Practical Solution Techniques for First-order MDPs ⋆

Scott Sanner *

Statistical Machine Learning Group
National ICT Australia
Canberra, ACT, 0200, Australia

Craig Boutilier

Department of Computer Science
University of Toronto
Toronto, ON M5S 3H5, Canada

Abstract

Many traditional solution approaches to relationally specified decision-theoretic planning problems (e.g., those stated in the probabilistic planning domain description language, or PPDDL) ground the specification with respect to a specific instantiation of domain objects and apply a solution approach directly to the resulting ground Markov decision process (MDP). Unfortunately, the space and time complexity of these grounded solution approaches are polynomial in the number of domain objects and exponential in the predicate arity and the number of nested quantifiers in the relational problem specification. An alternative to grounding a relational planning problem is to tackle the problem directly at the relational level. In this article, we propose one such approach that translates an expressive subset of the PPDDL representation to a first-order MDP (FOMDP) specification and then derives a domain-independent policy without grounding at any intermediate step. However, such generality does not come without its own set of challenges—the purpose of this article is to explore practical solution techniques for solving FOMDPs. To demonstrate the applicability of our techniques, we present proof-of-concept results of our first-order approximate linear programming (FOALP) planner on problems from the probabilistic track of the ICAPS 2004 and 2006 International Planning Competitions.

Key words: MDPS, first-order logic, planning

⋆ Parts of this article appeared in preliminary form in Sanner and Boutilier (2005, 2006).

* Corresponding author.
Email address: ssanner@nicta.com.au (Scott Sanner).

1 Introduction

There has been an extensive line of research over the years aimed at exploiting structure in order to compactly represent and efficiently solve decision-theoretic planning problems modeled as Markov decision processes (MDPs) (Boutilier et al., 1999). While traditional approaches from operations research typically use enumerated state and action models (Puterman, 1994), these have 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 by using a factored state and action representation and exploiting various independences in the reward and transition functions (Boutilier et al., 1999). The independencies and regularities laid bare by such representations can often be exploited in exact and approximate solution methods as well. Such techniques have permitted the practical solution of MDPs that would not have been possible using enumerated state and action models (Dearden and Boutilier, 1997; Hoey et al., 1999; St-Aubin et al., 2000; Guestrin et al., 2002).

However, factored representations are only one type of structure that can be exploited in the representation of MDPs. Many MDPs can be described abstractly in terms of classes of domain objects and relations between those domain objects that may change over time. For example, a logistics problem specified in the probabilistic planning domain description language (PPDDL) (Younes et al., 2005) may refer to domain objects such as boxes, trucks, and cities. If the objective is to deliver all boxes to their assigned destination cities then the locations of these boxes and trucks may change as a result of actions taken in pursuit of this objective. Since action templates such as loading or unloading a box are likely to apply generically to domain objects and can be specified independently of any ground domain instantiation (e.g., 4 trucks, 5 boxes, and 9 cities), this permits compact MDP descriptions by exploiting the existence of domain objects, relations over these objects, and the ability to express objectives and action effects using quantification.

Unfortunately, while relational specifications such as PPDDL permit very compact, domain-independent descriptions of a variety of MDPs, this compactness does not translate directly to effective solutions of the underlying planning problems. For example, one approach to solving a relational decision-theoretic planning problem might first construct sets of state variables and actions for all possible ground instantiations of each relation and action with respect to a specific domain (e.g., 4 trucks, 5 boxes, and 9 cities). Then this approach might apply known solution techniques to this ground factored representation of an MDP. Unfortunately, such an approach is domain-specific; and the size of the ground MDP grows polynomially in the number of domain objects, and exponentially in the predicate arity and the number of nested quantifiers in the problem specification. For sufficiently large do-

mains and complex relational MDP specifications, grounding may not be a viable option.

An alternative approach to grounding is to apply a solution approach directly at the relational level. In this article, we discuss one such technique that translates an expressive subset of the relational PPDDL representation to a *first-order MDP (FOMDP)* (Boutilier et al., 2001) specification. A symbolic policy may then be derived with respect to this FOMDP, resulting in a domain-independent solution that exploits a purely lifted version of the Bellman equations and avoids grounding at any intermediate step. This stands in contrast to alternate first-order approaches discussed in Section 6.2 that induce symbolic representations of the solution from samples of the Bellman equation in ground problem instances.

Unfortunately, the use of first-order logical languages to describe our FOMDP specification and solution introduces the need for computationally expensive logical simplification and theorem proving. While this means that exact solutions are not tractable for many FOMDPs, there is often a high degree of regularity and structure present in many FOMDPs that can be exploited by the approximate (heuristic) solution techniques proposed in this article. To this end, this article continues the tradition of exploiting structure to find effective solutions for large MDPs.

After providing a review of MDPs and relevant solution techniques in Section 2 and the FOMDP formalism and its solution via symbolic dynamic programming (Boutilier et al., 2001) in Section 3, we make the following contributions to the practical solution of FOMDPs:

(1) Section 3.2.2: We show how to translate a subset of PPDDL problems including universal and conditional effects to FOMDPs.
(2) Section 4.1: We show how to exploit the logical structure of reward, value, and transition functions using first-order extensions of algebraic decision diagrams (ADDs) (Bahar et al., 1993) for use in both exact and approximate FOMDP solutions.
(3) Section 4.2: We apply additive decomposition techniques to universal reward specifications in a manner that leads to efficient solutions for our FOMDP representation and reasonable empirical performance on example problems.
(4) Section 5.3: We show how to generalize the approximate linear programming technique for MDPs (Schweitzer and Seidmann, 1985; de Farias and Roy, 2003; Guestrin et al., 2002) to the case of FOMDPs by casting the optimization problem in terms of a first-order linear program.
(5) Section 5.4: We define a linear program (LP) with first-order constraints and provide a constraint generation algorithm that utilitizes a relational generalization of variable elimination (Zhang and Poole, 1996) to exploit constraint structure in the efficient solution of this *first-order LP (FOLP)*.

To demonstrate the efficacy of our techniques, we present proof-of-concept results

of our *first-order approximate linear programming (FOALP)* planner on problems from the probabilistic track of the ICAPS 2004 and 2006 International Planning Competitions in Section 5.6. Following this, we discuss a number of related first-order decision-theoretic planning approaches and discuss the relative advantages and disadvantages of each in Section 6. We conclude with a discussion of possible extensions to our techniques in Section 7.

2 Markov Decision Processes

Markov decision processes (MDPs) were first introduced and developed in the fields of operations research and economics (Bellman, 1957; Shapley, 1953; Howard, 1960). The MDP has since been adopted as a model for decision-theoretic planning with fully observable state in the field of artificial intelligence (Bertsekas, 1987; Bertsekas and Tsitsiklis, 1996; Boutilier et al., 1999) and as such provides the formal underpinning for the framework that we describe in this article. In this section, we describe various algorithmic approaches for making optimal sequential decisions in MDPs that we later generalize to the case of first-order MDPs. The following presentation derives from Puterman (1994).

2.1 The MDP model and Optimality Criteria

Formally, a finite state and action MDP is specified by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, h, \gamma \rangle$. $\mathcal{S}$ is a set of distinct states. An agent in an MDP can effect changes to its state by executing actions from the set $\mathcal{A}$. We base our initial presentation in this section on finite state and action MDPs; but in much of what follows, we will assume an infinite, discrete state and action space. The standard techniques for MDPs discussed here can be generalized to countable or continuous state and action spaces (Puterman, 1994).

The transition function $\mathcal{T}$ is a family of probability distributions $\mathcal{T}(s, a, s') = P(s'|a, s)$, which denotes the probability that the world transitions from $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ when action $a \in \mathcal{A}$ was executed. This representation enforces the Markov property: the distribution over states s_{t+1} at time $t + 1$ is independent of any previous state s_{t-i} and action a_{t-i}, $i \geq 1$, given s_t and a_t.

The preferences of the agent are encoded in a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. In addition to specifying single-step preferences, the agent must also specify how it trades off reward over the horizon h of remaining decision stages. In this article, we focus on the expected sum of discounted accumulated reward over an infinite horizon ($h = \infty$) since this is most compatible with the (approximate) linear programming approach that we adopt later. In the calculation of discounted

accumulated reward, we discount rewards t time steps into the future by a discount factor γ^t where $\gamma \in [0, 1]$. Throughout this article, we assume $\gamma < 1$. The use of $\gamma < 1$ allows one to model the notion that an immediate reward r is worth more than the equivalent reward delayed one or more time steps in the future. Practically, $\gamma < 1$ is required to ensure that the total expected reward is bounded in the case of infinite horizon MDPs.

A stationary policy takes the form $\pi : \mathcal{S} \rightarrow \mathcal{A}$, with $\pi(s)$ denoting the action to be executed in state s. The *value* of policy π is the expected sum of discounted future rewards over horizon h given that π is executed. Its value function is given by:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{h} \gamma^t \cdot r^t \;\middle|\; s_0 = s \right]. \tag{1}$$

where r^t is a reward obtained at time t, γ is a discount factor as defined above, and s_0 is the initial starting state.

A *greedy policy* π_V with respect to a value function V is simply any policy that takes an action in each state that maximizes expected value with respect to V, defined as follows:

$$\pi_V(s) = \arg\max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V(s') \right\} \tag{2}$$

Thus, from any value function, we can derive a corresponding greedy policy that represents the best action choice with respect to that value estimation.

