Considering Unseen States as Impossible in Factored Reinforcement Learning

Olga Kozlova[1,2], Olivier Sigaud[1], Pierre-Henri Wuillemin[3], and Christophe Meyer[4]

[1] Institut des Systèmes Intelligents et de Robotique
Université Pierre et Marie Curie - Paris 6, CNRS UMR 7222
4 place Jussieu, F-75005 Paris, France
Olivier.Sigaud@upmc.fr
[2] Thales Security Solutions & Services, Simulation
1 rue du Général de Gaulle, Osny BP 226
F95523 Cergy Pontoise, France
Olga.Kozlova@thalesgroup.com
[3] Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie - Paris 6, CNRS UMR 7606
4 place Jussieu, F-75005 Paris, France
Pierre-Henri.Wuillemin@lip6.fr
[4] Thales Security Solutions & Services, ThereSIS Research and Innovation Office
Campus Polytechnique
1, avenue Augustin Fresnel
91767 Palaiseau, France
Christophe.Meyer@thalesgroup.com

Abstract. The Factored Markov Decision Process (FMDP) framework is a standard representation for sequential decision problems under uncertainty where the state is represented as a collection of random variables. Factored Reinforcement Learning (FRL) is an Model-based Reinforcement Learning approach to FMDPs where the transition and reward functions of the problem are learned. In this paper, we show how to model in a theoretically well-founded way the problems where some combinations of state variable values may not occur, giving rise to impossible states. Furthermore, we propose a new heuristics that considers as impossible the states that have not been seen so far. We derive an algorithm whose improvement in performance with respect to the standard approach is illustrated through benchmark experiments.

1 Introduction

Based on the Markov Decision Process (MDP) framework (Sutton & Barto, 1998), the Factored Markov Decision Process (FMDP) framework (Boutilier et al., 1995) is the standard representation of sequential decision problems under uncertainty when the state of the problem can be decomposed as a set of random variables. In this framework, the state space of a sequential decision problem is represented as a collection of random variables $X = \{X_1, \ldots, X_n\}$. A state is then defined by a vector $x = (x_1, \ldots, x_n)$ with $\forall i, x_i \in \text{Dom}(X_i)$. FMDPs exploit the structure of the dependencies between variables to represent large MDPs compactly. For each action a, the transition model is defined by a separate Dynamic Bayesian Network (DBN) model (Dean &

W. Buntine et al. (Eds.): ECML PKDD 2009, Part I, LNAI 5781, pp. 721–735, 2009.

Kanazawa, 1989). The model G_a is a two-layer directed acyclic graph whose nodes are $\{X_1, \ldots, X_n, X'_1, \ldots, X'_n\}$ with X_i a variable at time t and X'_i the same variable at time $t+1$. The parents of X'_i are noted $\text{Parents}_a(X'_i)$. The transition model is quantified by *Conditional Probability Distributions* (CPDs), noted $P^a(X'_i|\text{Parents}_a(X'_i))$, associated to each node $X'_i \in G_a$.

The authors of (Boutilier et al., 2000) propose two structured Dynamic Programming (SDP) algorithms, namely Structured Value Iteration (SVI) and Structured Policy Iteration (SPI). These algorithms and later extensions like SPUDD (Hoey et al., 1999) deal with the case where there are no synchronic arcs in the DBNs, i.e. $\forall i, \text{Parents}_a(X'_i) \subseteq \{X_1, \ldots, X_n\}$ and $\text{Parents}_a(X_i) = \emptyset$. Thus, in such a model, the X'_i are independent of each other conditionally to $\{X_1, \ldots, X_n\}$. The independence assumption results in the opportunity to compute the joint probability of a collection of variables as a product: $P(X'|X) = \Pi_i P(X'_i|\text{Parents}_a(X'_i))$.

In this paper, we focus on the case where the structure and parameters of the FMDP are learnt from experience, and we address a wider class of problems where particular combinations of values of variables may not occur. We argue that this situation often happens in practice and we show through benchmark experiments that modifying standard algorithms to deal with such "impossible states" is more efficient than just ignoring this phenomenon. Finally, we show that considering as impossible a state that has never been seen so far is an efficient heuristic to learn quickly in FMDPs.

The paper is organized as follows. In the next section, we give a brief overview of SDP algorithms, insisting on the steps where the independence assumption is used and we give a brief overview of an algorithm that learns the structure of the FMDP while solving it. Then we show in section 3 how the presence of impossible states can be modeled and how the algorithms must be modified as a consequence. In section 4, we examine through benchmark experiments the benefits that can result from our method when impossible states are present. In section 5, we discuss these benefits depending on the rate of impossible states, before concluding on the possibility to extend this work in different directions.

