Approximate Solution Techniques for *Factored* First-order MDPs

Scott Sanner
Department of Computer Science
University of Toronto
Toronto, ON M5S 3H5, CANADA
ssanner@cs.toronto.edu

Craig Boutilier
Department of Computer Science
University of Toronto
Toronto, ON M5S 3H5, CANADA
cebly@cs.toronto.edu

Abstract

Most traditional approaches to probabilistic planning in relationally specified MDPs rely on grounding the problem w.r.t. specific domain instantiations, thereby incurring a combinatorial blowup in the representation. An alternative approach is to lift a relational MDP to a first-order MDP (FOMDP) specification whose solution approaches avoid grounding. Unfortunately, state-of-the-art FOMDPs are inadequate for specifying factored transition models or additive rewards that scale with the domain size – structure that is very natural in probabilistic planning problems. To remedy these deficiencies, we propose an extension of the FOMDP formalism known as a *factored* FOMDP and present generalizations of symbolic dynamic programming and linear-value approximation solutions to exploit factored FOMDP structure. Along the way, we also make important contributions to the field of first-order probabilistic inference (FOPI) by demonstrating novel first-order structures that can be exploited without domain grounding. We present empirical results to demonstrate that we can obtain effective solutions whose complexity scales polynomially in the logarithm of the domain size – novel results that are impossible to obtain with *any* previously proposed solution method.

Introduction

There has been a great deal of research in recent years aimed at exploiting structure in order to compactly represent and efficiently solve decision-theoretic planning problems in the Markov decision process (MDP) framework [1]. While traditional approaches to solving MDPs typically used an enumerated state and action model, this approach has proved impractical for large-scale AI planning tasks where the number of distinct states in a model can easily exceed the limits of primary and secondary storage on modern computers.

Fortunately, many MDPs can be compactly described in a propositionally factored model that exploits various independences in the reward and transition functions. And not only can this independence be exploited in the problem representation, it can often be exploited in exact and approximate solution methods as well [9; 18; 8]. However, while recent techniques for factored MDPs have proven effective, they cannot generally exploit first-order structure. Yet many realistic planning domains are best represented in first-order terms, exploiting the existence of domain objects, relations over these objects, and the ability to express objectives and action effects using quantification.

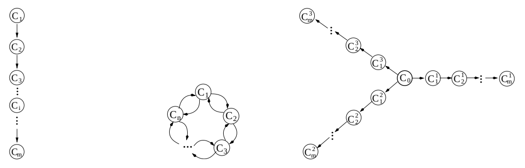
These deficiencies have motivated the development of the first-order MDP (FOMDP) framework [2] that directly exploits the first-order representation of MDPs to obtain domain-independent solutions. While FOMDP approaches have demonstrated much promise [12; 11; 15; 16], current formalisms are inadequate for specifying both exogenous actions and additive rewards that *scale with the domain size*.

To remedy these deficiencies, we propose a novel extension of the FOMDP formalism known as a *factored* FOMDP. This representation introduces product and sum aggregator extensions to the FOMDP formalism that permit the specification of factored transition and additive reward models that scale with the domain size. We then generalize symbolic dynamic programming and linear-value approximation solutions to exploit product and sum aggregator structure, solving a number of novel problems in first-order probabilistic inference (FOPI) in the process. Having done this, we present empirical results to demonstrate that we can obtain effective solutions on the well-studied SYSADMIN problem whose complexity scales polynomially in the logarithm of the domain size – novel results that are impossible to obtain with *any* previously proposed solution method.

Markov Decision Processes

Factored Representation

In a factored MDP, states will be represented by vectors $\vec{x}$ of length n, where for simplicity we assume the state variables $x_1, \ldots, x_n$ are in $\{0,1\}$; hence the total number of states is $N = 2^n$. We also assume a set of actions $A = \{a_1, \ldots, a_n\}$. An MDP is defined by: (1) a state transition model $P(\vec{x}'|\vec{x},a)$ which specifies the probability of the next state $\vec{x}'$ given the current state $\vec{x}$ and action a; (2) a reward function $R(\vec{x},a)$ which specifies the immediate reward obtained by taking action a in state $\vec{x}$; and (3) a discount factor $\gamma,\ 0 \le \gamma < 1$. A policy $\pi(\vec{x})$ specifies which action a to take in each state $\vec{x}$. Our goal is to find a policy that maximizes the value function, defined using the infinite horizon, discounted reward criterion: $V^{\pi}(\vec{x}) = E_{\pi}[\sum_{t=0}^{\infty} \gamma^t \cdot r^t|\vec{x}]$,



(a) Line (b) Uni-Ring (c) Star

Figure 1: Diagrams of the three example SYSADMIN connection topologies that we will focus on in this document.

where r^t is the reward obtained at time t.

Many MDPs often have a natural structure that can be exploited in the form of a factored MDP [1]. For example, the transition function can be factored as a dynamic Bayes net (DBN) where the conditional probability table (CPT) $P(x_i'|\vec{x}_i, a)$ for each next state variable x_i' is only dependent upon the action a and its direct parents $\vec{x}_i$ in the DBN. Then the transition model can be compactly specified as $P(\vec{x}'|\vec{x}, a) = \prod_{i=1}^n P(x_i'|\vec{x}_i, a)$. Likewise, when the reward can be factored additively over m subsets of state variables $\vec{x}_1, \ldots, \vec{x}_m$, the resulting reward can be compactly expressed as $R(\vec{x}, a) = \sum_{r=1}^m R_i(\vec{x}_r, a)$.

We define a *backup* operator B^a for action a as follows:

$$B^a[V(\vec{x})] = \gamma \sum_{\vec{x}'} \prod_{i=1}^{n} P(x_i'|\vec{x}_i, a) V(\vec{x}') \tag{1}$$

If π^* denotes the optimal policy and V^* its value function, we have the fixed-point relationship $V^*(\vec{x}) = \max_{a \in A} \{\sum_{r=1}^m R_i(\vec{x}_r, a) + B^a[V^*(\vec{x})]\}$.