An *optimal policy* π^* in an infinite horizon MDP maximizes the value function for all states. An optimal policy π^* is any greedy policy with respect to the optimal value function V^* and likewise the optimal value function is the value of an optimal policy, $V_{\pi^*}(s) = V^*(s)$. We note that V^* satisfies the following fixed-point equality:

$$V^*(s) = \max_a \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \cdot V^*(s') \right\}. \tag{3}$$

Finding V^* constitutes finding an *exact solution* to an MDP. Throughout the article, we use the term *solution* more loosely to denote some attempt at approximating V^*, whether the approximation guarantees error bounds or is simply heuristic.

2.2 MDP Solution Algorithms

In this section we describe several exact and approximate solution techniques for MDPs that we later extend to the first-order case.

2.2.1 Value iteration

We begin our discussion of MDP solutions by providing two equations that form the basis of the stochastic dynamic programming algorithms used to solve MDPs.

We define $V_\pi^0(s) = R(s, \pi(s))$ and then inductively define the *t-stage-to-go value function* for a policy π as follows:

$$V_\pi^t(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') \cdot V_\pi^{t-1}(s') \tag{4}$$

Based on this definition, Bellman's *principle of optimality* (Bellman, 1957) establishes the following relationship between the optimal value function at stage t and the optimal value function at the previous stage $t-1$:

$$V^t(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') \cdot V^{t-1}(s') \right\} \tag{5}$$

The computation of V^t from V^{t-1} via this relationship is referred to as a *Bellman backup*. The *value iteration* algorithm consists of repeatedly performing Bellman backups to compute these t-stage-to-go value functions.

We note that the Bellman backup is often rewritten in the following two steps to separate out the backup of a value function through a single action and the maximization of this value over all actions:

$$Q^t(s, a) = R(s, a) + \gamma \cdot \sum_{s' \in \mathcal{S}} T(s, a, s') \cdot V^{t-1}(s') \tag{6}$$

$$V^t(s) = \max_{a \in \mathcal{A}} \left\{ Q^t(s, a) \right\} \tag{7}$$

Puterman (1994) shows that terminating once the following Bellman error condition is met

$$\max_s |V^t(s) - V^{t-1}(s)| < \frac{\epsilon(1-\gamma)}{2\gamma} \tag{8}$$

guarantees that the estimated value function V^t is *ϵ-optimal* over an infinite horizon, that is, its value is within ϵ of the optimal value: $\max_s |V^t(s) - V^*(s)| < \epsilon$.

We note that the value iteration approach requires time polynomial in the backup depth d and the number of states and actions, i.e., $O(|\mathcal{S}|^2 \cdot |\mathcal{A}| \cdot d)$. Puterman (1994) provides a proof that value iteration converges linearly.

2.2.2 Linear programming

An MDP can also be solved using the following linear program (LP):

$$\begin{aligned} \text{Variables:} \quad & V(s),\ \forall s \in \mathcal{S} \\ \text{Minimize:} \quad & \sum_{s \in \mathcal{S}} V(s) \\ \text{Subject to:} \quad & 0 \geq R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)V(s') - V(s);\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A} \end{aligned} \quad (9)$$

Puterman (1994) provides a proof that the solution to this LP is the optimal value function for an MDP.

2.2.3 Approximate Linear Programming

One general and popular approximate solution technique for MDPs is that of linear-value function approximation (Schweitzer and Seidmann, 1985; Tsitsiklis and Van Roy, 1996; Koller and Parr, 1999, 2000; Schuurmans and Patrascu, 2001; Guestrin et al., 2002). Representing value functions as a linear combination of basis functions has many convenient computational properties, many of which will become evident as we incorporate relational structure in our MDP model. However, perhaps one of the most useful properties is that linear value function representations lead to MDP solutions requiring optimization with respect to linear objectives and linear constraints—that can be formulated as LPs.

In an n-state MDP, the exact value function can be specified as a vector in $\mathbb{R}^n$. This vector can be approximated by a value function $\tilde{V}_{\vec{w}}$ that is a linear combination of k fixed basis functions (or n-vectors), denoted $b_i(s)$:

$$\tilde{V}_{\vec{w}}(s) = \sum_{i=1}^{k} w_i \cdot b_i(s) \quad (10)$$

The linear subspace spanned by the basis set will generally not include the true value function, but one can use projection methods to minimize some error measure between the true value function and the linear combination of basis functions. The basis functions themselves can be specified by domain experts, constructed or learned in an automated fashion (e.g., Poupart et al. (2002); Mahadevan (2005)). We will consider first-order methods for automated basis function construction in Section 5 and related work in Section 6.

Approximate linear programming (ALP) is simply an extension of the linear programming solution of MDPs to the case where the value function is approximated. In a linear value function representation, the objective and constraints will be linear in the weights being optimized, leading to a direct LP formulation. Consequently,

we arrive at the following variant of the previous exact LP solution:

$$\begin{aligned}
&\text{Variables:} && \vec{w} \\
&\text{Minimize:} && \sum_{s \in \mathcal{S}} \tilde{V}_{\vec{w}}(s) \\
&\text{Subject to:} && 0 \geq R(s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)\tilde{V}_{\vec{w}}(s') - \tilde{V}_{\vec{w}}(s);\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A} \quad (11)
\end{aligned}$$

2.3 Selecting an MDP Solution Approach

The choice of whether to use a linear programming or dynamic programming solution to MDPs is not always clear. Linear programming offers a simple one-shot solution, but it relies on efficient LP solvers. Dynamic programming is straightforward to implement, but may require a large number of iterations to converge. However, the choice of exact vs. approximate is almost invariably determined by the size of the state space. For sufficiently large state spaces, approximate solution techniques are the only viable option. But this last statement depends critically on how one measures the size of the state space.

Despite their promise, the exact and approximate solution techniques discussed above must represent the value function (and policy, if required) as vectors or functions over an explicitly enumerated state (and action) space. This is simply not feasible for large-scale AI planning problems. Fortunately, there are many representations (e.g., factored or relational) well suited to decision-theoretic planning that do not require explicit state or action enumeration in either the problem representation or the solution. To this end, we will be concerned with the exploitation of relational planning structure for the remainder of this article.

3 First-order MDPs

Given that relational representations seem natural for planning problems, it makes sense to attempt to exploit this relational structure at a first-order level without resorting to grounding. This is precisely the idea behind the first-order MDP model (FOMDP) and its symbolic dynamic programming solution (Boutilier et al., 2001), which we review in this section. For the remainder of this article, when we refer to a FOMDP without further qualification, we refer to the specific formalization presented in Boutilier et al. (2001), although there are other possible first-order MDP formalizations and associated solution approaches (we discuss these alternatives in Section 6). The reader already familiar with the motivations for FOMDPs and the presentation and notation in Boutilier et al. (2001) may wish to skip this section and proceed directly to the main contributions of this article in Sections 4 and 5.

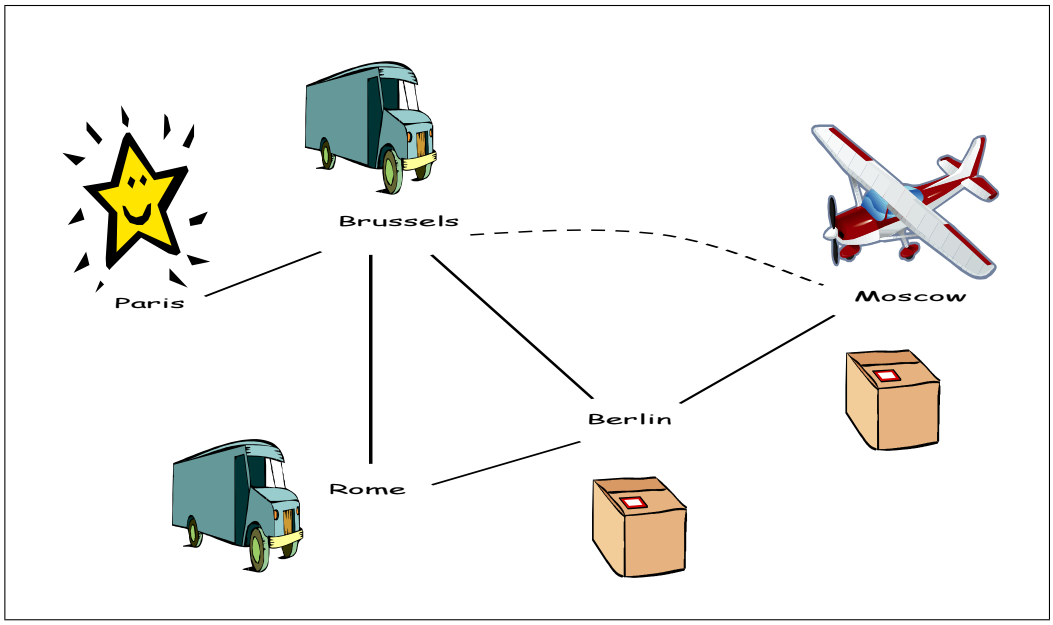

Fig. 1. An example BOXWORLD problem. Trucks may drive along solid lines and planes may fly along dashed lines. The goal in this instance is to get all boxes in Paris (indicated by the star).

3.1 Motivation

Before we introduce FOMDPs and their solution, we begin with the basics of relational planning problem specifications and motivate the need for exploiting this structure at a lifted first-order level rather than at a ground propositional level.