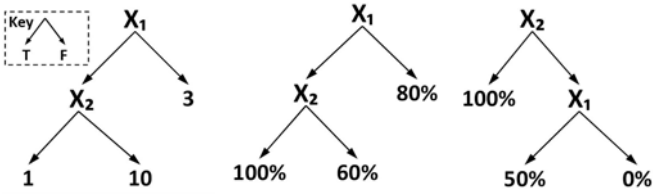
2 Factored Reinforcement Learning

In this section we briefly present structured dynamic programming algorithms and then SPITI, a model-based reinforcement learning approach dedicated to FMDPs.

2.1 Structured Dynamic Programming

Standard SDP algorithms such as SVI and SPI use decision trees as factored representation. In the rest of the paper, a function F represented as a decision tree is noted $Tree(F)$. SVI and SPI can be seen as an efficient way to perform the Bellman-backup operation on trees, expressed as follows:

$$Tree(Q_a^V)(x) = Tree(R(x,a)) + \gamma Tree(\sum_{x'} P(x'|x,a)V(x')). \quad (1)$$



(a) $Reward$ (b) $P(x'_1|X_1, X_2, a_0)$ (c) $P(x'_2|X_1, X_2, a_0)$

Fig. 1. A toy example: the tree representation of a reward function and of the transition functions of binary variables X_1 and X_2 given an unique action a_0. Notations are similar to the ones in (Boutilier et al., 2000), but in the boolean case we note x_i for $X_i = true$ and $\bar{x}_i$ for $X_i = false$.

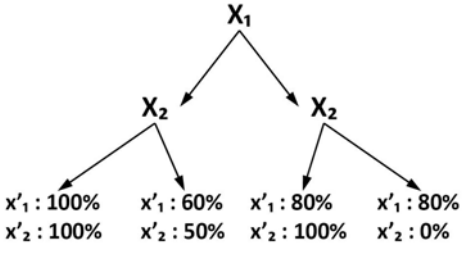


Fig. 2. $P(X'|X, a_0)$ computed by the $PRegress$ operator from the example of Fig. 1

In SVI and SPI, the $Regress(Tree(V), a)$ algorithm performs Bellman-backup operations. Inside $Regress$, $PRegress$ computes $Tree(\sum_{x'} P(x'|x, a)V(x'))$ using the structure of $Tree(P)$ and $Tree(V)$.

Consider the toy example whose reward and transition functions for an unique action a_0 are given in Fig. 1.

From the transition trees, $PRegress$ first computes the $Tree(P(X'|X, a_0))$ shown in Fig. 2. This tree represents the individual probabilities of each variable value at $t + 1$ given the variable values at t. For instance, the rightmost branch of the tree reads as follows: if X_1 and X_2 were false, the probability that X'_1 and X'_2 are true are 80% and 0% respectively. Note that the probability of one of the values can be omitted in the representation and inferred from the other probabilities. Furthermore, $PRegress$ only computes the combinations of values that are necessary to perform regression from the current value function (see (Boutilier et al., 2000) for details).

Given the tree represented in Fig. 2 and considering that the variables at $t + 1$ are independent conditionally to those at t, $PRegress$ computes the joint probabilities as a product, as shown in Table 1. Note that the table representation is not computed explicitly in the algorithm: in the more general case with any number of variables and enumerated values, this calculation is implemented by expanding a tree of all variable values combinations and computing probabilities at the leaves as a product on individual probabilities.

The last step of $PRegress$ computes $\sum_{x'} P(x'|x, a)V(x')$ using the structure shown in Fig. 3. Then $Regress$ computes $Tree(Q_a^V)$ according to equation (1) by performing the product with γ and the sum with $Tree(R(X, a))$.

Table 1. Probabilities for joint variables, resulting from Fig. 2

$P(X'\|X, a_0)$	$x'_1x'_2$	$x'_1\bar{x_2}'$	$\bar{x_1}'x'_2$	$\bar{x_1}'\bar{x_2}'$
x_1x_2	100%	0%	0%	0%
$x_1\bar{x_2}$	30%	30%	20%	20%
$\bar{x_1}x_2$	80%	0%	20%	0%
$\bar{x_1}\bar{x_2}$	0%	80%	0%	20%

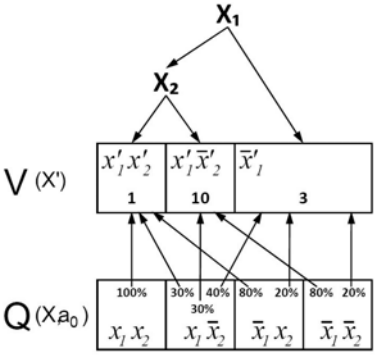


Fig. 3. Regression with structured representations using the values computed in Table 1. We consider the first iteration where $Tree(V) = Tree(R)$. Notice that transitions to $\bar{x_1}x_2$ and $\bar{x_1}\bar{x_2}$ are summed to $\bar{x_1}$.