Example 1 (SYSADMIN Factored MDP). *In the SYSADMIN problem [8], we have n computers $c_1, \ldots, c_n$ connected via a directed graph topology (c.f. Figure 1). Let variable x_i denote whether computer c_i is up and running (1) or not (0). Let $Conn(c_j, c_i)$ to denote a connection from c_j to c_i. We have n actions: $reboot(c_1), \ldots, reboot(c_n)$. The CPTs in the transition DBN have the following form:*

$$P(x_i' = 1|\vec{x}_i, a) = \begin{cases} a = reboot(c_i) : 1 \\ a \neq reboot(c_i) : (0.05 + 0.9x_i) \\ \quad \cdot \frac{|\{x_j | j \neq i \wedge x_j = 1 \wedge Conn(c_j, c_i)\}| + 1}{|\{x_j | j \neq i \wedge Conn(c_j, c_i)\}| + 1} \end{cases}$$

If a computer is not rebooted then its probability of running in the next time step depends on its current status and the proportion of computers with incoming connections that are also currently running. The reward is the sum of computers that are running at any time step: $R(\vec{x}, a) = \sum_{i=1}^n x_i$. An optimal policy in this problem will reboot the computer that has the most impact on the expected future discounted reward given the network configuration.

Solution Methods

Value iteration is a simple *dynamic programming* algorithm for constructing optimal policies. It proceeds by constructing a series of t-stage-to-go value functions V^t. Setting $V^0(\vec{x}) = max_{a \in A} R(\vec{x}, a)$, we define

$$V^{t+1}(\vec{x}) = \max_{a \in A} \{R(\vec{x}, a) + B^a[V(\vec{x})]\} \tag{2}$$

The sequence of value functions V^t produced by value iteration converges linearly to the optimal value function V^*.

Approximate Linear Programming (ALP) is a technique for linearly approximating a value function. In a linear value function representation, we represent V as a linear combination of k basis functions $b_j(\vec{x}_j)$ over subsets of variables $\vec{x}_j$:

$$V(\vec{x}) = \sum_{j=1}^{k} w_j b_j(\vec{x}) \tag{3}$$

Our goal is to find weights that approximate the optimal value function as closely as possible. One way of doing this is to cast the optimization problem as a linear program (LP) that directly solves for the weights of an L_1-minimizing approximation of the optimal value function:

$$\begin{aligned} \text{Variables: } & w_1, \ldots, w_k \\ \text{Minimize: } & \sum_{\vec{x}} V(\vec{x}) \\ \text{Subject to: } & 0 \geq R(\vec{x}, a) + B^a[V(\vec{x})] - V(\vec{x}) \;;\; \forall a, \vec{x} \end{aligned} \tag{4}$$

Following notation in [17], we exploit the factored nature of the basis functions to simplify the objective to the following compact form: $\sum_{\vec{x}} V(\vec{x}) = \sum_{\vec{x}} \sum_{j=1}^k w_j b_j(\vec{x}) = \sum_{j=1}^k w_j y_j$ where $y^j = 2^{n - |\vec{x}_j|} \sum_{\vec{x}_j} b_j(\vec{x}_j)$. We also exploit the linearity property of the B^a operator to distribute it through the sum of basis functions to rewrite the constraints (note the $\max_{\vec{x}}$):

$$0 \geq \max_{\vec{x}} \left\{ \sum_{r=1}^{m} R_i(\vec{x}_r, a) + \sum_{j=1}^{k} w_j B^a[b_j(\vec{x})] - \sum_{j=1}^{j} w_j b_j(\vec{x})] \right\} \;;\; \forall a \tag{5}$$

This permits us to apply $B^a[\cdot]$ individually to each factored basis function and provides us with a factored $\max$-$\sum$ (i.e., cost network) form of the constraints. This factored form can then be exploited by LP constraint generation techniques [17] that iteratively solve the LP and add the maximally violated constraint on each iteration. Constraint generation is guaranteed to terminate in a finite number of steps with the optimal LP solution.

Factored First-order MDPs

Joint Actions and the Situation Calculus

The situation calculus [14] is a first-order language for axiomatizing dynamic worlds; in this work, we assume the logic is partially sorted (i.e., some, but not all variables will have sorts). Its basic ingredients consist of parameterized *action* terms, *situation* terms[1], and relational *fluents* whose truth values vary between states. A domain theory is axiomatized in the situation calculus; among the most important of these axioms are *successor state axioms (SSAs)*, which embody a solution to the frame problem for deterministic actions [14]. Previous work [2] provides an explanation

[1]In contrast to states, situations reflect the entire history of action occurrences. However, since the dynamics are Markovian, this allows recovery of state properties from situation terms.

of the situation calculus as it relates to first-order MDPs (FOMDPs). Here we simply provide a partial description of the SYSADMIN problem:

- **Sorts:** C (read: Computer); throughout the paper, we assume $n = |C|$ so the instances of this sort are $c_1, \ldots, c_n$.[2]
- **Situations:** s, $do(a, s)$ (read: the situation resulting from doing action a in situation s), etc...
- **Fluents:** $Up(c, s)$ (read: Computer c is running in situation s)

We have omitted a specification of actions and SSAs for SYSADMIN as we must first generalize the representation of the situation calculus to handle joint actions for factored FOMDPs. When the agent reboots a computer in SYSADMIN, the status of each computer may evolve independently of the other computers. Given n computers, each with 2 possible status configurations (up or not), this would lead to 2^n deterministic actions if *each* joint outcome were specified with a separate action. Clearly, such a representation is inefficient and ignores problem structure that can be exploited.

A more compact representation would be to specify deterministic *sub-actions* whose effects are independent of each other and which can be composed into a deterministic *joint* action. In general, a set of deterministic *sub-action* outcomes can be defined over one or more sort domains $C_1, \ldots, C_k$. For each $\vec{c} \in \{C_1 \times \cdots \times C_k\}$, we label the *set* of outcomes for each sub-action as $A(\vec{c})$. Then the set of all deterministic joint actions is given by $A = \{\times_{\vec{c} \in \{C_1 \times \cdots \times C_k\}} A(\vec{c})\}$.[3] To make this concrete, we now apply it to SYSADMIN:

- **Sub-action Sets:** $A(c) = \{upS(c), upF(c)\}$; $\forall c \in C$
- **Joint action Set:** $A = \{A(c_1) \times \cdots \times A(c_n)\}$

Here, $upS(c)$ and $upF(c)$ are actions indicating whether computer c should be up and running (success) or not (failure). We specify a joint deterministic action as the associative-commutative $\circ$ composition of its constituent sub-actions, e.g., if $n = 4$ then one possible joint deterministic action $a \in A$ could be $a = upS(c_1) \circ upF(c_2) \circ upF(c_3) \circ upS(c_4)$.