3.1.1 Relational Planning Specifications

We assume basic familiarity with unsorted first-order logic with equality. While we use a sorted notation for specifying object types of variables and predicate slots, we assume this sort information is compiled into an unsorted logical form where $\forall Sort : c\ \phi(c)$ is rewritten as $\forall c.\ Sort(c) \supset \phi(c)$ and likewise $\exists Sort : c\ \phi(c)$ is rewritten as $\exists c.\ Sort(c) \wedge \phi(c)$. Assuming these transformations, we draw on the logical notation and semantics for unsorted first-order logic given in Brachman and Levesque (2004). Specifically:

- *Predicate Symbols:* We assume a set of predicates P_i of each arity $0 \leq i \leq m$ for some finite maximum m. We assume "$=$"$\in P_2$ with its usual interpretation.
- *Function Symbols:* We assume a set of function symbols f_j of each arity $0 \leq j \leq n$ for some finite maximum n.

In addition, we use a few notational conventions. All predicates (including unary predicates denoting domain object classes) are capitalized and all variables and constants are lowercased. We denote the types of predicate arguments using the

notation $\phi(Sort_1, \ldots, Sort_k)$ for some predicate of arity k.[1]

We can view many decision-theoretic planning problems as consisting of classes of domain objects and the changing relations that hold between those objects at different points in time. For example, in the BOXWORLD logistics problem (Veloso, 1992) illustrated in Figure 1, we have four classes of domain objects: Box, $City$, $Truck$, and $Plane$. For the relations that hold between them, we have $BoxIn(Box, City)$, $BoxOnTruck(Box, Truck)$, $TruckIn(Truck, City)$, $PlaneIn(Plane, City)$, $BoxOnPlane(Box, Plane)$). In this framework, generic action templates such as loading or unloading a box from a truck or plane or driving trucks and flying planes between cities are likely to apply generically to domain objects and thus the planning problem can be specified independently of any ground domain instantiation.

One recent language for representing relational probabilistic planning problems is PPDDL (Younes et al., 2005). At its core, PPDDL is a probabilistic extension of a subset of PDDL conforming to the deterministic ADL planning language (Pednault, 1989); ADL, in turn, introduced universal and conditional effects into the STRIPS representation (Fikes and Nilsson, 1971). To see the compactness of a relational representation, we provide a (P)PDDL representation of the BOXWORLD problem in Figure 2 where for simplicity, we omit the $Plane$ class of objects and associated actions and relations and abbreviate $BoxOnTruck(Box : b, Truck : t)$ as $BoxOn(Box : b, Truck : t)$.

General PPDDL specifications can be more compact for some problems than the PPDDL subset we refer to in this article. For example, in general PPDDL, universal and conditional effects and probabilities can be arbitrarily nested, thus allowing for exponentially more compact representations of probabilistic action effects than can be achieved with probabilities only at the top-level of effects (Rintanen, 2003). In addition, there are some general PPDDL specifications that *cannot* be translated to the PPDDL subset described here. If the general PPDDL specification uses probabilistic effects *nested under* universal effects (e.g., each box falls off a truck with some independent probability), it is generally impossible to translate such a problem to the restricted PPDDL subset used here because it requires an indefinitely factored transition probability model that cannot be expressed with finite probability specifications restricted to the top level of effects. While we do not discuss such model-expressivity here, we refer the reader to Sanner and Boutilier (2007) and Chapter 6 of Sanner (2008) for a treatment of such issues in first-order MDPs.

While the meaning of the PPDDL representation in Figure 2 is intended to be relatively straightforward, there are a few important points that should be explained. First, we assume that actions can be executed in all states so we do not encode explicit preconditions. While this assumption is not necessary, it does not have any effect on the value of an optimal policy in a domain that already has a $noop$ action

[1] Logically, this requires a background theory axiom $\forall x_1, \ldots, x_k \;\; \phi(x_1, \ldots, x_k) \supset \bigwedge_{1=1}^{k} Sort_i(x_i)$ for each predicate $\phi(Sort_1, \ldots, Sort_k)$.

- **Domain Object Types**: Box, $Truck$, $City$
- **Relational ($\mathcal{S}$)tate Descriptors (with parameter sorts)**:
  $BoxIn(Box, City)$, $TruckIn(Truck, City)$, $BoxOn(Box, Truck)$
- **($\mathcal{R}$)eward**: $if\ [\exists Box : b. BoxIn(b, paris)]\ then\ 10\ else\ 0$
- **($\mathcal{A}$)ctions (with parameter sorts) and ($\mathcal{T}$)ransition Function**:
  - $load(Box : b, Truck : t)$:
    Effects (probability 0.9):
    $when\ [\exists City : c.\ BoxIn(b, c) \wedge TruckIn(t, c)]\ then\ [BoxOn(b, t)]$
    $\forall City : c.\ when\ [BoxIn(b, c) \wedge TruckIn(t, c)]\ then\ [\neg BoxIn(b, c)]$
  - $unload(Box : b, Truck : t)$:
    Effects (probability 0.9):
    $\forall City : c.\ when\ [BoxOn(b, t) \wedge TruckIn(t, c)]\ then\ [BoxIn(b, c)]$
    $when\ [\exists City : c.\ BoxOn(b, t) \wedge TruckIn(t, c)]\ then\ [\neg BoxOn(b, t)]$
  - $drive(Truck : t, City : c)$:
    Effects (probability 1.0)
    $when\ [\exists City : c_1.\ TruckIn(t, c_1)]\ then\ [TruckIn(t, c)]$
    $\forall City : c_1.\ when\ [TruckIn(t, c_1)]\ then\ [\neg TruckIn(t, c_1)]$
  - $noop$
    No effects.

Fig. 2. A PPDDL-style representation of a simple variant of the BOXWORLD problem. The deterministic PDDL subset would exclude the probabilistic annotations of effects assuming that all effects occur with probability 1.0.

and it helps simplify our later notation. When an action executes, each probabilistic effect is realized independently according to the specified probability. For example, the $unload$ action realizes its effects only 90% of the time, whereas the $drive$ action deterministically realizes its effects on each execution.

Probabilistic effects at the top-level of the effect specification consist of conjunctions of effects. Each individual effect can be *universal* and *conditional*. Universal effects denoted by universally quantified variables in the $then$ clause permit the effect to apply to an arbitrary number of objects not explicitly named in the action's parameter list. Conditional effects denoted by $when$ can be arbitrary first-order formulae specifying that the effects listed in the $then$ clause hold in the post-action state if the $when$ conditions hold in the pre-action state. When universally quantified variables are shared between the $when$/$then$ clause pair, we refer to such effects as *universal conditional*. We note that each individual effect is only allowed to mention one positive or negative relation in the $then$ portion of the clause. A conjunction of $then$ effects can be easily specified as multiple effects with the same $when$ condition. Disjunctive (i.e., non-deterministic) effects are prohibited in PPDDL. For example, when the $load(b, t)$ action is executed, its effects are realized with probability 0.9. When these effects are realized, then for any city c that satisfies $BoxIn(b, c) \wedge TruckIn(t, c)$ in the pre-action state, $BoxOn(b, t) \wedge \neg BoxIn(b, c)$

- **Domain Object Instantiation**:
  - $Box = \{box_1, box_2, box_3\}$, $Truck = \{truck_1, truck_2\}$, $City = \{paris, berlin, rome\}$
- **($\mathcal{S}$)tate-variable Atoms (i.e., binary state variables)**:
  - $BoxIn$:
    $\{BoxIn(box_1, paris), BoxIn(box_2, paris), BoxIn(box_3, paris),$
    $BoxIn(box_1, berlin), BoxIn(box_2, berlin), BoxIn(box_3, berlin),$
    $BoxIn(box_1, rome), BoxIn(box_2, rome), BoxIn(box_3, rome)\}$
  - $TruckIn$:
    $\{TruckIn(truck_1, paris), TruckIn(truck_1, berlin), TruckIn(truck_1, rome),$
    $TruckIn(truck_2, paris), TruckIn(truck_2, berlin), TruckIn(truck_2, rome)\}$
  - $BoxOn$:
    $\{BoxOn(box_1, truck_1), BoxOn(box_2, truck_1), BoxOn(box_3, truck_1),$
    $BoxOn(box_1, truck_2), BoxOn(box_2, truck_2), BoxOn(box_3, truck_2)\}$
- **($\mathcal{A}$)ctions**:
  - $load$:
    $\{load(box_1, truck_1), load(box_2, truck_1), load(box_3, truck_1)$
    $load(box_1, truck_2), load(box_2, truck_2)\}, load(box_3, truck_2)\}$
  - $unload$:
    $\{unload(box_1, truck_1), unload(box_2, truck_1), unload(box_3, truck_1),$
    $unload(box_1, truck_2), unload(box_2, truck_2)\}, unload(box_3, truck_2)\}$
  - $drive$:
    $\{drive(truck_1, paris), drive(truck_1, berlin), drive(truck_1, rome)$
    $drive(truck_2, paris), drive(truck_2, berlin), drive(truck_2, rome)$
- **($\mathcal{T}$)ransition Function:**
  Follows directly from ground instantiation of PPDDL actions in Figure 2.
- **($\mathcal{R}$)eward**:
  $if\ [BoxIn(box_1, paris) \vee BoxIn(box_2, paris) \vee BoxIn(box_3, paris)]\ then\ 10\ else\ 0$

Fig. 3. One possible ground MDP instantiation of the BOXWORLD FOMDP.

will hold in the post-action state since both effects have equivalent $when$ conditions. When these effects are not realized on 10% of the $load(b, t)$ executions, no state changes occur and it is equivalent to a $noop$ action.