On top of $Regress$, SVI and SPI behave differently. In SVI, the value function $Tree(V)$ is computed by merging the set of action-value functions $Tree(Q_a^V)$ using maximization as combination function. It is shown in (Boutilier et al., 2000) that, given a perfect knowledge of the transition and reward functions and starting with $Tree(V_0) = Tree(R)$, $Tree(V)$ converges in a finite number of time steps to the optimal value function $Tree(V^*)$. Then one can extract $Tree(\pi^*)$ from $Tree(V^*)$ using a simple tree-based greedy operator.

In SPI, the process is slightly more complex. Policies $Tree(\pi)$ and value functions $Tree(V^\pi)$ are computed iteratively until convergence, making profit of the structure of $Tree(\pi)$ to optimize the computation of $Tree(V^\pi)$. We do not detail the algorithm since ours is based on SVI, but transferring the approach described in section 3 to SPI is straightforward.

2.2 SPITI

Reinforcement Learning in FMDPs is generally about the case where the structure of the DBNs are given, but the parameters of the CPDs are learnt from experience. By contrast, we call *Factored Reinforcement Learning* (FRL) the case where the structure of the DBNs itself is learnt.

An implementation of FRL is expressed in the SDYNA framework (Degris et al., 2006a; Degris et al., 2006b) as a structured version of the DYNA architecture (Sutton,

1991). In SDYNA, the model of transitions and of the reward are learned from experience under a compact form. The inner loop of SDYNA is decomposed into three phases:

- *Acting*: choosing an action according to the current policy, including some exploration;
- *Learning*: updating the model of the transition and reward functions of the FMDP from $\langle X, a, X', R\rangle$ observations;
- *Planning*: updating the value function $Tree(V)$ and policy $Tree(\pi)$ using one sweep of SDP algorithms.

SPITI is a particular instance of SDYNA using ϵ-$greedy$ as exploration method, the *Incremental Tree Induction* (ITI) algorithm (Utgoff, 1989) to learn the model of transitions and reward functions as a collection of decision trees, and the inner loop of SVI as planning method. An algorithmic description is given in (Degris et al., 2006b).

3 Dealing with Impossible States

The techniques presented so far address the case where there are no synchronic arcs in the DBNs that represent the structure of FMDPs. On the other extreme, the authors of (Boutilier et al., 2000) propose to model the case where the variables are not independent by adding synchronic arcs between variables at $t + 1$ and recording the joint probabilities of all groups of variables that are connected by such synchronic arcs. This results in more complex, slower and more memory-intensive algorithms, but that can deal with a much wider class of problems. A more detailed study of that case is presented in (Boutilier, 1997).

In this paper, we address a class of problems that is intermediate between the "no synchronic arcs" and the "any synchronic arcs" classes. It corresponds to problems where the variables at $t + 1$ behave as if they were independent, but some combinations of values for some variables do not occur in practice, which contradicts the independence assumption. We show below how such a situation can be modeled without using the general class of problems with synchronic arcs in the DBNs.

3.1 Modeling Impossible States

The class of problems we want to address can be modeled with the kind of DBNs shown in Fig. 4, given that there is one such DBN for each action. In this representation, there are no synchronic arcs between variables at $t + 1$, but there are some constraints K on whether some combinations of variable values are possible or not. K (resp. K') stands for the knowledge of impossible states x (resp. x'). The values of K and K' are either $true$ or $false$ depending on the possibility of the corresponding states.

As we illustrate in the experimental section, without using such constraints, the $Tree(V)$ and $Tree(Q_a^V)$ structures may represent many states that do not occur in practice. Dealing with the constraints explicitly is a way to avoid the computational and memory overhead resulting from this useless information by filtering out all impossible states in the data structures.

We show below that this filtering can be performed safely just by discarding the impossible states and normalizing again the probabilities when it is necessary.

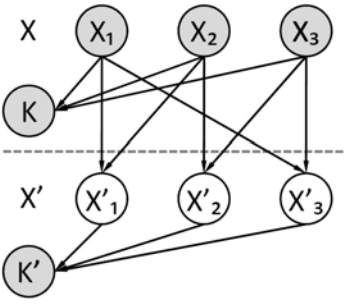


Fig. 4. DBN representation for problems with independent variables and impossible states. K and K' stand for constraints stating if a particular combination of values is possible. Variables in gray are observed. K' and relevant X_j are used to predict X'_i for all i. K' does not necessarily depend on all X'_i.

3.2 Impact on Regression

Let us show that the values computed by $PRegress$ taking the constraints into account, i.e. $\Sigma_{x'} P(X'|x,a,k').V(x')$, are equal to the values one would obtain without taking these constraints into account, just leaving the impossible states away and normalizing again the probabilities.

First, with the representation above, the probability distribution of X' for a particular action a can be computed given the values of the variables x_i and the value k' of the common constraint K'. Indeed, the distribution of X' can be expressed as $P(X'|x,k') \propto P(k'|X',x)P(X'|x)$. Furthermore we have $P(K'|X',X) = P(K'|X')$, since $K' \perp\!\!\!\perp X|X'$. Thus, we have:

$$P(X'|x,k') \propto P(k'|X')P(X'|x) \tag{2}$$

and $P(k'|X') = 0$ or 1 depending on the constraint.