Finally, we define SSAs based on the joint deterministic actions in SYSADMIN. Since fluents will be affected by sub-actions, the SSAs must test whether a joint action contains a particular sub-action. We do this with the $\sqsupseteq$ predicate that tests whether the term on the LHS is a compositional superset of the RHS term. For example, given a above for the case of $n = 4$, we know that $a \sqsupseteq upS(c_1)$ is true, but $a \sqsupseteq upF(c_4)$ is false. Now we can compactly write the SSA for SYSADMIN:

- **Successor State Axiom:**

$$Up(c, do(a, s)) \equiv a \sqsupseteq upS(c, s) \lor Up(c, s) \land \neg a \sqsupseteq upF(c, s)$$

[2]Throughout this paper, we will often omit a variable's sort if its first letter matches its sort (e.g., c, c_1, c^* are all of sort C).

[3]There may be multiple *classes* of sub-actions, in which case the Cartesian product for the joint action is over the union of sub-action sets for *each* class.

We do not yet consider the $reboot(c)$ action as we treat it as a joint *stochastic* action chosen by the user and thus defer its description until we introduce stochasticity into our model.

The regression of a formula ψ through an a joint action a is a formula ψ' that holds prior to a being performed iff ψ holds after a. We refer the reader to [14] for a a discussion of the $Regr[\cdot, \cdot]$ operator[4] and how it can be efficiently computed given the SSAs. For the SYSADMIN example, we note the result is trivial:

$$Regr[Up(c_i, s);\ \cdots \circ upS(c_i) \circ \cdots] \equiv \top \quad (6)$$

$$Regr[Up(c_i, s);\ \cdots \circ upF(c_i) \circ \cdots] \equiv \bot \quad (7)$$

Case Representation and Operators

Prior to generalizing the situation calculus to permit a first-order representation of MDPs, we introduce a *case notation* to allow first-order specifications of the rewards, transitions, and values for FOMDPs (see [2] for formal details):

$$t = \begin{array}{|c|}\hline \phi_1 : t_1 \\ \hline : \quad : \quad : \\ \hline \phi_n : t_n \\ \hline \end{array} \quad \equiv \quad \bigvee_{i \le n} \{\phi_i \land t = t_i\}$$

Here the ϕ_i are *state formulae* (whose situation term does not use do) and the t_i are terms. Often the t_i will be constants and the ϕ_i will partition state space.

Intuitively, to perform a *binary* operation on case statements, we simply take the cross-product of their partitions and perform the corresponding operation on the resulting paired partitions. Letting each ϕ_i and ψ_j denote generic first-order formulae, we can perform the "cross-sum" $\oplus$ of two case statements in the following manner:

$$\begin{array}{|c|}\hline \phi_1 : 10 \\ \hline \phi_2 : 20 \\ \hline \end{array} \oplus \begin{array}{|c|}\hline \psi_1 : 1 \\ \hline \psi_2 : 2 \\ \hline \end{array} = \begin{array}{|c|}\hline \phi_1 \land \psi_1 : 11 \\ \hline \phi_1 \land \psi_2 : 12 \\ \hline \phi_2 \land \psi_1 : 21 \\ \hline \phi_2 \land \psi_2 : 22 \\ \hline \end{array}$$

Likewise, we can perform $\ominus$ and $\otimes$ by, respectively, subtracting or multiplying partition values (as opposed to adding them) to obtain the result. Some partitions resulting from the application of the $\oplus$, $\ominus$, and $\otimes$ operators may be inconsistent and should be deleted.

We use three *unary* operators on cases [2; 15]: $Regr$, $\exists \vec{x}$, $\max$. Regression $Regr[C; a]$ and existential quantification $\exists \vec{x} C$ can both be applied directly to the individual partition formulae ϕ_i of case C. The maximization operation $\max C$ sorts the partitions of case C from largest to smallest, rendering them disjoint in a manner that ensures each portion of state space is assigned the highest value.

Sum and Product Case Aggregators

We introduce the novel construct of sum and product case aggregators that permit the specification of indefinite-length sums and products over all instantiations of a case statement for a given sort. The sum/product aggregators are defined in terms of the $\oplus$/$\otimes$ operators as follows (where $n = |C|$):

$$\sum_{c \in C} case(c, s) = case(c_1, s) \oplus \cdots \oplus case(c_n, s)$$

$$\prod_{c \in C} case(c, s) = case(c_1, s) \oplus \cdots \oplus case(c_n, s)$$

[4]For readability, we abuse notation and write $Regr(\phi(do(a, s)))$ instead as $Regr[\phi(s);\ a]$.

While the sum and product aggregator can easily be expanded for *finite* sums, there is generally no finite representation for indefinitely large n due to the piecewise constant nature of the case representation (i.e., even if the $\oplus$/$\otimes$ is explicitly computed, there may be an indefinite number of distinct constants to represent in the resulting case).

Since the sum aggregator is defined in terms of the $\oplus$ case operator, the standard properties of commutativity and associativity hold. Likewise, commutativity, associativity, and distributivity of $\otimes$ over $\oplus$ hold for the product aggregator due to its definition in terms of $\otimes$. Additionally, we know that $Regr[\cdot,\cdot]$ distributes through the $\oplus$/$\otimes$ operators [15], so we can also infer that it distributes through $\sum_c$ and $\prod_c$.