One can easily see that this relationally specified domain-independent specification allows very compact MDP specifications when compared to a corresponding ground factored MDP representation. For example, consider instantiating the PPDDL problem in Figure 2 to the ground factored MDP representation in Figure 3 where we assume a problem instance with a domain instantiation of three boxes, three cities, and two trucks. While this is a trivially small domain instantiation, we note that its factored MDP representation requires 21 propositional atoms corresponding to over two million distinct states and 18 distinct actions that can

be executed in each state. And the reward, which uses existential quantification in the relational PPDDL specification must be grounded to obtain the corresponding factored MDP representation. Clearly, for n objects, the grounded factor for the formula $\exists Box : b.\ BoxIn(b, paris)$ will contain $|Box|$ state variables, but if the reward were changed to $\forall City : c\, \exists Box : b.\ BoxIn(b, c)$, the ground reward representation would contain $|Box| \cdot |City|$ state variables—thus implying a combinatorial growth in the number of nested quantifiers.

In general, the number of ground atoms for a factored MDP representation will scale linearly in the number of relations, exponentially in the arity of each relation (assuming more than one domain object), and polynomially in the number of domain objects that fill each relation argument. To see this, let us assume for simplicity that all object class instantiations have k instances. Then a single unary relation would be represented by k ground atoms, a binary relation by k^2 atoms, and an n-ary relation by k^n atoms. Similarly, the size of the grounding of any quantified formula is exponential in the number of nested quantifiers, linear in the number of relations, and exponential in the size of the domain object classes being quantified. Assuming k instances for all object classes and q nested (non-vacuous) quantifiers over formulae containing r relations, the resulting unsimplified ground representation of the formula would require rk^q ground atoms.

For sufficiently small predicate arities and levels of quantifier nesting (assuming these remain constant for a problem as the domain size varies), the space requirements for representing a ground MDP may be acceptable. Thus, if we have adequate space to permit the grounding of a relational MDP to obtain a factored MDP *and* we have the time to find an optimal solution to this factored MDP, then grounding gives us one approach to representing and solving relational MDPs for specific domain instances. However we note that while solving MDPs exactly is known to be polynomial in the number of states (see Section 2.2.2), the number of states is exponential in the number of ground atoms in a factored representation. This is Bellman's (1957) well-known curse of dimensionality and since the number of ground atoms is at least linear in domain size, it implies that the exact solution methods discussed previously require time at least exponential in the domain size. This precludes the general possibility of exact solutions to grounded relational MDPs for all but the smallest domain sizes. While this suggests the use of approximation methods for solving grounded MDPs, there are useful lifted alternatives to representing and solving relational MDPs that we discuss next.

3.1.2 Grounded vs. Lifted Solutions

In contrast to the grounded approach to representing relational MDPs as factored MDPs, it is important to point out that no matter how many domain objects there may be in an actual problem instance, the size of the PPDDL relational planning problem specification in Figure 2 remains constant. Consequently, this invites the

- if $(\exists b.BoxIn(b, paris))$
  then do $noop$ (value = 100.00)
- else if $(\exists b^*, t^*.TruckIn(t^*, paris) \wedge BoxOn(b^*, t^*))$
  then do $unload(b^*, t^*)$ (value = 89.0)
- else if $(\exists b, c, t^*.BoxOn(b, t^*) \wedge TruckIn(t, c)$
  then do $drive(t^*, paris)$ (value = 80.0)
- else if $(\exists b^*, c, t^*.BoxIn(b^*, c) \wedge TruckIn(t^*, c))$
  then do $load(b^*, t^*)$ (value = 72.0)
- else if $(\exists b, c_1^*, t^*, c_2.BoxIn(b, c_1^*) \wedge TruckIn(t^*, c_2))$
  then do $drive(t^*, c_1^*)$ (value = 64.7)
- else do $noop$ (value = 0.0)

Fig. 4. A decision-list representation of the expected discounted reward value for an exhaustive partitioning of the state space in the BOXWORLD problem. The optimal action is also shown for each partition where the optimal bindings of the action variables (denoted by a *) correspond to any binding satisfying those variable names in the state formula.

following question: if we can avoid a domain-dependent blowup in the representation of a relational MDP as in PPDDL, can we avoid a domain-dependent blowup in its solution too? Although we have yet to discuss the specifics of how we might find a domain independent solution to this PPDDL representation, in Figure 4 we provide an optimal domain-independent value function and its corresponding policy for the relational PPDDL specification of the BOXWORLD problem in Figure 2 (using discount factor $\gamma = 0.9$).

The key features to note here are the state and action abstraction in the value and policy representation that are afforded by the first-order specification and solution of the problem. That is, this solution does not refer to any specific set of domain objects, say just $City = \{paris, berlin, rome\}$, but rather it provides a solution for *all possible domain object instantiations*. And while the BOXWORLD problem could not be represented as a grounded factored MDP for sufficiently large domain instantiations, much less solved, a domain-independent solution to this particular problem is quite simple and applies to domain instances of any size due to the power of state and action abstraction afforded by the first-order logical representation.

Thus, an alternative idea to grounding a relational MDP specification and solving it for a particular domain instance is to translate the PPDDL relational specification to a first-order MDP representation that is directly amenable to solutions via lifted symbolic dynamic programming. This approach obtains a solution that applies universally to all possible domain instantiations and has a time complexity that is independent of domain size. As we will see, the power of this lifted style of solution is that it exploits the existence of domain objects, relations over these objects, and the ability to express objectives and action effects using quantification.

3.2 Situation Calculus Background

Before we present the first-order MDP (FOMDP) formalism, we discuss the basics of the situation calculus, which in turn provides the logical foundations for our FOMDP representation. We begin by describing the necessary background material from the situation calculus and Reiter's default solution to the frame problem (Reiter, 2001) required to understand FOMDPs. This includes a discussion of the basic ingredients of the situation calculus formulation: actions, situations, and fluents along with relevant axioms (e.g., unique names for actions and domain-specific axioms). Next we introduce effect axioms and explain how these can be derived from a PDDL specification. Then we show how effect axioms can be compiled into the successor-state axioms that underly the default solution to the frame problem of the situation calculus. We conclude by introducing the regression operator $Regr$ that will prove crucial to our symbolic dynamic programming solution to first-order MDPs.

3.2.1 Basic Ingredients

The situation calculus is a first-order language for axiomatizing dynamic worlds (McCarthy, 1963). Its basic language elements consist of actions, situations and fluents:

- *Actions*: Actions are first-order terms consisting of an action function symbol and arguments. For example, an action for loading box b on truck t in the running BOXWORLD example is represented by $load(b, t)$.
- *Situations*: A situation is a first-order term denoting a specific state. The initial situation is usually denoted by s_0 and subsequent situations resulting from action executions are obtained by applying the do function, $do(a, s)$ representing the situation resulting from executing action a in situation s. For example, the situation resulting from loading box b on truck t in the initial situation s_0 and then driving truck t to city c is given by the term $do(drive(t, c), do(load(b, t), s_0))$.
- *Fluents*: A fluent is a relation whose truth value varies from situation to situation. A fluent is simply a relation whose last argument is a situation term. For example, imagine an initial state s_0 in which fluent $BoxOn(b, t, s_0)$ is false, but fluents $TruckIn(t, c, s_0)$ and $BoxIn(b, c, s_0)$ are true. Then under the semantics of a deterministic version of the $load(b, t)$ action (which we formally define in a moment), $BoxOn(b, t, do(load(b, t), s_0))$ holds. We do not consider functional fluents in this exposition, but they are easily added to the language without adverse computational side effects (Reiter, 2001).

3.2.2 From PDDL to a First-order Logic Domain Theory

To axiomatize a PDDL domain theory in first-order logic, we must first consider how to describe the effects and non-effects of actions. We can begin by describing

positive and negative effect axioms that characterize how fluents change as a result of actions. Note that in the following presentation, all relations that can change between states in PPDDL have been rewritten as fluents with an extra situation term. In addition, we assume all axioms are implicitly universally quantified.

- *Positive Effect Axioms*: positive effect axioms state which actions can explicitly make each fluent true; for example:

$$[\exists c.\, a = load(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)] \supset BoxOn(b,t,do(a,s))$$
$$[\exists t.\, a = unload(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)] \supset BoxIn(b,c,do(a,s))$$
$$[\exists c_1.\, a = drive(t,c) \wedge TruckIn(t,c_1,s)] \supset TruckIn(t,c,do(a,s))$$

- *Negative Effect Axioms*: negative effect axioms state which actions can explicitly make each fluent false; for example:

$$[\exists c.\, a = unload(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)] \supset \neg BoxOn(b,t,do(a,s))]$$
$$[\exists t.\, a = load(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)] \supset \neg BoxIn(b,c,do(a,s))]$$
$$[\exists c.\, a = drive(t,c) \wedge TruckIn(t,c_1,s)] \supset \neg TruckIn(t,c_1,do(a,s))$$

In general, positive and negative effect axioms can be specified by considering all of the ways in which each action can affect each fluent. Fortunately, these axioms are easy to derive directly from the PDDL representation given in Figure 2. In fact, one can verify that these effect axioms are simply syntactic rewrites of the PDDL effects where we have made the following transformations:

(1) The action name from the PDDL effect is placed in an equality on the LHS of the $\supset$.
(2) All universal quantifiers for universal effects are dropped as all unquantified variables are assumed to be universally quantified in the effect axioms.
(3) The $when$ conditions of the PDDL effect are conjoined on the LHS of the $\supset$ with all fluents specified in terms of the situation s.
(4) The $then$ portion of the effect (which should be a single literal) is placed on the RHS of the $\supset$ and is parameterized by the post-action situation $do(a,s)$. Whether the literal is negated or non-negated respectively determines whether the resulting axiom should be negative or positive.
(5) Any free variables appearing only on the LHS of the $\supset$ *and* not appearing free in the action term are explicitly existentially quantified in the LHS.