If the variables are independent, we have $P(X'|x,a,k') = \prod_i P(X'_i|x,a) = \prod_i P(X'_i|parent(X'_i),a)$, thus we are in the standard context where the proof of convergence given in (Boutilier et al., 2000) applies.

Now, if we consider impossible states, from (2) we have

$$\sum_{x'} P(X'|x,a,k') = \sum_{x'} N_{x,a} P(k'|X')P(X'|X,a) \tag{3}$$

where $N_{x,a}$ is a normalization factor such that

$$\forall x, \forall a, \sum_{x'} N_{x,a} P(k'|X')P(X'|X,a) = 1.$$

In (3), there are two categories of terms. If X' corresponds to impossible states, we have $P(k'|X') = 0$ and the corresponding term is removed. Otherwise, the state variables in X' are independent thus $P(X'|x,a,k') = \prod_i P(X'_i|parent(X'_i),a)$ and $P(k'|X') = 1$. Thus (3) can be simplified as

Table 2. Transition probabilities for joint variables resulting from Fig. 2, given that $\bar{x_1}'x_2'$ is impossible

$P(X'\|X, a_0, k')$	$x_1'x_2'$	$x_1'\bar{x_2}'$	$\bar{x_1}'\bar{x_2}'$
x_1x_2	100%	0%	0%
$x_1\bar{x_2}$	37.5%	37.5%	25%
$\bar{x_1}\bar{x_2}$	0%	0%	100%

$$\sum_{x'} N_{x,a} P(k'|X') \prod_i P(X_i'|parent(X_i'), a))$$

where only the existing states remain. We are back to the situation where we consider only possible states with independent variables and the proof from (Boutilier et al., 2000) applies again.

From the result above, it turns out that, if we take the constraints into account, the values can be computed in $PRegress$ as in the case without dependencies, just discarding the impossible states and normalizing again so that the sum of probabilities over all remaining states is 1.

Note that this way to remove impossible states in the computation of $PRegress$ is the only one which results in the possibility to renormalize. Otherwise, if, for instance, we remove the leaves corresponding to impossible states in $Tree(Q_a^V)$ or $Tree(V)$, we cannot perform the normalization since the probability information is lost in these trees. Furthermore, in addition to being theoretically well-founded, this way of filtering out impossible states at the heart of the $PRegress$ operator is much more efficient than filtering later on, since this other solution would result in expanding the number of leaves in the value tree before reducing it, which is exactly what we want to prevent.

To illustrate our approach, let us consider again the example given in the previous section. Now, assume that we have some information that the state $\bar{x_1}x_2$ is impossible. As a consequence, $P(\bar{x_1}'x_2')$ is null and the probability of any state at $t+1$ given $\bar{x_1}x_2$ is pointless. Thus, the corresponding probabilities in Table 1 must be filtered out and the remaining values must be renormalized as shown in Table 2. Here again, Table 2 is not computed explicitly, it is represented as a tree as in the standard case, adding in the algorithm the filtering out of the branches corresponding to impossible states and finally normalizing again the values at the leaves.

3.3 Impact on Other Tree Operations

After modifying $PRegress$ as presented above, the trees corresponding to (3) are free from impossible states for all actions.

But then, performing a Bellman-backup according to equation (1) and the other operations required to run SVI or SPI implies some operations over the resulting trees. As exemplified in Fig. 5, despite the filtering performed in $PRegress$, simple operations on several value-related trees ($Tree(R)$, $Tree(V)$, $Tree(Q_a^V)$) can generate leaves representing impossible states if these trees do not share the same structure. Indeed, whereas generalization over identical values can "hide" the expression of impossible states in

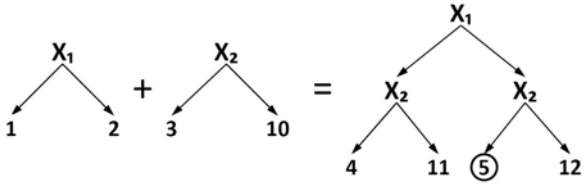


Fig. 5. Combining two trees can generate leaves about impossible states (here, $\bar{x_1}x_2$ is impossible)

the source trees, the operations can make these states appear in the resulting trees. As a result, we must filter impossible states out in such operations.

3.4 IMPSPITI

So far, we have described modifications that apply to SDP algorithms. To evaluate the impact of these considerations in the context of FRL, we design a new system inspired from SPITI, called IMPSPITI, that incorporates the following modifications in its *Planning* phase:

- in $PRegress$, when computing the joint probabilities over variable values, the branches in the tree corresponding to impossible states are discarded and the probabilities are normalized again;
- in the sum $Tree(R(x,a)) + \gamma \sum_{x'} Tree(P(x'|X,a))Tree(V(x'))$ and the maximization $Tree(V) = \text{argmax}_a\, Tree(Q_a^V)$, the branches of the resulting tree corresponding to impossible states are discarded;

The model of transition and reward functions being learnt from experience, they cannot contain impossible states, thus the *Learning* phase does not need to be modified. Algorithm 1 schematically describes the planning phase if IMPSPITI.

