very short solutions: for example, the solution of the halting problem is a single bit, but finding it is undecidable. Therefore, proving that the solutions of a problem cannot be represented in polynomial space cannot be done directly from complexity results.

The result we prove affects the choice of how policies are generated and executed. There are two ways of planning in a scenario that is formalizable by an MDP: the first one is to determine all actions to execute at once (that is, determine the whole policy); the second one is to only determine the first action, execute it, check the resulting state, and then find the next action, etc. In the second solution, the process of finding the action to execute has to be done at each time step; this process is as hard as finding the optimal reward. The advantage of the first solution is that, once a polynomial sized factored representation of an optimal policy is known, finding the action to execute in a state is polynomial. This can be seen as a form of preprocessing of the problem: the policy is determined from the MDP only, and it then makes the problem of determining the action to execute in a specific state polynomial. The proof we give is actually a proof of unfeasibility of such preprocessing algorithms, and it therefore applies to any factored form that allows determining the action to execute in polynomial time.

Preliminaries

The main parts of a Markov Decision Process (MDP) are a set of operators (actions), whose effect may be stochastic, and a function that evaluates states according to a notion of goodness.

Formally, an MDP is a 5-tuple $\mathcal{M} = \langle \mathcal{S}, s_0, \mathcal{A}, t, r \rangle$, where $\mathcal{S}$ is a set of states, s_0 is a distinguished state (the initial state), and $\mathcal{A}$ is a set of actions. The functions t and r represent the effects of the actions and the reward associated to states, respectively.

We consider factored representations of MDPs: states are propositional interpretations over a given alphabet of binary variables (state variables). The set S is therefore implicit in the set of state variables. Thus, we do not count it in the size of an MDP. The set of state variables is assumed to be finite, and S is therefore finite as well.

The result of an action is not assumed to be known for sure, but only according to some probability distribution. Therefore, we cannot represent the effects of actions using a function that maps a state and an action into another state. t is instead a function from actions and pairs of states to numbers in the interval $[0, 1]$. It represents the probability of transitions: $t(s_1, s_2, a) = p$ means that the result of action a in state s_1 is the state s_2 with probability p.

The reward function is a function from states to integer numbers, formalizing how much a state matches our goals. In this paper, we assume that the functions t and r are represented by boolean circuits. This representation of MDPs subsumes other factored representations (Boutilier, Dean, & Hanks 1999; Mundhenk *et al.* 2000).

Planning in deterministic domains consists in determining a sequence of actions that allows reaching a goal state; in nondeterministic domains we have two points to take into account: first, the extent to which the goal is reached can only be probabilistically determined; second, the action to execute in a time point cannot be uniquely determined from the initial state and the actions executed so far.

Namely, the aim of planning in a nondeterministic domain is that of determining a set of actions to execute that result in the best possible states (according to the reward function). Since the result of actions is not known for sure, we can only determine an average reward. For example, if the result of applying a in the state s_0 is s_1 with probability 1/3, and s_2 with probability 2/3, then the expected reward of executing a is given by $1/3 \cdot r(s_1) + 2/3 \cdot r(s_2)$, where $r(s_1)$ and $r(s_2)$ are the rewards of s_1 and s_2, respectively. Formally, we use the average undiscounted reward, i.e., we evaluate the average reward for all states, weighted by the probability of reaching them.

The second effect of nondeterminism is that the best action to execute depends on the current state, which is not uniquely determined from the initial state and the actions executed so far, as the effect of actions are only probabilistically known. For example, executing a may result in state s_1 or in state s_2. After a is executed, we know which one is the actual result. At this point, it may be that the best action to execute in s_1 is a', while the best choice in s_2 is a''. The best action depends on the current state: in the simplest case, a policy is a function that gives the best action to execute in each state.

The reward associated to a policy is the expected average reward obtained by executing, in each state, the associated action. We assume a finite horizon, i.e., we consider only what happens up to a given number of steps. We also assume that this number is polynomial in the total size of the instance, where the size of the instance is the size of $\mathcal{M} = \langle \mathcal{S}, s_0, \mathcal{A}, t, r \rangle$ minus the size of S which is implicitly represented. Technically, this means that the unary representation of the horizon is part of the instance.

More complex forms of policies are possible. Indeed, we may decide what to do using not only the current state, but also the sequence of actions executed so far. Such policies can be represented as trees, in which each node is labeled with an action, and its children are associated to the possible outcoming states. These policies are called *history-dependent policies*. Policies depending only on the current state form a subclass of them, and are called *stationary policies*.