This takes care of specifying *what changes*, however we have not provided any axioms for specifying *what does not change*, i.e., the so-called *frame axioms*. Obviously, if we want to prove anything useful in our theory, we have to specify frame axioms. Otherwise, we would never be able to infer the properties of a successor or predecessor state for an action as simple as a $noop$. However, specifying exactly

what does not change in a *compact* manner has been an extremely difficult problem to solve for the situation calculus—this is, of course, the infamous *frame problem*.

An especially elegant solution to the *frame problem* is that proposed by Reiter (1991). In this solution, we specify all positive and negative effects for a fluent, which conveniently, we have just done in our translation from PDDL above. We use the following normal form for positive effect axioms where F is a fluent and $\gamma_F^+(\vec{x}, a, s)$ represents a first-order formula that, if true in s, results in $F(\vec{x}, do(a, s))$ being true after action $a(\vec{x})$ is executed in situation s:

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)) \tag{12}$$

Likewise, we use the following normal form for negative effect axioms where $\gamma_F^-(\vec{x}, a, s)$ represents a first-order formula that if true in s, results in $F(\vec{x}, do(a, s))$ being false after action $a(\vec{x})$ is executed in situation s:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)) \tag{13}$$

We note that the potential difference between our previous presentation of positive and negative effect axioms and this normal form is that there is exactly *one* positive effect axiom for each positive fluent and *one* negative effect axiom for each negative fluent. This just happens to be the case in our example, but if it were otherwise, we could use the simple logical equivalence

$$[(C_1 \supset F) \wedge (C_2 \supset F)] \equiv [(C_1 \vee C_2) \supset F], \tag{14}$$

to rewrite any set of effect axioms derived from the PDDL subset of PPDDL into this normal form.

Next, we need to add in *unique name axioms* for all pairs of distinct action names A and B stating that

$$A(\vec{x}) \neq B(\vec{y}), \tag{15}$$

and also that identical actions have identical arguments:

$$A(x_1, \ldots, x_k) = A(y_1, \ldots, y_k) \supset x_1 = y_1 \wedge \ldots \wedge x_k = y_k \tag{16}$$

From this normal form, unique names axioms, and *explanation closure axioms* that state these are the only effects that hold in our world model, Reiter showed that we can build *successor state axioms (SSAs)* that compactly encode both the effect and frame axioms for a fluent. The format of the successor state axiom for a fluent F is as follows:

$$\begin{aligned} F(\vec{x}, do(a, s)) &\equiv \Phi_F(\vec{x}, a, s) \\ &\equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \end{aligned} \tag{17}$$

For our running BOXWORLD example, we obtain the following SSAs:

$$
\begin{aligned}
&BoxOn(b,t,do(a,s)) \equiv \Phi_{BoxOn}(b,t,a,s) \\
&\quad \equiv [\exists c.\, a = load(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)] \\
&\quad\quad \vee BoxOn(b,t,s) \wedge \neg [\exists c.\, a = unload(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)]
\end{aligned}
$$

$$
\begin{aligned}
&BoxIn(b,c,do(a,s)) \equiv \Phi_{BoxIn}(b,c,a,s) \\
&\quad \equiv [\exists t.\, a = unload(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)] \\
&\quad\quad \vee BoxIn(b,c,s) \wedge \neg [\exists t.\, a = load(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)]
\end{aligned}
$$

$$
\begin{aligned}
&TruckIn(t,c,do(a,s)) \equiv \Phi_{TruckIn}(t,c,a,s) \\
&\quad \equiv [\exists c_1.\, a = drive(t,c) \wedge TruckIn(t,c_1,s)] \\
&\quad\quad \vee TruckIn(t,c,s) \wedge \neg [\exists c_1.\, a = drive(t,c) \wedge TruckIn(t,c_1,s)]
\end{aligned}
$$

While the notation might seem a bit cumbersome, the meaning of the axioms is quite intuitive. For example, the successor state axiom for $BoxOn(b,t,\cdot)$ states that a box b is on a truck t after an action *iff* the action loaded box b on truck t or box b was already on truck t to begin with and the action did not unload it.

3.2.3 Regression

An important tool in the development of first-order MDPs is the ability to take a first-order state description ψ and "backproject" it through a deterministic action to see what conditions must have held prior to executing the action if ψ holds after executing the action. This is precisely the definition of *regression*. Fortunately, the SSAs lend themselves to a very natural specification definition of regression: if we want to regress a fluent $F(\vec{x}, do(a,s))$ through an action a, we need only replace the fluent with its equivalent pre-action formula $\Phi_F(\vec{x},a,s)$. In general, we can inductively define a regression operator $Regr(\cdot)$ for all first-order formulae as follows (Reiter, 2001):

- $Regr(F(\vec{x}, do(a,s))) = \Phi_F(\vec{x},a,s)$
- $Regr(\neg\psi) = \neg Regr(\psi)$
- $Regr(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_2)$
- $Regr((\exists x)\psi) = (\exists x) Regr(\psi)$

Using the unique names assumption for actions and these regression rules, we can perform regression on any first-order logic formula. For example, if

$$\exists b.\, BoxIn(b, paris, do(unload(b^*, t^*), s))$$

holds then we can use the regression operator to determine what must have held in the pre-action situation s. Following is a derivation using the above rules:

$$
\begin{aligned}
&Regr(\exists b.\, BoxIn(b, paris, do(unload(b^*, t^*), s))) \\
&\quad = \exists b.\, Regr(BoxIn(b, paris, do(unload(b^*, t^*), s))) \\
&\quad = \exists b.\, \Phi_{BoxIn}(b, paris, unload(b^*, t^*), s) \\
&\quad = \exists b.\, \big[\, [\exists t.\, unload(b^*, t^*) = unload(b, t) \wedge BoxOn(b, t, s) \wedge TruckIn(t, paris, s)] \\
&\qquad \vee BoxIn(b, paris, s) \\
&\qquad\quad \wedge \neg\, [\exists t.\, unload(b^*, t^*) = load(b, t) \wedge BoxIn(b, paris, s) \wedge TruckIn(t, paris, s)] \,\big]
\end{aligned}
$$

At this point, we can use the unique names axioms for actions to simplify, and exploit rules for distributing quantifiers and renaming variables with respect to equality to obtain the following equivalent representation:

$$
\begin{aligned}
&= [\exists b, t.\, b = b^* \wedge t = t^* \wedge BoxOn(b, t, s) \wedge TruckIn(t, paris, s)] \\
&\quad \vee \exists b.\, BoxIn(b, paris, s) \\
&= [(\exists b.b = b^*) \wedge (\exists t.t = t^*) \wedge BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] \\
&\quad \vee \exists b.\, BoxIn(b, paris, s)
\end{aligned}
$$

We will assume throughout the rest of this article that all object domains are non-empty. [2] This leads to the following fully simplified form of the regression:

$$
\begin{aligned}
&Regr(\exists b.\, BoxIn(b, paris, do(unload(b^*, t^*), s))) \\
&\qquad = [BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] \vee \exists b.\, BoxIn(b, paris, s)
\end{aligned}
\tag{18}
$$

This final result is very intuitive: it states that if there exists a box b in $paris$ after unloading some box b^* from some truck t^*, then either the truck t^* was in $paris$, or a box was in $paris$ to begin with.

3.3 FOMDP Representation

Having defined the deterministic situation calculus translation of a simple PDDL model, we use this as a building block to obtain a first-order MDP (FOMDP) (Boutilier et al., 2001) from the restricted PPDDL syntax for relational MDPs that we introduced earlier. A FOMDP can be thought of as a universal MDP that abstractly defines the state, action, transition, and reward tuple $\langle S, A, T, R\rangle$ for all possible domain instantiations (i.e., an infinite number of ground MDPs). In this subsection we formalize the building blocks of FOMDPs. We begin by introducing the *case* notation and operations and discuss the representation of the reward and value function as case statements. Then we describe how stochastic actions are represented by building on our previous situation calculus formalization. Once all

[2] Logically, this requires a background theorem axiom for every object type $Sort$ that states $\exists o.\ Sort(o)$. With this, we can use the simplification $(\exists Sort : o.\ o = o^*) \supset \top$.

of these components are defined, we will have everything needed to generalize the dynamic programming solution of MDPs from the ground case to the lifted case of symbolic dynamic programming for FOMDPs.

3.3.1 Case Representation of Rewards, Values, and Probabilities

We introduce two useful variants of a *case notation* along with its logical definition to allow first-order specifications of the rewards, probabilities, and values required for FOMDPs:

$$(t \;=\; case[\phi_1, t_1; \cdots; \phi_n, t_n]) \;\equiv\; \left(t \;=\; \begin{array}{|c|} \hline \phi_1 : t_1 \\ \hline : \;\; : : \\ \hline \phi_n : t_n \\ \hline \end{array} \right)$$
$$\equiv \left(\bigvee_{i \le n} \{\phi_i \wedge t = t_i\} \right) \tag{19}$$

Here the ϕ_i are *state formulae* where fluents in these formulae do not contain the term do and the t_i are terms. We note that in contrast to states, situations reflect the entire history of action occurrences. However, the specification of our FOMDP dynamics is Markovian and allows recovery of state properties from situation terms. For this reason, we can always represent the situation term using the free variable s without loss of generality. Often the t_i will be numerical constants and the ϕ_i will partition state space.