Finally, we need a function to decide that a state is impossible. Since this function is called at three places in each step of the central iteration, it may result in a significant time overhead.

Algorithm 1. IMPSPITI- *Planning* phase

Input : FMDP $\mathcal{F}[Tree(P(x'|x,a))], Tree(V_{t-1})$;
1. $Tree(Q_a^t) = addTrees[Tree(R(x,a), \gamma.PRegress[Tree(V_t), a, N_{x,a}]]$ for each action $a \in A$ discarding impossible states in addTrees()
2. $Tree(V_t) = MaxMerge_a[Tree(Q_a^t)]$ discarding impossible states in MaxMerge().
3. Return : $Tree(V_t)$ and $\{Tree(Q_a^t), \forall a \in A\}$.

In the experimental section, we will compare two approaches. One consists in using problem specific expert rules. The second is more general, it consists in building a tree where the states already visited by the agent are stored as possible. Thus we consider all states that have not been visited yet as impossible. In the worst case, this representation would boil down to the complete enumeration of states, but this is also true for the trees

manipulated in standard SDP algorithms (see (Boutilier et al., 2000) for a discussion). In most cases of interest, however, this representation will benefit from factorization over states.

4 Experimental Study

To illustrate the benefits of our approach, we perform experiments on two benchmarks: MAZE6 and BLOCKS WORLD. The algorithms are coded in C# and run on Intel Core2Duo 1.80GHz processor with 2Go RAM. All results presented below are averaged over 50 runs where each run performs 50 episodes limited to 50 steps. The ϵ-$greedy$ exploration policy uses $\epsilon = 0.1$.

4.1 Maze6

Maze environments are standard benchmark problems in the Learning Classifier Systems (LCSs) literature, LCSs being a heuristic approach to FRL (see (Sigaud & Wilson, 2007)). Mazes are represented by a two-dimensional grid. Each cell can be occupied by an obstacle, denoted as variable value by a `'1'`, a reward, denoted by a `'R'`, or can be empty, denoted by a `'0'`. The agent perceives the eight adjacent cells starting with the cell to the north and coding clockwise. Fig. 6 shows MAZE6, one of such mazes, designed so that Markov property holds.

For example, an agent located in the cell below the reward perceives `'R1110011'` whereas an agent located as shown in Fig. 6 perceives `'00110101'`. Although there are only 37 actual states within the problem, the combinatorial representation results in $3^8 = 6561$ states. The agent can perform eight actions, the movements to adjacent cells. If a movement leads to a cell containing an obstacle, the action has no effect and there is no penalty. In the stochastic case, the chosen action may result in a move corresponding to an immediately adjacent action, with probability 10%. Once the reward position is reached, the environment provides a reward of 1000 and the episode ends. In that case, the agent starts again in a randomly chosen empty cell.

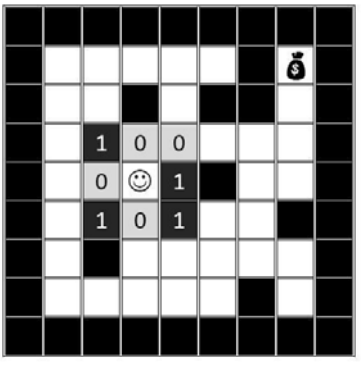


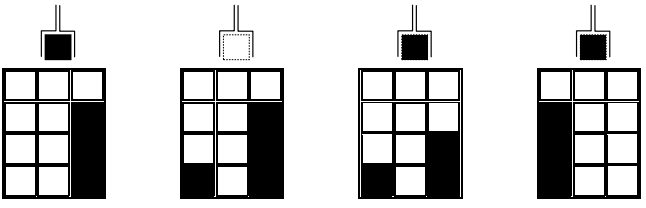
Fig. 6. Maze6

4.2 Blocks World

In the BLOCKS WORLD problem, introduced in (Butz et al., 2002) (see Fig. 7), b blocks are distributed over a given number s of stacks. At the beginning of each episode, blocks are distributed randomly. The agent can manipulate the stacks by the means of a gripper that can either grip or release a block on a certain stack. Additionally, the problem contains a goal state, which consists in putting a particular number $y \leq b$ of blocks on the left hand stack. The stacks are not limited in height. Fig. 7(d) shows the goal in the problem with $b = 4, s = 3, y = 3$.

The complexity of BLOCKS WORLD highly depends on the representation chosen to encode the states. We developed three such representations.