Stochastic Joint Actions and the Situation Calculus

In the factored FOMDP framework, we must specify how stochastic actions $U(\vec{x})$ under the control of the "user" decompose according to Nature's choice probability distribution [2] into joint deterministic actions so that we can use them within the deterministic situation calculus. Recalling our previous discussion, let $A(\vec{c})$ denote a set of deterministic sub-action outcomes and define the random variable $a(\vec{c}) \in A(\vec{c})$. We motivated the definition of sub-actions by the fact that they represented independent outcomes, so let us specify the distribution over these outcomes as independent probabilities conditioned on the joint stochastic action and current situation: $P(a(\vec{c})|U(\vec{x}),s) = pCase_U(\vec{c},\vec{x},s)$. Now we can easily express the distribution over the joint deterministic actions $a \in A$ in a factored manner using the product aggregator:

$$P(a|U(\vec{x}),s) = \prod_{\vec{c}} P(a(\vec{c})|U(\vec{x}),s) = \prod_{\vec{c}} pCase_U(\vec{c},\vec{x},s) \quad (8)$$

It is straightforward to see that this defines a proper probability distribution over the decomposition of joint stochastic actions into joint deterministic actions. This distribution makes an extreme independence assumption of sub-action outcomes, but this can be relaxed by jointly clustering small sets of sub-actions into a joint random variables.

Formalizing the SYSADMIN Factored FOMDP

Now that we have specified all of the ingredients of a factored FOMDP, let us specify the remaining aspects of the SYSADMIN problem. In addition to an axiomatization of the deterministic situation calculus aspects of a factored FOMDP (already defined for SysAdmin), we must specify the reward and transition function:

Example 2 (SYSADMIN Factored FOMDP). *The reward is easily expressed using sum aggregators:*

$$rCase(s) = \sum_c \left(\begin{array}{|l|}\hline Up(c,s) \ : \ 1 \\ \hline \neg Up(c,s) : \ 0 \\ \hline \end{array} \right) \quad (9)$$

We introduce the predicate $Conn(c_j,c_i)$ to indicate that there is a directed connection from computer c_j to c_i. Next we specify the components of Nature's choice probability distribution $pCase_{reboot}(c,x,s)$ over deterministic sub-action outcomes of the user's action $reboot(x)$:

$$P(upS(c_i)|reboot(x) \wedge x = c_i, s) = \boxed{\top \ : \ 1} \quad (10)$$

$$P(upS(c_i)|reboot(x) \wedge x \neq c_i, s) = \quad (11)$$

$$\begin{array}{|l|}\hline Up(c_i,s) \ : \ 0.95 \\ \hline \neg Up(c_i,s) : \ 0.05 \\ \hline \end{array} \otimes \frac{1 + \sum_d \left(\begin{array}{|l|}\hline Conn(d,c_i) \wedge Up(d,s) \ : \ 1 \\ \hline \neg Conn(d,c_i) \vee \neg Up(d,s) : \ 0 \\ \hline \end{array} \right)}{1 + \sum_d \left(\begin{array}{|l|}\hline Conn(d,c_i) \ : \ 1 \\ \hline \neg Conn(d,c_i) : \ 0 \\ \hline \end{array} \right)}$$

Here we see that the probability that a computer will be running if it was explicitly rebooted is 1. Otherwise, the probability that a computer is running depends on its previous status and the proportion of computers with incoming connections that are running. The probability of the failure outcome $upF(c_i)$ is just the complement of the success case:

$$P(upF(c_i)|U(c_i)) = \boxed{\top \ : \ 1} \ominus P(upS(c_i)|U(c_i)) \quad (12)$$

We can easily combine $P(a(c)|U(x),s)$ into a joint probability $P(a|U(\vec{x},s))$ based on Eq 8.

Symbolic Dynamic Programming

Exploiting Irrelevance

An important aspect of efficiency in the dynamic programming solution of propositionally factored MDPs is exploiting probabilistic independence in the DBN representation of the transition distribution. The same will be true for factored FOMDPs except that now we must provide a novel first-order generalization of probabilistic independence:

Definition 1. *A set of deterministic sub-action outcomes B is* irrelevant *to a formula $\phi(s)$ (abbreviated $Irr[\phi(s),B]$)* iff *the following holds:*

$$\forall b \in B. \ Regr[\phi(s),b] \equiv \phi(s) \ . \quad (13)$$

In general, we can prove case equivalence by converting the case representation to its logical equivalent [2] and querying an off-the-shelf theorem prover. Most often though, a simple syntactic comparison will allow us to show structural equivalence without the need for theorem proving.

To make use of this axiom, we must make some additional constraints on the definition of deterministic sub-actions:

Assumption 1. *For all joint deterministic actions $a \in A$ and deterministic sub-actions $a \sqsupseteq b$ and $a \sqsupseteq c$ where $b \neq c$, no ground fluent can be affected by both b and c.*

Effectively we are claiming here that the effects of all sub-actions of a joint action are orthogonal and do not interfere with each other. While this may seem like a strong assumption, it is only a modeling constraint and *any* sub-action that violates this assumption can be decomposed into multiple correlated sub-actions that obey this constraint – an idea similar to joining variables connected by sychronic arcs in DBNs. Nonetheless, it is easy to see that this assumption holds directly for SYSADMIN because each stochastic sub-action outcome $a \in A(c_i)$ affects only $Up(c_i,s)$ and no other $Up(c_j,s)$ when $c_j \neq c_i$. With this, we arrive at the following axiom that will allow us to simplify our representation during first-order decision-theoretic regression:

$$Irr[\phi(s),B] \supset \big\{\forall b \in B. \forall a \in A. \forall c. \ (a = b \circ c) \supset \quad (14)$$
$$Regr[\phi(s), b \circ c] \equiv Regr[\phi(s),c]\big\}$$

Backup Operators

Now that we have specified a compact representation of Nature's probability distribution over deterministic actions, we will exploit this structure in the backup operators that are the building blocks of symbolic dynamic programming.

In the spirit of the B^a backup operator from propositionally factored MDPs, we begin with the basic definition of the backup operator $B^{U(\vec{x})}[\cdot]$ for stochastic joint action $U(\vec{x})$ in factored FOMDPs:

$$B^{U(\vec{x})}[vCase(s)] = \gamma \bigoplus_{a \in A} \Big[P(a|U(\vec{x})) \otimes Regr[vCase(s), a] \Big]$$

This is the same operation as first-order decision-theoretic regression (FODTR) [2], except with an implicitly factored transition distribution (recall Eq 8). We refer the reader to the literature for details.

