the Seventh International Conference on Machine Learning. pp. 216–224.

Utgoff, P. E., Berkman, N. C., Clouse, J. A., 1997. Decision tree induction based on efficient tree restructuring. Machine Learning 29 (1), 5–44.

Wang, C., Joshi, S., Khardon, R., 2008. First order decision diagrams for relational MDPs. Journal of Artificial Intelligence Research, 431–472.

White, A. P., Liu, W. Z., 1994. Technical note: Bias in information-based measures in decision tree induction. Machine Learning 15 (3), 321–329.