The first is the one used in (Butz et al., 2002). We call it *Binary* representation. The agent perceives the current blocks distribution coding each stack with b variables. One additional variable indicates if the gripper is currently holding a block. In this representation, many arbitrary combinations of variable values correspond to states that



(a) (b) (c) (d)

Fig. 7. A BLOCKS WORLD scenario, from a random initial position (a) to the goal position (d)

Table 3. BLOCKS WORLD representations of the situation given in Fig. 7 (G stands for gripper)

(a)	(b)	(c)	(d)
	BINARY		
0000,0000,1110,1	1000,0000,1110,0	1000,0000,1100,1	1110,0000,0000,1
	STACKS		
STACK_1=0	STACK_1=1	STACK_1=1	STACK_1=3
STACK_2=0	STACK_2=0	STACK_2=0	STACK_2=0
STACK_3=3	STACK_3=3	STACK_3=2	STACK_3=0
G =TRUE	G =FALSE	G =TRUE	G =TRUE
	BLOCKS		
BLOCK_1=S_3	BLOCK_1=S_3	BLOCK_1=S_3	BLOCK_1=G
BLOCK_2=S_3	BLOCK_2=S_3	BLOCK_2=S_3	BLOCK_2=S_1
BLOCK_3=S_3	BLOCK_3=S_3	BLOCK_3=G	BLOCK_3=S_1
BLOCK_4=G	BLOCK_4=S_1	BLOCK_4=S_1	BLOCK_4=S_1

do not occur in practice. Indeed, all states where a block is lying neither on top of another block nor on the table are impossible. The more empty cells in the problem, the more such impossible states.

In the second representation (called *Stacks*), there is one variable per stack giving the number of blocks it contains, and one additional variable indicating if the gripper is holding a block. The impossible states are the states where the total number of blocks is not b. Thus, in that case, an *ad hoc* way to decide if a state is possible consists in simply summing the represented blocks and comparing to b.

Finally, the third representation (called *Blocks*), there are b variables whose values are given by the gripper or stack where the corresponding block is currently placed. The only impossible states are the ones where several blocks are in the gripper, which gives a straightforward *ad hoc* rule to decide if a state is possible. The major drawback of this representation is that, blocks being identical, there are many ways to represent the same state of the problem. For instance, $\{\text{block}_1=\text{s}_1, \text{block}_2=\text{s}_2, \ldots\}$ and $\{\text{block}_1=\text{s}_2, \text{block}_2=\text{s}_1, \ldots\}$ represent the same configuration and there are 12 different ways to represent Fig. 7(c). This results in a greater number of possible states than necessary, thus in more complex value and policy trees structures.

Table 3 shows an example of these three representations with $b = 4, s = 3, y = 3$.

4.3 Empirical Results

Table 4 recaps the performance on MAZE6 of SPITI and IMPSPITI and Fig. 8 shows their convergence speed in number of episodes considering the number of steps needed to perform each episode.

IMPSPITI clearly outperforms SPITI both in the deterministic and in the stochastic case: it converges faster in time and in number of learning episodes, but also requires less memory to represent value and policy functions. More precisely, IMPSPITI only considers the 37 states that are actually possible (the variance in value and policy size comes from cases where the run ended before all states were explored).

Fig. 9 shows on different BLOCKS WORLD problems the value function size, computed as the number of leaves in $Tree(V)$ after 50 episodes.

Table 5 gives the rate of impossible states derived from the number of states shown in labels in Fig. 9 and the time required to perform one learning step by SPITI and IMPSPITI respectively. IMPSPITI uses trees to represent impossible states (the time with the *ad hoc* rules described in section 4.2 is indicated between parentheses).

Fig. 10 shows the convergence speed in number of episodes for the BLOCKS WORLD of size 4-3-4, that is a representative middle size problem. IMPSPITI takes less episodes

Table 4. MAZE6 performance (cost in memory and time)

	Value size	Policy size	Time/step(sec)
SPITI det	214 ± 10	241 ± 4	1.7 ± 0.8
SPITI stoc	252 ± 14	240 ± 12	3.7 ± 1.2
IMPSPITI det	35 ± 2	35 ± 2	0.1 ± 0.08
IMPSPITI stoc	35 ± 2	35 ± 2	0.24 ± 0.1