We note that $B^{U(\vec{x})}[\cdot]$ is a linear operator just like $B^a[\cdot]$ and thus can be distributed through the $\oplus$ case operator. We will heavily exploit this fact later when we introduce linear-value approximation for factored FOMDPs, just as we did in the case of propositionally factored MDPs.

We now provide a formal example of how we can perform $B^U_{\max}(vCase(s))$ for the SysAdmin problems. Say that we start with $vCase(s) = rCase(s)$ from Equation 9. Then we apply the backup to this value function for stochastic action $reboot(x)$ and push the $Regr$ in as far as possible to obtain the following:

$$B^{reboot(x)}[vCase(s)] = \gamma \bigoplus_{a_1 \in A(c_1), \ldots, a_n \in A(c_n)}$$
$$\Big[\Big(\prod_{i=1}^{n} P(a_i|U) \Big) \otimes \sum_c \begin{array}{|l|}\hline Regr[Up(c,s), a_1 \circ \cdots \circ a_n] \;\; : 1 \\ \hline Regr[\neg Up(c,s), a_1 \circ \cdots \circ a_n] : 0 \\ \hline \end{array} \Big]$$

Now we distribute $\prod$ through $\sum$ and reorder $\sum$ with $\bigoplus$:

$$B^{reboot(x)}[vCase(s)] = \gamma \sum_c \Big[\bigoplus_{a_1 \in A(c_1), \ldots, a_n \in A(c_n)}$$
$$\Big(\prod_{i=1}^{n} P(a_i|U) \Big) \otimes \begin{array}{|l|}\hline Regr[Up(c,s), a_1 \circ \cdots \circ a_n] \;\; : 1 \\ \hline Regr[\neg Up(c,s), a_1 \circ \cdots \circ a_n] : 0 \\ \hline \end{array} \Big]$$

This last step is extremely important because it captures the factored probability distribution $\prod$ inside a specific c being summed over. Thus, for all SysAdmin problems, we now exploit the othogonality of sub-action effects from Assumption 1 to prove $Irr[Up(c,s), A(d)]$ for all $A(d)$ s.t. $d \neq c$. Thus, we can substantially simplify as a result of Axiom 14:

$$B^{reboot(x)}[vCase(s)] =$$
$$\gamma \sum_c \Big[\bigoplus_{a \in A(c)} P(a|U) \otimes \begin{array}{|l|}\hline Regr[Up(c,s), a] \;\; : 1 \\ \hline Regr[\neg Up(c,s), a] : 0 \\ \hline \end{array} \Big]$$

Now we explicitly perform the $\bigoplus$ over the sub-actions:

$$B^{reboot(x)}[vCase(s)] =$$
$$\gamma \sum_c \Big[P(upS(c)|U(x))) \otimes \begin{array}{|l|}\hline Regr[Up(c,s), upS(c)] \;\; : 1 \\ \hline Regr[\neg Up(c,s), upS(c)] : 0 \\ \hline \end{array}$$
$$\oplus \;\; (1 - P(upF(c)|U(x))) \otimes \begin{array}{|l|}\hline Regr[Up(c,s), upF(c)] \;\; : 1 \\ \hline Regr[\neg Up(c,s), upF(c)] : 0 \\ \hline \end{array} \Big]$$

Recalling the results of regression from Eqs 6 & 7, we note that the regressed top product simplifies to 1 and the regressed bottom product simplifies to 0. Thus, recalling the definition of $P(upS(c)|reboot(x))$ from Example 2, we obtain the final pleasing result:

$$B^{reboot(x)}[vCase(s)] = \gamma \sum_c pCase_{reboot}(c, x, s)$$

To complete the symbolic dynamic programming (SDP) step required for value iteration [2], we need to add the reward and apply the unary $\exists x$ operator followed by the unary $\max$ operator. Doing this ensures (symbolically) that the maximum possible value is achieved over *all* possible action instantiations, thus achieving *full action abstraction*. In the special case of SysAdmin where $reboot(x)$ is the only action schema, this completes one SDP step when $vCase^0 = rCase(s)$:

$$vCase^1(s) = rCase(s) \oplus \gamma \max \exists x . B^{reboot(x)}[vCase^0(s)] \quad (15)$$
$$= \sum_c \left(\begin{array}{|l|}\hline Up(c,s) \;\;\; : 1 \\ \hline \neg Up(c,s) : 0 \\ \hline \end{array} \right) \oplus \gamma \max \exists x . \sum_c pCase_{reboot}(c, x, s)$$

At this point, we have left two open questions:

Multiple Actions: We have discussed the solution for one action $U(\vec{x})$, but in practice, there can of course be many action schemata $U_i(\vec{x})$. It turns out that maximizing over multiple actions can be achieved easily via the solution to the next question.

Maximization and Quantification: So far, we have left the $\max$ in symbolic form although it should be clear that doing so prevents structure from being exploited in future backups. What we really want to do is perform an explicit maximization that gets rid of the $\max$ operator, however the indefinite $\sum_c$ makes this much harder than simply pushing the $\exists \vec{x}$ into each case statement as done for non-factored SDP [2]. However, we can do this indirectly through the axiomatization of a policy for "choosing" the optimal $\vec{x}$ in every situation s, this removing the need for the explicit $\max \exists \vec{x}$. We note that it is a straightforward exercise to generalize the policy extraction method of [8] to the first-order case.

We conclude our discussion of SDP by noting that in practice, the value function representation blows up uncontrollably in only a few steps of value iteration due to the combinatorial effects of the case operators applied during value iteration. The difficulty obtaining exact solutions leads us to explore approximate solution techniques that need only perform *one* backup in the next section.