We emphasize that the case notation for a logical formula (whether in the syntactic form $t = case[\phi_1, t_1; \cdots; \phi_n, t_n]$ or in the tabular form above) is simply a meta-logical notation used as a compact representation of the logical formula itself. In the meta-logical notation of cases, all formulae ϕ_i, terms t_i and parameters of the case statement such as the situation term s refer to symbols of the underlying logical language. At a meta-logical level, a case statement may be viewed as a relation since the case "partition" formulae may overlap and may not be exhaustive. Case statements may be compared with (in)equalities and manipulated with arithmetic operations to produce other case statements (all at a meta-logical level).

To illustrate this notation concretely, we represent our BOXWORLD FOMDP reward function $R(s)$ from our PPDDL representation in Figure 2 as the following $rCase(s)$ statement that reflects the immediate reward obtained in situation s:

$$rCase(s) = \begin{array}{|l|} \hline \exists b. BoxIn(b, paris, s) \;\; : 10 \\ \hline \neg\exists b. BoxIn(b, paris, s) : \;\; 0 \\ \hline \end{array} \tag{20}$$

For simplicity of presentation, we will assume the reward is not action depen-

dent, but such dependencies can be introduced without difficulty. Throughout the text, $R(s)$ will be used to represent a generic FOMDP reward case statement and $rCase(s)$ will refer to the specific reward function. Thus, for BOXWORLD, we write $R(s) = rCase(s)$ and wherever $R(s)$ occurs, we can substitute the logical formula for $rCase(s)$.

Here we see that the first-order formulae in the case statement divide all possible ground states into two regions of constant-value: when there exists a box in $paris$, a reward of 10 is achieved, otherwise a reward of 0 is achieved. Likewise the value function $V(s)$ that we derive through symbolic dynamic programming can be represented in exactly the same case format. Indeed, $V^0(s) = R(s)$ in the first-order version of value iteration.

The case representation can also be used to specify transition probabilities (as we will see below). We first discuss the operations that can be performed on case statements.

3.3.2 Case Operations

In this subsection we introduce various operations that can be applied to case statements providing both a formal logical definition and a graphical example that intuitively demonstrates the application of the case operation.

We begin by formally introducing the following binary $\otimes$, $\oplus$, and $\ominus$ operators on case statements (Boutilier et al., 2001):

$$case[\phi_i, t_i : i \leq n] \otimes case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i \cdot v_j : i \leq n, j \leq m] \quad (21)$$
$$case[\phi_i, t_i : i \leq n] \oplus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i + v_j : i \leq n, j \leq m] \quad (22)$$
$$case[\phi_i, t_i : i \leq n] \ominus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i - v_j : i \leq n, j \leq m] \quad (23)$$

Intuitively, to perform an operation on case statements, we simply perform the corresponding operation on the cross-product of all case partitions of the operands. Letting each ϕ_i and ψ_j denote generic first-order formulae, we can perform the "cross-sum" $\oplus$ of case statements in the following manner:

$\phi_1 : 10$
$\phi_2 : 20$

$\oplus$

$\psi_1 : 1$
$\psi_2 : 2$

$=$

$\phi_1 \wedge \psi_1 : 11$
$\phi_1 \wedge \psi_2 : 12$
$\phi_2 \wedge \psi_1 : 21$
$\phi_2 \wedge \psi_2 : 22$

Likewise, we can perform $\ominus$, $\otimes$, and $\max$ operations by, respectively, subtracting, multiplying, or taking the max of partition values. Note that for a binary operation involving a scalar and a case statement, a scalar value C may be viewed as

$case[\top, C]$ where $\top$ is a tautology. We use the $\oplus$ and $\otimes$ operators to, respectively, denote summations and products of multiple case operands.

It is important to note that some partitions resulting from the application of the $\oplus$, $\ominus$, and $\otimes$ operators may be inconsistent; if we can identify such inconsistency, we simply discard such partitions. When the case partitions contain general first-order logic formulae, inconsistency detection is undecidable. However, for the symbolic dynamic programming algorithm discussed in this section, it is not required that all inconsistent partitions be discarded; failing to do so simply results in a non-minimal case representation that contains partitions not corresponding to any world state. In practice, we rely on time-limited incomplete theorem proving for inconsistency pruning.

We define a few additional operations on case statements, the first being the binary $\cup$ operation:

$$case[\phi_i, t_i : i \leq n] \cup case[\psi_j, v_j : j \leq m] = case[\phi_1, t_1; \cdots; \phi_n, t_n; \psi_1, v_1; \cdots; \psi_m, v_m] \tag{24}$$

In this operation we simply construct the union of the partitions from each of the case statements; for example:

$$\begin{array}{|c|}\hline \phi_1 : 10 \\ \hline \phi_2 : 20 \\ \hline \end{array} \quad \cup \quad \begin{array}{|c|}\hline \psi_1 : 1 \\ \hline \psi_2 : 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|}\hline \phi_1 : 10 \\ \hline \phi_2 : 20 \\ \hline \psi_1 : \ 1 \\ \hline \psi_2 : \ 2 \\ \hline \end{array}$$

Next we define two unary operations. The $\exists \vec{x}.\, case(\vec{x})$ operation simply existentially quantifies the $case(\vec{x})$ statement. Since $case(\vec{x})$ is defined logically with a disjunction, we can can distribute the $\exists \vec{x}$ inside the disjunction:

$$\begin{aligned} \exists \vec{x}. \left(t \ = \ \begin{array}{|c|}\hline \phi_1(\vec{x}) : t_1 \\ \hline : \qquad : \ : \\ \hline \phi_n(\vec{x}) : t_n \\ \hline \end{array} \right) &\equiv \exists \vec{x}. \bigvee_{i \leq n} \{\phi_i(\vec{x}) \wedge t = t_i\} \\ &\equiv \bigvee_{i \leq n} \{\exists \vec{x}. \, \phi_i(\vec{x}) \wedge t = t_i\} \\ &\equiv \left(t \ = \ \begin{array}{|c|}\hline \exists \vec{x}. \, \phi_1(\vec{x}) : t_1 \\ \hline : \qquad : \ : \\ \hline \exists \vec{x}. \, \phi_n(\vec{x}) : t_n \\ \hline \end{array} \right) \end{aligned} \tag{25}$$

Normally we assume an implicit "$t =$" for a $case$ statement but show it above for logical clarity.

The second unary operation is denoted "casemax" (and not "$\max$") since it produces a case statement as opposed to a single numerical value. The result of casemax is a case statement where the maximal possible value of its case argument is assigned to each region of state space in the resulting case statement. Assuming that the case partitions are pre-sorted such that $t_i > t_{i+1}$ and all partitions of equal value have been disjunctively merged we can formally define this operation as follows:

$$\text{casemax } case[\phi_1, t_1; \cdots; \phi_n, t_n] \;=\; case[\phi_i \wedge \bigwedge_{j<i} \neg\phi_j, t_i : i \leq n] \qquad (26)$$

Following is a more intuitive graphical exposition of the same casemax operation:

$$\text{casemax} \begin{array}{|c|}\hline \phi_1 : t_1 \\ \hline \phi_2 : t_2 \\ \hline \vdots \;:\; \vdots \\ \hline \phi_n : t_n \\ \hline \end{array} = \begin{array}{|l|}\hline \phi_1 \qquad\qquad\qquad\qquad\qquad : t_1 \\ \hline \phi_2 \wedge \neg\phi_1 \qquad\qquad\qquad\quad : t_2 \\ \hline \vdots \qquad\qquad\qquad\qquad\qquad\quad : \vdots \\ \hline \phi_n \wedge \neg\phi_1 \wedge \neg\phi_2 \wedge \cdots \wedge \neg\phi_{n-1} : t_n \\ \hline \end{array}$$

One can easily verify that if the partitions are sorted from the highest value t_1 to the lowest t_n, then the highest value consistent with any state formula in the input case statement is assigned to the unique partition consistent with that state formulae in the resulting case statement. (If the ϕ_i in the input are mutually exclusive, then the casemax results in a case statement logically equivalent to the original.) The application of casemax requires constructing new partition formulae, up to n times the length of the original formulae for a case statement with n partitions. Fortunately, the use of inconsistency detection discussed previously and first-order ADDs (FOADD) that we introduce in the next section will mitigate the impact of this blowup by respectively pruning inconsistent case partitions and simplifying the representation of case formulae.

It is important to point out that all of the case operators are purely symbolic in that the t_i case partition values are not necessarily restricted to constant numerical values, but can be arbitrary symbolic (possibly state-dependent) terms (Boutilier et al., 2001). However, the casemax operator (as defined here) implicitly requires an ordering on the t_i. We assume for the rest of this section that the case values are numeric rather than symbolic, and apply the natural $<$ operator for our ordering.

3.3.3 Stochastic Actions and Transition Probabilities

To state the FOMDP transition function for an action, stochastic "agent" actions are decomposed into a *collection* of deterministic actions, each corresponding to a possible outcome of the stochastic action. We then use a case statement to specify a distribution according to which "Nature" may choose a deterministic action from this set whenever the stochastic action is executed. As a consequence we need only

formulate SSAs using the deterministic *Nature's choices* (Bacchus et al., 1995; Poole, 1997; Boutilier et al., 2000; Reiter, 2001), thus obviating the need for a special treatment of stochastic actions in SSAs.