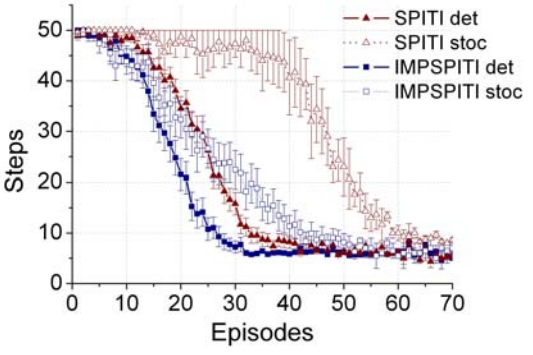
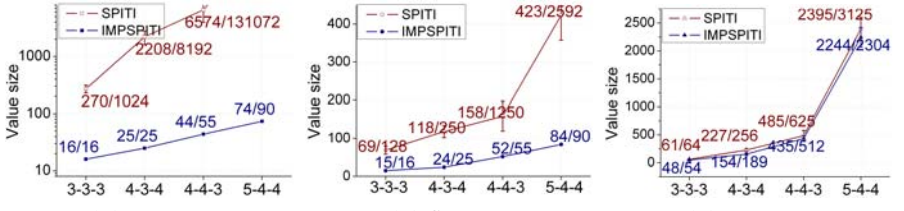
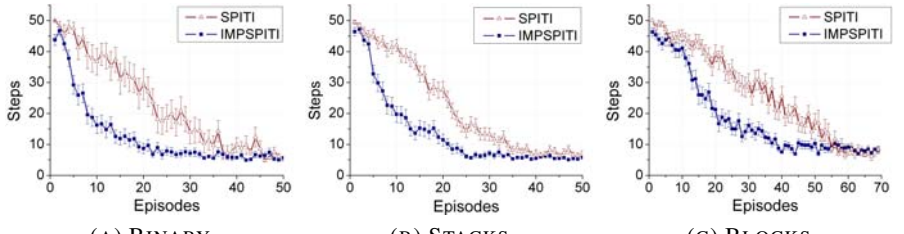


Fig. 8. Convergence over episodes on MAZE6



(A) BINARY (B) STACKS (C) BLOCKS

Fig. 9. BLOCKS WORLD: value function size as a function of the size of the problem. Labels on points indicate this value / the total number of states considered. Note the log scale for *Binary*.



(A) BINARY (B) STACKS (C) BLOCKS

Fig. 10. BLOCKS WORLD performance along episodes (size 4-3-4). We run 70 episodes with *Blocks* to wait for convergence.

than SPITI to reach the optimal policy with all representations. This result is explained by the fact that IMPSPITI uses smaller trees with a simpler structure, therefore it takes less steps to propagate values over the trees. But note that, as expected, the difference is smaller when the rate of impossible states is smaller. Note also that, even when there are very few impossible states as is the case with the *Blocks* representation, the policy improves faster with IMPSPITI.

Now, comparing the performance in time on a single step from Table 5, in *Binary* representation, where the rate of impossible states grows very fast, IMPSPITI

Table 5. BLOCKS WORLD: rate of impossible states and time to perform one step (in seconds). A "-" indicates that the value could not be obtained after three days of computation.

		BINARY		STACKS		BLOCKS	
B-S-Y		SPITI	IMPSPITI	SPITI	IMPSPITI	SPITI	IMPSPITI
3-3-3	% IMP.	98.5%		87.5%		15.6%	
	TIME	0.28 ±0.03	0.04 ± 0.01	0.05 ±0.02	0.01 (0.01)	0.04	0.03 (0.03)
4-3-4	% IMP.	99.7%		90%		26.1%	
	TIME	2.9 ± 0.4	0.08 ± 0.01	0.06 ±0.02	0.01 (0.01)	0.17 ± 0.02	0.18 ± 0.01 (0.13 ± 0.01)
4-4-3	% IMP.	>99.99%		95.6%		18%	
	TIME	34 ± 7	1.2 ± 0.2	0.13 ± 0.06	0.02 (0.02)	0.34 ± 0.2	0.9± 0.07 (0.5 ± 0.03)
5-4-4	% IMP.	>99.99%		96.5%		26.3%	
	TIME	-	2.6 ± 0.3	0.2 ± 0.09	0.05 (0.04)	2.6 ± 0.3	6.5 ± 0.9 (3.1 ± 0.2)

unquestionably outperforms SPITI in time and memory use. With the *Stacks* representation, both the size of the problem and the rate of impossible states grows slower, thus the time difference between SPITI and IMPSPITI keeps small. Finally, the rate of impossible states is much smaller in the *Blocks* representation, therefore both algorithms perform similarly for small BLOCKS WORLDs and SPITI takes less time per step than IMPSPITI as the problem size grows, due to the time overhead required to check for the existence of states. This is also true using *ad hoc* rules to detect impossible states, even if the overhead is smaller in that case.

5 Discussion

The first message of this paper is that, although using a factored representation in reinforcement learning results in the possibility to address larger problems, designing a factored representation so that it does not artificially increase the number of considered states may be very difficult. Consider the MAZE6 problem, where there are only 37 states, but a somewhat standard factored representation may result in 6561 potential states. This problem is an idealization of a standard robot navigation problem where, given usual robot sensors, one could not say in advance which sensory values cannot occur simultaneously. Similarly, if we take the BLOCKS WORLD problem, it proved difficult to design a representation that would fit the number of actual states. The *Blocks* representation results in fewer impossible states, but at the price of some redundancy that makes it very inefficient for larger problems (for instance, with a size 6-5-5 problem, we have 34375 possible states represented whereas there are only 334 actual states). We have shown that IMPSPITI was able to stick to the number of possible states of the problem given a representation, resulting in the possibility to address a much wider class of FMDPs than standard SDP methods.