Approximate Linear Programming

Linear-value Approximation

In this section, we demonstrate how we can represent a compact approximation of a value function for a factored FOMDP defined with rewards expressed using sum aggregators. We represent each first-order basis function as a sum of k basis functions much as we did for propositionally factored MDPs. However, using the sum aggregator, we can tie parameters across k *classes* of basis functions given by a parameterized $bCase_i(c, s)$ statement:

$$vCase(s) = \bigoplus_{i=1}^{k} w_i \sum_c bCase_i(c, s) \quad (16)$$

Even though we are only finding an approximation of the value function, it should not be hard to see that only a few simple first-order basis functions should account for a large portion of the optimal policy. For example, the following value function representation accounts for the single (unary) and pair (binary) basis functions commonly used in the SYSADMIN literature [8; 17] if the parameters are tied for each of the unary and pair basis function classes:

$$vCase(s) = w_1 \sum_c \boxed{\begin{array}{l|l} Up(c,s) & : 1 \\ \hline \neg Up(c,s) & : 0 \end{array}}$$

$$\oplus\ w_2 \sum_c \boxed{\begin{array}{l|l} Up(c,s) \wedge \exists c_2 Conn(c,c_2) \wedge Up(c_2) & : 1 \\ \hline \neg(Up(c,s) \wedge \exists c_2 Conn(c,c_2) \wedge Up(c_2)) & : 0 \end{array}}$$

There are a few motivations this value representation:

Expressivity: Our approximate value function *should* be able to exactly represent the reward. Clearly the sum over the first basis function above allows us to exactly represent the reward in SYSADMIN, while if it were defined with an $\exists c$ as opposed to a $\sum_c$, it would be impossible for a fixed-weight value function to *scale proportionally* to the reward as the domain size increased.

Efficiency: The use of basis function classes and parameter tying considerably reduces the complexity of the value approximation problem by compactly representing an indefinite number of ground basis function instances. While current ALP solutions scale with the number of basis functions, we will demonstrate that our solutions scale instead with the number of basis function *classes*.

Space limitations prevent us from discussing basis function generation, but we note that regression-based techniques of [6; 16] directly generalize to factored FOMDPs.

Factored First-order ALP

Now, we combine ideas from the previously described approximate linear programming (ALP) approach to factored MDPs [8; 17] with ideas from first-order ALP (FOALP) for FOMDPs [15] to specify the following factored FOALP (fFOALP) solution for factored FOMDPs:

$$\begin{aligned} &\text{Variables: } w_i \ ; \quad \forall i \le k \\ &\text{Minimize: } \sum_s vCase(s) \qquad (17) \\ &\text{Subject to: } 0 \ge rCase(s) \oplus B^{U(\vec{x})}[vCase(s)] \\ &\qquad\qquad \ominus vCase(s) \ ; \ \forall\, U(\vec{x}), s \end{aligned}$$

One can verify that this LP solution is in the spirit of the ALP LP from Eq 4. However, we have two problems: first, we have infinitely many constraints (i.e., a constraint for all possible action $U(\vec{x})$ and situation s instantions); second, the objective is ill-defined as it is a sum over an infinite number of situations s. Drawing on ideas from [15] and exploiting commutativity of $\oplus$ with $\sum$, we approximate the above fFOALP objective as follows:

$$\begin{aligned} \sum_s vCase(s) &= \bigoplus_{i=1}^{k} w_i \cdot \sum_s \sum_c bCase_i(c,s) \\ &\sim \sum_{i=1}^{k} w_i \sum_{\langle \phi_j, t_j \rangle \in bCase_i} \frac{t_j}{|bCase_i|} \qquad (18) \end{aligned}$$

Here, $|bCase_i|$ represents the number of partitions in $bCase_i$ and for each basis function, we sum over the value t_j of each partition $\langle \phi_j, t_j \rangle \in bCase_i$ normalized by $|bCase_i|$. This gives an approximation of the importance of each weight w_i in proportion to the overall value function.

Constraint Generation

Now we turn to solving for maximally violated constraints in a constraint generation solution to the first-order LP in Eq 17. We assume here that each basis function takes the form $\sum_c bCase_i(c,s)$ and the reward takes the form $\sum_c rCase(c,s)$ where the $\sum_c$ in each refer to the same object domain.

$$\begin{aligned} 0 \ge \sum_c (rCase(c,s)) \oplus \bigoplus_{i=1}^{k} w_i \left(\sum_c B^{U(\vec{x})}[bCase(s)] \right) \\ \ominus \bigoplus_{i=1}^{k} w_i \left(\sum_c bCase_i(c,s) \right) \ ; \ \forall\, U(\vec{x}), s \end{aligned}$$

As done for propositionally factored MDPs, we have exploited linearity of the first-order $B^{U(\vec{x})}[\cdot]$ operator to distribute it through the $\oplus$ and $\sum$. Unsurprisingly, these constraints exhibit the same concise factored form that we observed in Eq 5. However, the one difference is the infinite number of constraints mentioned previously. For now, we resolve these issues with symbolic $\max$ and $\exists \vec{x}$ case operators, ensuring each state is assigned its highest value:

$$\begin{aligned} 0 \ge \max_s \exists \vec{x}. \Bigg\{ \sum_c (rCase(c,s)) \oplus \bigoplus_{i=1}^{k} w_i \left(\sum_c B^{U(\vec{x})}[bCase(s)] \right) \\ \ominus \bigoplus_{i=1}^{k} w_i \left(\sum_c bCase_i(c,s) \right) \Bigg\} \ ; \ \forall\, U \end{aligned}$$

At this point, the constraint expression is quite cluttered although we note that it has a simple generic form that is a sum over p parameterized case statements that we can achieve after renaming and exploiting commutativity of $\sum$ with $\oplus$:

$$0 \ge \max_s \exists \vec{x} \Big[\sum_c case_1(c,\vec{x},s) \oplus \ldots \oplus \sum_c case_p(c,\vec{x},s) \Big] \quad (19)$$

This constraint form is very similar to that solved in [15] with one exception – here we have the addition of the sum aggregator which prevents us from achieving a finite representation of the constraints in all cases (recall that $\sum_c$ is an indefinitely large sum). We tackle this problem next.

Solving Indefinite Constraints

While we could conceive of trying to find a finite number of constraints that closely approximate the form in Eq 19, it is not clear how to ensure a good approximation for all domain sizes. On the other hand, grounding these constraints for a specific domain instantiation is clearly not a good idea since this solution would scale proportionally to the domain size.