A problem of interest is whether a policy can be represented in polynomial space. The trivial representation of a policy (using a tree) may be exponentially larger than the MDP even if only one action has more than one possible outcome. However, policies may take less space in factored form. We represent policies using circuits: the input is the binary representation of the current state and of the history; the output is the next action to execute. Some policies are much smaller, in this representation: for example, the policy of executing the same action regardless of the state and the history is polynomial, even if it would require an exponentially large tree.

The question we consider in this paper is whether it is always possible to represent the optimal policy of an MDP

with a polynomial circuit. We show that a positive answer would have a consequence considered unlikely in complexity theory: the polynomial hierarchy coincide with the complexity class Π_2^p. The proof is based on the fact that the existence of such circuits would allow solving the problem of determining the action to execute in each state in two steps: first, determine the whole circuit using the MDP alone; then, use the circuit and the current state to determine the next action to execute in polynomial time. This is a form of algorithm with preprocessing, in which a first step only works on part of the instance (the MDP alone, without the current state), and makes the solving of the rest of the problem easier. The unfeasibility of this schema implies the impossibility of always representing optimal policies with polynomial circuits. To this end, we use compilability classes and reductions, which we briefly present in the next section.

Complexity and Compilability

We assume the reader is familiar with the basic complexity classes such as P, NP, and the classes of the polynomial hierarchy (Stockmeyer 1976; Garey & Johnson 1979). In the sequel, C, C$'$, etc. denote arbitrary classes of the polynomial hierarchy. We assume that the input instances of problems are strings over an alphabet Σ. The *length* of a string $x \in \Sigma^*$ is denoted by $||x||$.

We summarize some definitions and results about complexity of preprocessing (Cadoli *et al.* 2002; Gogic *et al.* 1995). We consider problems whose instances can be divided into two parts; one part is *fixed* (known in advance), and the second one is *varying* (known when the solution is needed.) The problem of determining the action to execute in a state has this structure: the MDP is the part known in advance, as it is known from the description of the domain; the state is instead only determined once the previous actions have been executed. Compilability classes and reductions formalize the complexity of such problems when the first input can be preprocessed.

A function f is called *poly-size* if there exists a polynomial p such that, for all strings x, it holds $||f(x)|| \leq p(||x||)$. When x represents a natural number we instead impose $||f(x)|| \leq p(x)$. A function g is called *poly-time* if there exists a polynomial q such that, for all x, $g(x)$ can be computed in time less than or equal to $q(||x||)$. These definitions extend to binary functions in the obvious manner.

We define a *language of pairs* S as a subset of $\Sigma^* \times \Sigma^*$. We define a hierarchy of classes of languages of pairs, the *non-uniform compilability classes*, denoted as $\|\!\leadsto$C, where C is a generic uniform complexity class, such as P, NP, coNP, or Σ_2^p.

Definition 1 ($\|\!\leadsto$C classes) *A language of pairs $S \subseteq \Sigma^* \times \Sigma^*$ belongs to $\|\!\leadsto$C iff there exists a binary poly-size function f and a language of pairs $S' \in$ C such that, for all $\langle x, y\rangle \in S$, it holds:*

$$\langle x, y\rangle \in S \text{ iff } \langle f(x, ||y||), y\rangle \in S'$$

$\|\!\leadsto$C contains problems that are in C after a suitable polynomial-size preprocessing step. Clearly, any problem in C is also in $\|\!\leadsto$C. Compilation is useful if a problem in C is in $\|\!\leadsto$C$'$, where C'⊂C, that is, preprocessing decreases the complexity of the problem. There are problems for which such reduction of complexity is possible (Cadoli *et al.* 2002).

For these classes it is possible to define the notions of *hardness* and *completeness*, based on a suitable definition of reductions.

Definition 2 (nucomp reductions) *A nucomp reduction from a problem A to a problem B is a triple $\langle f_1, f_2, g\rangle$, where f_1 and f_2 are poly-size functions, g is a polynomial function, and for every pair $\langle x, y\rangle$ it holds that $\langle x, y\rangle \in A$ if and only if $\langle f_1(x, ||y||), g(f_2(x, ||y||), y)\rangle \in B$.*

Definition 3 ($\|\!\leadsto$C-completeness) *Let S be a language of pairs and C a complexity class. S is $\|\!\leadsto$C-hard iff for all problems $A \in \|\!\leadsto$C there exists a nucomp reduction from A to S. If S is also in $\|\!\leadsto$C, it is called $\|\!\leadsto$C-complete.*

The hierarchy formed by the compilability classes is proper if and only if the polynomial hierarchy is proper (Cadoli *et al.* 2002; Karp & Lipton 1980; Yap 1983) — a fact widely conjectured to be true.