Letting $A(\vec{x})$ be a stochastic action with Nature's choices (i.e., deterministic actions) $n_1(\vec{x}), \cdots, n_k(\vec{x})$, we represent the probability of $n_i(\vec{x})$ given $A(\vec{x})$ is executed in s by $P(n_j(\vec{x}), A(\vec{x}), s)$. Continuing with the translation of our simple PPDDL example, we note that the $load(b,t)$ action has one set of effects that occurs with probability 0.9. We use the deterministic action $loadS(b,t)$ to denote the successful occurrence of these effects, and we let the deterministic action $loadF(b,t)$ denote the failure of these effects to execute. To do this, we must redefine our SSAs from the previous PDDL case: now $load(b,t)$ is a stochastic action executed by the agent with $loadS(b,t)$ and $loadF(b,t)$ being possible outcomes (i.e., deterministic actions chosen by Nature). Similarly, we interpret the other two actions using $unloadS(b,t)$/$unloadF(b,t)$ as the two deterministic outcomes for $unload(b,t)$, and $driveS(t,c)$/$driveF(t,c)$ as the two deterministic outcomes for $drive(t,c)$. For completeness and correctness, we redefine our SSAs for BOXWORLD in terms of these new deterministic actions for the BOXWORLD FOMDP:

$$
\begin{aligned}
&BoxOn(b,t,do(a,s)) \equiv \Phi_{BoxOn}(b,t,a,s)\\
&\quad\equiv [\exists c.\, a = loadS(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)]\\
&\qquad \vee BoxOn(b,t,s) \wedge \neg [\exists c.\, a = unloadS(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)]
\end{aligned}
$$

$$
\begin{aligned}
&BoxIn(b,c,do(a,s)) \equiv \Phi_{BoxIn}(b,c,a,s)\\
&\quad\equiv [\exists t.\, a = unloadS(b,t) \wedge BoxOn(b,t,s) \wedge TruckIn(t,c,s)]\\
&\qquad \vee BoxIn(b,c,s) \wedge \neg [\exists t.\, a = loadS(b,t) \wedge BoxIn(b,c,s) \wedge TruckIn(t,c,s)]
\end{aligned}
$$

$$
\begin{aligned}
&TruckIn(t,c,do(a,s)) \equiv \Phi_{TruckIn}(t,c,a,s)\\
&\quad\equiv [\exists c_1.\, a = driveS(t,c) \wedge TruckIn(t,c_1,s)]\\
&\qquad \vee TruckIn(t,c,s) \wedge \neg [\exists c_1.\, a = driveS(t,c) \wedge TruckIn(t,c_1,s)]
\end{aligned}
$$

Here, we have simply replaced our previous deterministic action names from the PDDL version with the deterministic *success* versions of Nature's choice actions that we will use in our FOMDP. Note that since the "failure" versions of the actions correspond to the "no effects" case, they obviously do not play any role in the SSAs. The frame assumption present in the SSAs ensures that what was not explicitly changed remains the same.

We can now specify a distribution $P(n_j(\vec{x}), A(\vec{x}), s)$ over Nature's choice deterministic outcome using case statements to specify families of distributions, where the partitions in the case statements correspond to different classes of states and stochastic action parameters on which the distributions are conditioned. We denote specific instances of $P(n_j(\vec{x}), A(\vec{x}), s)$ with the case statement $pCase(n_j(\vec{x}), A(\vec{x}), s)$

where $\top$ is a tautology, for example:

$$pCase(loadS(b,t), load(b,t), s) = \boxed{\top : 0.9}$$
$$pCase(loadF(b,t), load(b,t), s) = \boxed{\top : 0.1}$$

$$pCase(unloadS(b,t), unload(b,t), s) = \boxed{\top : 0.9} \quad (27)$$
$$pCase(unloadF(b,t), unload(b,t), s) = \boxed{\top : 0.1} \quad (28)$$
$$pCase(driveS(b,t), drive(b,t), s) = \boxed{\top : 1.0}$$
$$pCase(driveF(b,t), drive(b,t), s) = \boxed{\top : 0.0}$$

The above axiomatization does not fully illustrate the power of the FOMDP representation in that the probabilities are not state or action dependent, so we briefly digress to demonstrate a slightly more interesting variant. Suppose that the success of driving a truck to a city depends on whether the truck contains a box b with volatile material denoted by the predicate $Volatile(b)$. Then we can specify the family of distributions over Nature's choices for this stochastic action as follows:

$$pCase(\ driveS(t,c), drive(t,c), s\ ) = \boxed{\begin{array}{l} \exists b.BoxOn(b,t,s) \wedge Volatile(b) \quad : 0.9 \\ \neg(\exists b.BoxOn(b,t,s) \wedge Volatile(b)) : 1.0 \end{array}}$$

$$pCase(\ driveF(t,c), drive(t,c), s\ ) = \boxed{\begin{array}{l} \exists b.BoxOn(b,t,s) \wedge Volatile(b) \quad : 0.1 \\ \neg(\exists b.BoxOn(b,t,s) \wedge Volatile(b)) : 0.0 \end{array}}$$

Here we see the transition probability of $drive(t,c)$ can be easily conditioned on state properties of s and action parameters t and c.

It is important to note that the probabilities over all deterministic Nature's choices for a stochastic action sum to one:

$$\bigoplus_{j=1}^{k} P(n_j(\vec{x}), A(\vec{x}), s) = \boxed{\top : 1}\ ;\ \forall \vec{x},\ s$$

In addition, each $P(n_j(\vec{x}), A(\vec{x}), s)$ should be a disjoint partitioning of state space such that no two case partitions ambiguously assign multiple probabilities to the same state. These two properties are crucial to having a well-defined probability distribution over all possible deterministic action outcomes for every possible state.

For this last example, the second property can be easily verified:

$$pCase(\ driveS(t,c), drive(t,c), s\ ) \oplus pCase(\ driveF(t,c), drive(t,c), s\ ) = \boxed{\top : 1}$$

3.4 Symbolic Dynamic Programming (SDP)

Symbolic dynamic programming (SDP) (Boutilier et al., 2001) is a dynamic programming solution to FOMDPs that produces a logical case description of the optimal value function. This is achieved through the symbolic operations of first-order decision-theoretic regression and maximization that perform the traditional dynamic programming Bellman backup at an abstract level without explicit enumeration of either the state or action spaces of the FOMDP. Among many possible applications, the use of SDP leads directly to a domain-independent value iteration solution to FOMDPs.

We will assume a constant numerical representation of values in order to explicitly perform the casemax during SDP in this article. However, we note that an appropriate generalization of casemax (c.f., Chapter 6 of Sanner (2008)) along with $Regr$ of functional fluents (Reiter, 2001) allows the definitions covered here to apply to general symbolic value representations using general terms rather than constants, hence the original use of "symbolic" in the name of the SDP algorithm.

3.4.1 First-order Decision-theoretic Regression

Suppose we are given a value function $V(s)$. The first-order decision-theoretic regression (FODTR) (Boutilier et al., 2001) of this value function through an action $A(\vec{x})$ yields a case statement containing the logical description of states and values that would give rise to $V(s)$ after doing action $A(\vec{x})$. This is analogous to classical goal regression, the key difference being that action $A(\vec{x})$ is stochastic. In MDP terms, the result of FODTR roughly corresponds to a Q-function (albeit one with free variables for the action parameters), which corresponds to the first half of a Bellman backup operation given in Equation 6. [3]

We define the ***first-order decision theoretic regression (FODTR)*** as the situation calculus analog of Equation 6 where we note that different successor states only arise through different Nature's choice deterministic actions:

$$FODTR[V(s), A(\vec{x})] = R(s) \oplus \gamma \cdot \left[\bigoplus_{j=1}^{k} \{ P(n_j(\vec{x}), A(\vec{x}), s) \otimes V(do(n_j(\vec{x}), s)) \} \right] \tag{29}$$

$FODTR$ uses a meta-logical notation that takes as arguments $V(s)$ representing the logical case statement for a value function with situation variable s and a parame-

[3] We do not use an action dependent reward $R(s, A(\vec{x}))$, but could substitute it if needed.

terized stochastic action term $A(\vec{x})$ with free variables $\vec{x}$. All subsequently defined operations on case statements in this article will be defined analogously.

The only problem with the $FODTR[V(s), A(\vec{x})]$ operation as currently defined is that the formula $V(do(n_j(\vec{x}), s))$ refers not to the current situation s, but to the future situation $do(n_j(\vec{x}), s)$, but this is easily remedied with regression:

$$FODTR[V(s), A(\vec{x})] = \\ R(s) \oplus \gamma \cdot \left[\bigoplus_{j=1}^{k} \{ P(n_j(\vec{x}), A(\vec{x}), s) \otimes Regr(V(do(n_j(\vec{x}), s))) \} \right] \quad (30)$$

This is equivalent to the $FODTR$ operation in Equation 29 since the $Regr$ operation preserves equivalence (by definition). Also on account of the equivalence preserving properties of $Regr$, we note that if $V(s)$ partitions the state space then so must the resulting case statement for $FODTR[V(s), A(\vec{x})]$. Thus, from a logical description of $V(s)$ we can derive one for its decision-theoretic regression $FODTR[V(s), A(\vec{x})]$. This is key to avoiding state and action enumeration in dynamic programming.

We denote an instance of the value function $V(s)$ by the case statement $vCase(s)$. As defined previously, we also assume that the reward function $R(s)$ and instances of Nature's choice probabilities $P(n_j(\vec{x}), A(\vec{x}), s)$ are denoted by $rCase(s)$ and $pCase(n_j(\vec{x}), A(\vec{x}), s)$, respectively.