Fortunately, there is a middle ground that has received a lot of research attention very recently – first-order probabilistic inference (FOPI) [13; 3]. In this approach, rather than making a *domain closure* assumption and grounding, a much less restrictive *domain size* assumption is made. This

Given: $case_i =$

x_i	x_{i+1}	
$\perp$	$\perp$	1
$\perp$	$\top$	-5
$\top$	$\perp$	-5
$\top$	$\top$	0

$\mathbf{r(2)}:$

x_1	x_3	
$\perp$	$\perp$	2
$\perp$	$\top$	-4
$\top$	$\perp$	-4
$\top$	$\top$	0

$\mathbf{r(4)}:$

x_1	x_5	
$\perp$	$\perp$	4
$\perp$	$\top$	-2
$\top$	$\perp$	-2
$\top$	$\top$	0

$\mathbf{r(8)}:$

x_1	x_9	
$\perp$	$\perp$	8
$\perp$	$\top$	2
$\top$	$\perp$	2
$\top$	$\top$	0

$\mathbf{r(16)}:$

x_1	x_{17}	
$\perp$	$\perp$	16
$\perp$	$\top$	10
$\top$	$\perp$	10
$\top$	$\top$	4

Compute: $r(n) = \max_{x_2,\ldots,x_n} \sum_{i=1}^{n} case_i$

Cost Network:

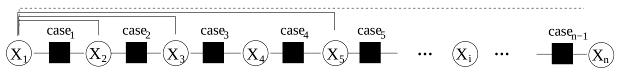


Figure 2: An example of *linear elimination*. Given identically structured, linearly connected case statements and the goal to compute the max over all variables except the first and last, each previous solution can be used to *double* the size of the next solution due to the symmetry inherent in the elimination (e.g., $r(2) = \max_{x_2} \sum_{i=1}^{2} case_i$ and is structurally identical to $\max_{x_4} \sum_{i=3}^{4} case_i$ modulo variable renaming, thus leaving x_3 to be eliminated from the sum to obtain $r(4)$). Thus, the elimination can be done in $O(\log n)$ space and time.

allows the solution to be carried out in a lifted manner and the solutions to be parameterized by the domain size. Recent work [4] has explicitly examined a "first-order" $\max$-$\sum$ cost network similar to Eq 19 that we would need to evaluate during constraint generation.

Since we are making a domain size assumption, we can exploit a non-obvious, but very powerful transformation for rewriting an $\exists \vec{x}$ in a concise $\sum_c case(c)$ format. We handle the case for a single $\exists x$ since it can be applied sequentially for each variable in the $\exists \vec{x}$ case. Assuming x's sort is C and $n = |C|$, we know $(\exists x \in C) \equiv (x = c_1 \vee \ldots \vee x = c_n)$. Let us now introduce a new relation $b(c)$ and a function $next(c)$ that defines some total order over all $c \in C$. Intuitively, we assign the meaning of $b(c)$ to be "$x = d$ for some d coming *before* c in the total order". Now we define the following case statement:

$$eCase(c,s) = \begin{array}{|l|}\hline b(c) \supset b(next(c)) \;\; : \;\; 0 \\ \hline b(c) \wedge \neg b(next(c)) : -\infty \\ \hline\end{array} \quad (20)$$

Given this definition, it should be clear that $\sum_c eCase(c)$ will only take the value 0 when $b(c) \supset b(next(c))$ for all $c \in C$. Why are we doing this? Because now $\sum_c eCase(c)$ can be used to encode the $\exists c$ constraint in a $\max$-$\sum$ setting by specifying that x is "chosen" to be *exactly one* of the $c \in C$. Clearly, the transition from $b(c) = \perp$ to $b(next(c)) = \top$ will occur once in a maximal constraint containing $\sum_c eCase(c)$. So quite simply, $(x = c) \equiv \neg b(c) \wedge b(next(c))$ so that now any occurrence of $(x = c)$ can be replaced with $b(c) \wedge b(next(c))$. If we perform this replacement, we obtain the final form of the constraints exactly as we need them to apply to FOPI:

$$0 \geq \max_s \sum_c \big(case_1(c,s) \oplus .. \oplus case_p(c,s) \oplus eCase(c,s) \big) \quad (21)$$

To generate maximally violated constraints for SysAdmin without grounding, we use two specific FOPI techniques: *inversion elimination* [4] and a novel technique termed *linear elimination* that we briefly cover here. Inversion elimination simply exploits cost networks with identical repeated subcomponents by evaluating the subcomponent once and multiplying the result by the number of "copies". Alternately, linear elimination exploits the evaluation of identical, linearly connected case statements. Because space does not permit a description of the full algorithm, we simply provide an intuitive example in Figure 2. We make two important notes regarding linear elimination: (1) It requires time and space logarithmic in the length of the chain. (2) Extracting lifted assignments can be done efficiently due to the known symmetry in the assignments.

To apply linear elimination to our SysAdmin problem, we note the following elements of the fFOALP constraints in Eq 21 all exhibit the identical, but linearly translated structure observed in Figure 2: (1) the $eCase(c,s)$ from Eq 20, (2) the basis functions from Eq 16, and (3) the SDP backup of the basis functions from Eq 15. As a consequence, their sum will exhibit identical linear structure and thus linear elimination can be applied to perform constraint generation in time that is logarithmic in the number of domain objects.

Empirical Results

We applied ALP and fFOALP solutions to the SysAdmin problem configurations from Figure 1 using unary basis functions; each of these network configurations represents a distinct class of MDP problems with its own optimal policy. Solution times and empirical performance are shown in Figure 3. We did not tie parameters for ALP in order to let it exploit the properties of individual computers; had we done so, ALP would have generated the same solution as fFOALP.