Informally, $\|\!\leadsto$NP-hard problems are "not compilable to P". Indeed, if such compilation were possible, then it would be possible to define f as the function that takes the fixed part of the problem and gives the result of compilation (ignoring the size of the input), and S' as the language representing the on-line processing. This would implies that a $\|\!\leadsto$NP-hard problem is in $\|\!\leadsto$P, and this implies the collapse of the polynomial hierarchy.

$\|\!\leadsto$NP-hardness can be proved as follows: let us assume that $\langle r, h\rangle$ is a reduction from sat to a problem of pairs S, that is, Π is satisfiable if and only if $\langle r(\Pi), h(\Pi)\rangle \in S$. This implies that S is NP-hard, but tells nothing about hardness w.r.t. compilability classes. However, if the additional property of monotonicity holds, then S is also $\|\!\leadsto$NP-hard (Liberatore 2001). The pair $\langle r, h\rangle$ is a monotonic polynomial reduction if, for any pair of clauses Π_1 and Π_2 over the same literals, with $\Pi_1 \subseteq \Pi_2$, it holds:

$$\langle r(\Pi_1), h(\Pi_1)\rangle \in S \text{ iff } \langle r(\Pi_2), h(\Pi_1)\rangle \in S$$

Note that the second instance combines a part from Π_2 and a part from Π_1: this is not a typo. A problem is $\|\!\leadsto$NP-hard, if there exists a polynomial reduction from sat to it that can be proved to be monotonic. The definition of monotonicity will be more clear when applied to a specific problem, as we do in the next section.

Finding Policies Cannot Be Compiled

The impossibility of representing policies in polynomial space is proved using compilability classes as follows: suppose that we are able to find a circuit that: a. is of polynomial size; and b. represents an optimal policy. Such a circuit allows for deciding which action to execute in a state in polynomial time. We have therefore an algorithm for solving the problem of determining the optimal action in a state: in a first (possibly long) phase the circuit is found; a second, polynomial, phase finds the action to execute. The first phase only works on a part of the problem data, and produces a polynomial sized data structure that makes the whole

problem polynomial: the problem is in $\|\!\!\leadsto$P. Polynomiality of policy size is then equivalent to the question: is the problem of determining the action to execute in a state in $\|\!\!\leadsto$P? Proving that the problem is $\|\!\!\leadsto$NP-hard gives a negative answer.

Formally, we consider the following problem: given an MDP $\mathcal{M}$, one of its states s, and one of its actions a, decide whether a is the action to execute in s according to some optimal policy. We prove this problem to be $\|\!\!\leadsto$NP-hard. This is done by first showing it NP-hard, and then proving that the employed reduction is monotonic.

Let Π be a set of clauses, each composed of three literals, over a set of variables $X = \{x_1, \ldots, x_n\}$. The reduction is as follows: given a set of clauses Π, we build the triple $\langle \mathcal{M}, s, a \rangle$, where the first element is an MDP, the second is one of its states, and the third is one of its actions.

Let L be the set of literals over X, and let $E = L \cup \{\mathsf{sat}, \mathsf{unsat}\}$. The MDP $\mathcal{M}$ is defined as:

$$\mathcal{M} = \langle \mathcal{S}, \epsilon, \mathcal{A}, t, r \rangle$$

The components of $\mathcal{M}$ are defined as follows.

States: $\mathcal{S}$ is the set of sequences of at most $(2n)^3 + n + 1$ elements of E;

Initial state: is the empty sequence ϵ;

Actions: $\mathcal{A}$ contains three actions A, S, and U, and one action a_i for each x_i;

Transition function: the effect of A is to randomly select (with equal probability) a literal of L and adding it to the sequence representing the current state; the effect of S and U is to add sat and unsat to the sequence, respectively (these are deterministic actions); the effect of a_i is to add either x_i or $\neg x_i$ to the sequence, with the same probability;

Reward function: This is the most involved part of the MDP. Given a sequence of $3m$ literals of L, we define a $\mathsf{3cnf}$ formula as follows:

$$C(l_1^1, l_2^1, l_3^1, \ldots, l_1^m, l_2^m, l_3^m) = \{l_1^1 \vee l_2^1 \vee l_3^1, \ldots, l_1^m \vee l_2^m \vee l_3^m\}$$

Since the number of different clauses over L is less than $(2n)^3$, any set of clauses can be represented as a sequence of $3m$ literals, where $m = (2n)^3$. This is a way to encode all sets of clauses over L as sequences of literals.