Our second message is about the time overhead resulting from the necessity to check whether a state is possible. Considering the computation time per step and the number of steps required to converge, IMPSPITI always performs faster if there are enough impossible states. More precisely, as illustrated with the *Blocks* representation, the larger the problem, the larger the necessary rate of impossible states, since the tree of possible states will grow larger, resulting in a large overhead. However, using domain specific

rules to detect impossible states generally results in a further gain in speed, but it is at the price of generality.

Finally, the fact that the policy improves faster with IMPSPITI than with SPITI even when there are very few impossible states, as is the case with the *Blocks* representation, tends to indicate that considering states as impossible until they are seen is an efficient heuristics even in the absence of actually impossible states. This heuristics itself will deserve further analyses. In particular, it would be of much interest to study the potential interactions between this *pessimistic* heuristics and using efficient optimistic exploration strategies such as "Optimism in the Face of Uncertainty" (Szita & Lőrincz, 2008) that drives the agent towards unseen states. At first glance, these heuristics are contradictory, since in the former we do not want to represent impossible states which we do not distinguish from unseen states, whereas in the latter we want to attribute a large value to unseen states, thus we need to represent them.

6 Conclusion

We have shown that there exists a practically relevant class of FMDPs between the "no synchronic arcs" and the "any synchronic arcs" classes that corresponds to problems where some combinations of state variable values do not occur. Though standard SDP algorithms such as SVI and their derivatives such as SPITI can be applied to this class of problems, we have shown that modifying the algorithm to take the presence of impossible states into account can result in significant performance improvements. Moreover, we have shown that, in the context where an agent has to explore its environment to learn its structure, considering as impossible the states that have not yet been encountered is also beneficial to the performance.

Note that the approach described in section 3 can be applied to other standard SDP algorithms. Here, we focused on one particular FRL method, namely SPITI, but the impact on other instances of SDYNA using SPUDD or Guestrin's linear programming approach (Guestrin et al., 2003) remains to be studied. Furthermore, we want to compare IMPSPITI with XACS (Butz et al., 2002), an Anticipatory Learning Classifier System endowed with most FRL systems properties, but which uses genetic algorithm heuristics instead of SDP and incremental tree induction methods. A previous comparison between XACS and SPITI (Sigaud et al., 2009) has shown that XACS can deal efficiently with impossible states, but a closer comparison between the mechanisms of IMPSPITI presented here and those of XACS remains to be performed.

References

Boutilier, C.: Correlated action effects in decision theoretic regression. In: Proceedings of the 13th International Conference on Uncertainty in Artificial Intelligence, pp. 30–37. AUAI Press (1997)

Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, pp. 1104–1111 (1995)

Boutilier, C., Dearden, R., Goldszmidt, M.: Stochastic dynamic programming with factored representations. Artificial Intelligence 121, 49–100 (2000)

Butz, M.V., Goldberg, D.E., Stolzmann, W.: The Anticipatory Classifier System and Genetic Generalization. Natural Computing 1, 427–467 (2002)

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

Degris, T., Sigaud, O., Wuillemin, P.-H.: Chi-square tests driven method for learning the structure of factored MDPs. In: Proceedings of the 22nd International Conference on Uncertainty in Artificial Intelligence, Massachusetts Institute of Technology, pp. 122–129. AUAI Press, Cambridge (2006a)

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

Guestrin, C., Koller, D., Parr, R., Venkataraman, S.: Efficient Solution Algorithms for Factored MDPs. Journal of Artificial Intelligence Research 19, 399–468 (2003)

Hoey, J., St-Aubin, R., Hu, A., Boutilier, C.: SPUDD: Stochastic planning using decision diagrams. In: Proceedings of the 15th Conference on International Conference on Uncertainty in Artificial Intelligence, Stockholm, pp. 279–288 (1999)

Sigaud, O., Butz, M.V., Kozlova, O., Meyer, C.: Anticipatory Learning Classifier Systems and Factored Reinforcement Learning. In: ABiALS 2008. LNCS (LNAI), vol. 5499. Springer. Heidelberg (2009)

Sigaud, O., Wilson, S.W.: Learning Classifier Systems: a survey. Journal of Soft Computing 11, 1065–1078 (2007)

Sutton, R.S.: DYNA, an integrated architecture for learning, planning and reacting. In: Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures (1991)

Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press, Cambridge (1998)

Szita, I., Lőrincz, A.: The many faces of optimism: a unifying approach. In: ICML 2008: Proceedings of the 25th international conference on Machine learning, pp. 1048–1055. ACM Press, New York (2008)

Utgoff, P.E.: Incremental induction of decision trees. Machine Learning 4, 161–186 (1989)