The most striking feature of the solution times is the scalability of fFOALP over ALP. While ALP's solution time grows fast and becomes impractical by 10^2 computers on all network configurations, fFOALP easily provides solutions for up to 10^5 computers. ALP's time complexity turns out to scale quadratically in the domain size n since it must evaluate exactly n ground constraints (i.e., n ground actions), each of length $O(n)$ (i.e., n basis functions). fFOALP avoids these sources of complexity by applying an action schema to perform one backup for *all* possible action instantiations at once *and* exploiting the symmetric relational structure of the constraints by using linear elimination (plus inversion elimination for the star network) to evaluate them in $O(\log(n))$ time. Thus, the fFOALP solutions generated $O(\log(n))$ constraints in $O(\log(n))$ time for an LP with a constant number of variables. Since LPs are polynomial-time solvable, the complexity of the fFOALP solutions for the above SysAdmin domains are provably polynomial in $O(log(n))$

In terms of performance, as the number of computers in the network increases, the problem becomes much more difficult, leading to a necessary degradation of even the optimal policy value. Comparatively though, the implicit pa-

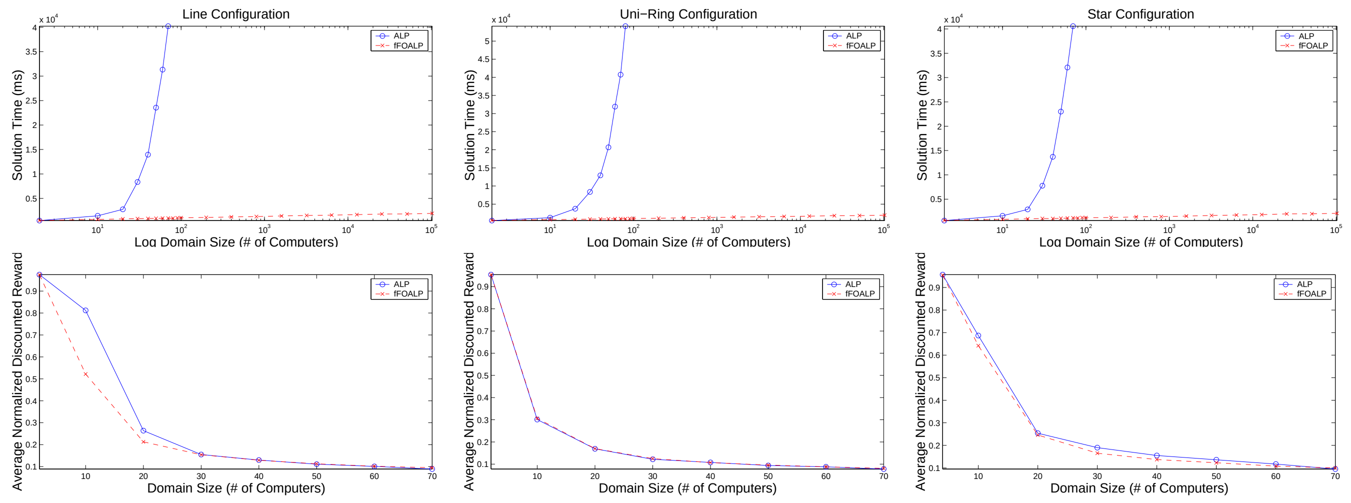


Figure 3: Factored FOALP and ALP solution times (top) and average normalized discounted reward (bottom) sampled over 200 trials of 200 steps vs. domain size for various network configurations (left:line, middle:unidirectional-ring, right:star) in the SYSADMIN problem.

rameter tying of fFOALP's basis function classes does not hurt it considerably in comparison to ALP; certainly, the difference becomes negligible for the networks as the domain size grows. This indicates that tying parameters across basis function classes may be a reasonable approach for large domains. Secondly, for completely symmetric cases like the unidirectional ring, we see that ALP and fFOALP produce exactly the same policy – albeit with fFOALP having produced this policy using much less computational effort.

Related Work and Concluding Remarks

We note that all other first-order MDP formalisms [2; 15; 16; 10; 11; 12] cannot represent factored structure in FOMDPs. Other non first-order approaches [5; 6; 7], require sampling where in the best case these approaches could never achieve sub-linear complexity in the number of domain objects.

In summary, we have contributed the sum and product aggregator language extension that has allowed us to specify factored FOMDPs that were previously impossible to represent in a domain-independent manner. And we have generalized solution techniques to exploit novel definitions of first-order independence as well as sum and product aggregator structure, including a novel FOPI technique for exploiting linear connectivity in relational structure. We have shown empirically that we can potentially solve these factored FOMDPs in time and space that scales polynomially in the logarithm of the domain size – results that were *impossible* to obtain for previous techniques that relied on grounding. Altogether, the language extensions, algorithms, and results presented here present a breakthrough in the state-of-the-art for the representation and solution of FOMDPs.

References

[1] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR*, 11:1–94, 1999.

[2] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *IJCAI-2001*.

[3] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *IJCAI-2005*.

[4] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. MPE and partial inversion in lifted probabilistic variable elimination. In *AAAI-2006*.

[5] Alan Fern, SungWook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias. In *NIPS-2003*.

[6] Charles Gretton and Sylvie Thiebaux. Exploiting first-order regression in inductive policy selection. In *UAI-2004*.

[7] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI-2003*.

[8] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venktaraman. Efficient solution methods for factored MDPs. *JAIR*, 19:399–468, 2002.

[9] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *UAI-1999*.

[10] Steffen Hölldobler and Olga Skvortsova. A logic-based approach to dynamic programming. In *In AAAI-2004 Workshop on Learning and Planning in Markov Processes*.

[11] Eldar Karabaev and Olga Skvortsova. A heuristic search algorithm for solving first-order MDPs. In *UAI-2005*.

[12] Kristian Kersting, Martijn van Otterlo, and Luc de Raedt. Bellman goes relational. In *ICML-2004*.

[13] David Poole. First-order probabilistic inference. In *IJCAI-2003*.

[14] Ray Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[15] Scott Sanner and Craig Boutilier. Approximate linear programming for first-order MDPs. In *UAI-2005*.

[16] Scott Sanner and Craig Boutilier. Practical linear-value approximation techniques for first-order MDPs. In *UAI-2006*.

[17] Dale Schuurmans and Relu Patrascu. Direct value approximation for factored MDPs. In *NIPS-2001*.

[18] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *NIPS-2000*.