The reward function is defined in terms of C: a sequence of $3m$ literals followed by unsat has reward 1; a sequence s of $3m$ literals followed by sat, further followed by a sequence $s' = l_1, \ldots, l_n$ (where l_i is either x_i or $\neg x_i$) has reward $2 \cdot 2^n$ if $C(s)$ is satisfied by s', otherwise 0; all other sequences have reward 0.

Note that most of the states have reward 0. While the total reward is an average over all reached states, r is defined in such a way all states preceeding and succeeding a nonzero reward state have reward zero. r is defined this way for the sake of making the proof simpler; however, we are still calculating the total reward over all states, including "intermediate" ones.

This MDP has a single optimal policy: execute A for $3m$ times, then execute either U or the sequence $S, a_1, \ldots, a_n$, depending on the result of the execution of A. Namely, any possible result of the execution of the first $3m$ actions corresponds to a set of clauses. The next action of the optimal policy is U if the set is unsatisfiable and S if it is satisfiable.

This is the definition of the MDP $\mathcal{M}$. The instance of the problem is composed of an MDP, a state, and an action, and the problem is to check whether the action is optimal in the state. The action is S, and the state s is the sequence of literals such that $C(s) = \Pi$. We can now prove that f is a reduction from satisfiability to the problem of the next action.

Theorem 1 *The MDP $\mathcal{M}$ has an unique optimal policy. The set of clauses Π is satisfiable if and only if S is the action to execute from s in the optimal policy of $\mathcal{M}$.*

Proof. The MDP has policies with positive reward: the sequence of actions $A^{3m}U$ has a reward equal to 1, since all its leaves have reward 1 and all internal nodes have reward 0. Note that, since the actions executed so far can be derived from the current state, stationary and history-based policies coincide.

The policies with positive reward are very similar to each other. Indeed, they all begin by executing $3m$ times the action A. Then, either U or the sequence of $S, a_1, \ldots, a_n$ is executed. Policies with positive reward can only differ on this choice of executing U or $S, a_1, \ldots, a_n$. However, a policy can take the first choice in a state, and the other one in another state: policies are not forced to make the same choice regardless of the current state.

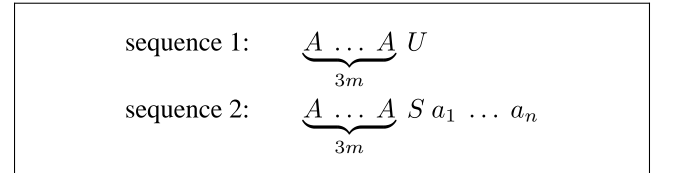

Figure 1: Sequences that can generate a reward > 0;
All their fragments and extensions have reward 0.

Let us now consider the state after the execution of A for $3m$ times. Since each execution of A generates a random literal, at this point the state is a sequence of $3m$ literals. Such a sequence represents the set of clauses that is later used by the reward function. Intuitively, at this point we want the optimal policy to execute U if the set of clauses is unsatisfiable, and $S, a_1, \ldots, a_n$ if it is satisfiable. This is obtained by giving reward 1 to the subtree composed by U alone, and reward equal to double the number of models of the set of clauses to the subtree $S, a_1, \ldots, a_n$. The optimal choice will then be U if the formula is unsatisfiable (reward 1, instead of 0), and $S, a_1, \ldots, a_n$ if it is satisfiable (reward ≥ 2, instead of 1).

Summarizing, the first $3m$ actions generates a random set of clauses, while the sequence $a_1, \ldots, a_n$ generates a random interpretation. The leaves in which the interpretation satisfies the set of clauses have reward $2 \cdot 2^n$, while the other

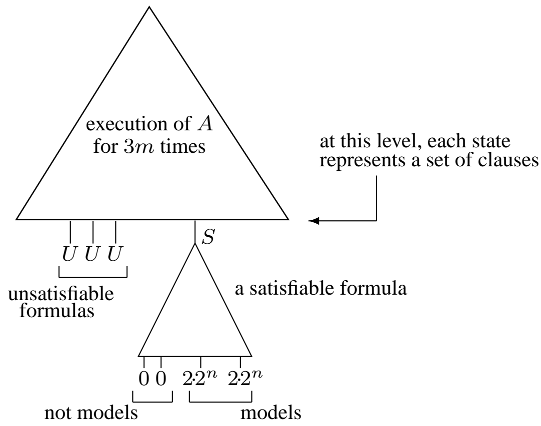

Figure 2: The optimal policy of the MDP of the proof.

ones have reward 0. If the formula is unsatisfiable then U is the best choice, as it gets a reward 1 instead of 0. □

This theorem implies that choosing the next action is an NP-hard problem, a result of really little significance by itself, in light of the PSPACE-hardness of related problems. However, the reduction can be used to prove that the problem of choosing the next action cannot be simplified to P thanks to a polynomial data structure depending only on the MDP. This, in turn, implies the nonexistence of a polynomial sized circuit representing the optimal policy.

In particular, a polynomial reduction from 3sat to a problem that satisfies the condition of monotonicity (Liberatore 2001) implies that the problem is $\|\!\leadsto$NP-hard, and therefore cannot be compiled. Since the problem instances can be divided into two parts, the reduction itself can be decomposed into two separate functions, one generating the part of the instance that can be compiled, and the other generating the rest of the instance. In our case, $\mathcal{M}$ is the part that can be compiled, while s and S are the rest of the instance. As a result, if $\langle \mathcal{M}, s, S \rangle$ is the MDP that corresponds to a set of clauses Π, the two functions are defined as:

$$\begin{aligned} r(\Pi) &= \mathcal{M} \\ h(\Pi) &= \langle s, S \rangle \end{aligned}$$

Monotonicity holds if, for every pairs of sets of clauses Π_1 and Π_2 over the same set of literals, with $\Pi_1 \subseteq \Pi_2$ it holds that $\langle r(\Pi_1), h(\Pi_1) \rangle$ is a "yes" instance if and only if $\langle r(\Pi_2), h(\Pi_1) \rangle$ is a "yes" instance. Note that the second instance is $\langle r(\Pi_2), h(\Pi_1) \rangle$, that is, it combines a part derived from Π_2 and a part from Π_1.

Instead of trying to explain this definition better, we consider its specialization to the case of MDPs. Let $\mathcal{M}_1 = r(\Pi_1)$ and $\mathcal{M}_2 = r(\Pi_2)$ be the MDPs corresponding to the sets of clauses Π_1 and Π_2 using the construction above. Let $\langle s, S \rangle = h(\Pi_1)$. Monotonicity can be expressed as: for any two sets of clauses Π_1 and Π_2 over the same set of literals, with $\Pi_1 \subseteq \Pi_2$, it must hold that S is the optimal action to execute in the state s for the MDP $\mathcal{M}_1$ if and only if it is so in the MDP $\mathcal{M}_2$. The reduction we have defined satisfies this condition.

Theorem 2 *The reduction $\langle r, h \rangle$ is a monotonic polynomial reduction.*

Proof. Let Π_1 and Π_2 be two sets of clauses over the same set of variables, and let $\mathcal{M}_1$ and $\mathcal{M}_2$ be their corresponding MDPs. Since the MDP corresponding to a set of clauses depends—by construction—on the number of variables only, $\mathcal{M}_1$ and $\mathcal{M}_2$ are exactly the same MDP. As a result, for any state s and action S, the latter is the optimal action to execute in $\mathcal{M}_1$ if and only if it is so for $\mathcal{M}_2$, and this is the definition of monotonicity for the case of MDPs. □

This theorem implies that the problem of the next action is hard for the compilability class $\|\!\leadsto$NP. In turns, this result implies that there is no polynomial-sized representation of a policy that allows determining the next action in polynomial time.

Theorem 3 *If, for any MDP, there exists a polynomial data structure that allows determining the next action to execute according to an optimal policy in polynomial time, then Σ_2^p=Π_2^p=PH.*

Proof. If such representation exists, the problem of the next action is in $\|\!\leadsto$P. Indeed, given the fixed part of the problem (the MDP), we can get such polynomial representation of the optimal policy. Then, determining the next action can be done in polynomial time. This implies Σ_2^p=Π_2^p=PH (Cadoli *et al.* 2002). □

The circuit representation of policies is a subcase of data structures allowing the determination of the next state in polynomial time.

Corollary 1 *If, for any MDP, there exists a polynomial circuit representing an optimal policy, then Σ_2^p=Π_2^p=PH.*

Conclusions

Optimal policies of MDPs cannot always be represented by polynomial-sized circuits. We proved this claim using compilability classes; namely, it is implied by the fact that there is no polynomial data structure that allows determining the next action to execute in a state in polynomial time; circuits are a special case of such structures. This implies that, in a probabilistic domain, we cannot determine an optimal policy all at once, and then execute it. What we can do is either to determine only the first optimal action to execute (and then repeating once the resulting state is known), or to use a suboptimal policy that can be represented in polynomial space.

Let us now discuss how the results presented in this paper relate to similar one in the literature. As already remarked in the Introduction, complexity does not necessarily imply that policies cannot be compactly represented. Namely, even a result of undecidability does not forbid compactness of policies. As a result, such nonexistence results are not implied by complexity results.

On the other hand, a result of non-polynomiality of the size of policies of POMDPs already appeared in the literature, namely, in Papadimitriou and Tsitsiklis' paper (Papadimitriou & Tsitsiklis 1987). Their result does not imply ours, as it holds for POMDPs in the explicit representation with only nonpositive rewards; the same problem

that is PSPACE-hard in their formalization is polynomial in ours (Mundhenk *et al.* 2000). To be precise, the two results cannot be derived from each other. The technique used by Papadimitriou and Tsitsiklis is also different from ours, but can be nonetheless applied in our settings, and allows for proving that the existence of a short policy implies that PSPACE is contained in $\mathrm{NP}^{\mathrm{PP}}$; technical details can be found in a technical report (Liberatore 2002).

We conclude the paper by observing that all our results holds for POMDPs, since MDPs are special cases of POMDPs in which everything is observable.

Acknowledgments

Many thanks to the anonymous referees, who helped the author to improve the quality of the paper.

References

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

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

Cadoli, M.; Donini, F. M.; Liberatore, P.; and Schaerf, M. 2002. Preprocessing of intractable problems. *Information and Computation*. To Appear.

Cassandra, A.; Littman, M.; and Zhang, N. 1997. Incremental pruning: a simple, fast, exact method for partially observable markov decision processes. In *Proc. of UAI-97*.

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

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

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, Ca: W.H. Freeman and Company.

Gogic, G.; Kautz, H. A.; Papadimitriou, C.; and Selman, B. 1995. The comparative linguistics of knowledge representation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, 862–869.

Hansen, E., and Feng, Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Proc. of AIPS-00*, 130–139.

Karp, R. M., and Lipton, R. J. 1980. Some connections between non-uniform and uniform complexity classes. In *Proceedings of the Twelfth ACM Symposium on Theory of Computing (STOC'80)*, 302–309.

Koller, D., and Parr, D. 1999. Computing factored value functions for policies in structured MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, 1332–1339.

Liberatore, P. 2001. Monotonic reductions, representative equivalence, and compilation of intractable problems. *Journal of the ACM* 48(6):1091–1125.

Liberatore, P. 2002. On polynomial sized MDP factored policies. Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza".

Littman, M.; Dean, T.; and Kaebling, L. 1995. On the complexity of solving markov decision processes. In *Proc. of UAI-95*.

Littman, M. 1997. Probabilistic propositional planning: representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, 748–754.

Madani, O.; Hanks, S.; and Condon, A. 1999. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, 541–548.

Mundhenk, M.; Goldsmith, J.; Lusena, C.; and Allender, E. 2000. Complexity of finite-horizon markov decision processes problems. *Journal of the ACM* 47(4):681–720.

Papadimitriou, C., and Tsitsiklis, J. 1987. The complexity of markov decision processes. *Mathematics of Operations Research* 12(3):441–450.

Stockmeyer, L. J. 1976. The polynomial-time hierarchy. *Theoretical Computer Science* 3:1–22.

Yap, C. K. 1983. Some consequences of non-uniform conditions on uniform classes. *Theoretical Computer Science* 26:287–300.

Zhang, N., and Zhang, W. 2001. Speeding up the convergence of value iteration in partially observable markov decision processes. *Journal of Artificial Intelligence Research* 14:29–51.