As an example, let us compute the FODTR for $vCase(s) = rCase(s)$ through the stochastic action $A(\vec{x}) = unload(b^*, t^*)$ where $rCase(s)$ is the BOXWORLD reward as previously defined in Equation 20. Since $vCase(s)$ is logically defined, we can push the $Regr$ operator into the individual $vCase(s)$ partitions as follows:

$$FODTR[vCase(s), unload(b^*, t^*)] = rCase(s) \oplus \\ \gamma \left[\bigoplus_{j=1}^{k} \left\{ pCase(n_j(\vec{x}), unload(b^*, t^*), s) \right. \right. \\ \left. \left. \otimes \begin{array}{|l|} \hline Regr(\exists b. BoxIn(b, paris, do(n_j(\vec{x}), s))) \quad : 10 \\ \hline Regr(\neg \exists b. BoxIn(b, paris, do(n_j(\vec{x}), s))) : \quad 0 \\ \hline \end{array} \right\} \right]$$

Now, since the stochastic action $A(\vec{x}) = unload(b^*, t^*)$, we know that Nature's deterministic action choices $n_j(\vec{x})$ range over $unloadS(b^*, t^*)$ and $unloadF(b^*, t^*)$. We now substitute the $pCase$ definitions for the deterministic actions $unloadS(b^*, t^*)$

and $unloadF(b^*, t^*)$ from Eqs. 27 and 28, respectively, obtaining:

$$FODTR[vCase(s), unload(b^*, t^*)] = rCase(s) \oplus$$
$$\gamma \left[\left\{ \boxed{\top : 0.9} \otimes \begin{array}{|l:r|}\hline Regr(\exists b.BoxIn(b, paris, do(unloadS(b^*, t^*), s))) & : 10 \\ \hline Regr(\neg\exists b.BoxIn(b, paris, do(unloadS(b^*, t^*), s))) & : \ 0 \\ \hline \end{array} \right\} \right.$$
$$\left. \oplus \left\{ \boxed{\top : 0.1} \otimes \begin{array}{|l:r|}\hline Regr(\exists b.BoxIn(b, paris, do(unloadF(b^*, t^*)))) & : 10 \\ \hline Regr(\neg\exists b.BoxIn(b, paris, do(unloadF(b^*, t^*)))) & : \ 0 \\ \hline \end{array} \right\} \right]$$

We have already computed $Regr(\exists b.BoxIn(b, paris, do(unloadS(b^*, t^*))))$ from Equation 18 where the deterministic $unload(b^*, t^*)$ from the PDDL case has been renamed to $unloadS(b^*, t^*)$. And by the properties of $Regr$, we know that $Regr(\neg\phi) = \neg Regr(\phi)$ so we can easily obtain the regression of the negated partition in $rCase(s)$. It is important to note that if $rCase(s)$ partitioned the post-action state space, the $Regr$ operator preserves this partitioning in the pre-action state space. We note that

$$Regr(\phi(\vec{x}, do(unloadF(b^*, t^*)))) = \phi(\vec{x}, s)$$

can be easily derived since $unloadF(b^*, t^*)$ has no effects and is thus equivalent to a $noop$ action. Making these substitutions, explicitly multiplying in the action probabilities and discount factor $\gamma = 0.9$, and explicitly writing out $rCase(s)$, we obtain the following (where, for readability, we use $\neg$" to denote the conjunction of the negation of *all* partitions above the given partition in the case statement):

$$FODTR[vCase(s), unload(b^*, t^*)] = \begin{array}{|l:r|}\hline \exists b.BoxIn(b, paris, s) & : 10 \\ \hline \neg\text{"} & : \ 0 \\ \hline \end{array}$$
$$\oplus \begin{array}{|l:r|}\hline [BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & \\ \vee\exists b.\ BoxIn(b, paris, s) & : 8.1 \\ \hline \neg\text{"} & : \ 0 \\ \hline \end{array}$$
$$\oplus \begin{array}{|l:r|}\hline \exists b.BoxIn(b, paris, s) & : 0.9 \\ \hline \neg\text{"} & : \ 0 \\ \hline \end{array}$$

Finally, explicitly carrying out the $\oplus$'s and simplifying yields the final result:

$$FODTR[vCase(s), unload(b^*, t^*)]$$
$$= \begin{array}{|l:r|}\hline \exists b.\ BoxIn(b, paris, s) & : 19.0 \\ \hline \neg\text{"} \wedge [BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : \ 8.1 \\ \hline \neg\text{"} & : \ 0 \\ \hline \end{array} \quad (31)$$

The case statement resulting from FODTR contains free variables for the action parameters; in this case $A(\vec{x}) = unload(b^*, t^*)$ so the free parameters are b^* and t^*. This result is intuitive: it states that if a box was already in $paris$ then we get reward 19 (10 for the current reward and 9 for the discounted 1-step reward). Otherwise, if a box is not in $paris$ in the current state, but box b^* was on truck t^* in $paris$ and the action was specifically $unload(b^*, t^*)$, then we get an expected future reward of 8.1 taking into account the success probability of unloading the box and the discount factor. Finally, if no box is in $paris$ in the current state and we do not unload a box then we get 0 total reward.

This case statement represents the value of taking stochastic action $unload(b^*, t^*)$ and acting so as to obtain the value given by $rCase(s)$ thereafter. However, what we really need for symbolic dynamic programming is a logical description of a Q-function (recall Equation 6) that tells us the possible values that can be achieved for *any* action instantiation of b^* and t^*. This leads us to the following definition $Q(A, s)$ of a first-order Q-function that makes use of the previously defined $\exists \vec{x}$ unary case operator:

$$Q^t(A, s) = \exists \vec{x}.\, FODTR[V^{t-1}(s), A(\vec{x})] \tag{32}$$

We denote a specific instance of $Q^t(A, s)$ by the case statement $qCase^t(s, A)$. We can think of $qCase^t(s, A)$ as a logical description of the Q-function for action $A(\vec{x})$ indicating the values that could be achieved by *any* instantiation of $A(\vec{x})$. By using the first-order case representation of states as well as action quantification via the $\exists \vec{x}$ operation, FODTR effectively achieves *both action and state abstraction*.

Letting $vCase^0(s) = rCase(s)$, we can continue our running example to obtain a Q-function description for action $unload$ where we have removed vacuous quantifiers. Technically, $qCase^1(unload, s)$ would not be an exhaustive partitioning of the state space in that the 0 value partition from Equation 32 is not the same one implied here from the $\neg$" because the partition formulae above it have been quantified. However, throughout this article, we can exploit our assumption that all FOMDPs have a $noop$ action to assume that the minimum value for any state is 0 (as opposed to being undefined). Thus we can always show the final 0 partition as $\neg$" to indicate that any partitions not explicitly assigned a value by the above partitions are assigned a default value 0. Thus, we arrive at the following intuitive result:

$$qCase^1(unload, s) \;=\; \exists b^*, t^*.\, FODTR[vCase^0(s), unload(b^*, t^*)]$$

$$= \begin{array}{|l:r|} \hline \exists b.\, BoxIn(b, paris, s) & : 19.0 \\ \hline \exists b^*, t^*.\, [\neg" \wedge BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : \;\; 8.1 \\ \hline \neg" & : \quad 0 \\ \hline \end{array}$$

In words, this states if the box was already in $paris$ then we get a discounted reward of 19. Otherwise, if a box is not in $paris$ in the current state, but there *exists* some

box on a truck in $paris$, then we could unload it to get an expected discounted reward of 8.1. Finally, if there is no box on a truck to unload in $paris$ and there is no box already in $paris$ then we get 0 expected discounted reward. It is instructive to compare this description to the prior description of FODTR without existential action quantification—the difference is subtle, but important for action abstraction.

3.4.2 Symbolic Maximization

At this point, we can decision-theoretically regress the value function through a ***single*** stochastic action to obtain a representation of its Q-function, but to complete the dynamic programming (Bellman backup) step in the spirit of Equation 7 from Section 2, we need to know the maximum value that can be achieved by ***any*** action. For example, in the BOXWORLD FOMDP, our possible action choices are $unload(b,t)$, $load(b,t)$, and $drive(t,c)$ and our Q-function computations using Equation 32 give us $qCase^1(unload, s)$, $qCase^1(load, s)$, and $qCase^1(drive, s)$. In general, we will assume that we have m stochastic actions $\{A_1(\vec{x}_1), \ldots, A_m(\vec{x}_m)\}$ and a corresponding set of Q-functions $\{qCase^t(A_1, s), \ldots, qCase^t(A_m, s)\}$ derived from a common value function $vCase^{t-1}(s)$.

We might try to obtain a case description of the value function $vCase^t(s)$ by simply applying the case $\cup$ operator to merge all partitions of the Q-functions, i.e., $qCase^t(s, A_1) \cup \ldots \cup qCase^t(s, A_m)$. While this provides us with a description of possible values, it is not a value ***function*** because the state spaces of each Q-function may overlap, thus potentially assigning multiple values to the same underlying state. What we really want instead is to assign the ***highest*** possible value to each portion of state space. Fortunately, this is quite easy with the casemax operator. Thus we get the following equation for the symbolic maximization of Q-functions:

$$V^t(s) = \text{ casemax } \left[Q^t(A_1, s) \cup \ldots \cup Q^t(A_m, s)\right] \tag{33}$$

Recalling the way in which the casemax operation is computed from Equation 26, every resulting instance $vCase^t(s)$ of the value function $V^t(s)$ will have the following case statement format where value case partition ψ_j corresponds to value v_j and $v_i > v_{i+1}$:

$$vCase^t(s) \quad = \quad \begin{array}{|l:l|} \hline \psi_1 & : v_1 \\ \hline \psi_2 \wedge \neg\psi_1 & : v_2 \\ \hline \vdots & : \ \vdots \\ \hline \psi_n \wedge \neg\psi_1 \wedge \neg\psi_2 \wedge \cdots \wedge \neg\psi_{n-1} & : v_n \\ \hline \end{array}$$

This approach effectively gives us a decision-list representation of our value function (recall the optimal value function representation from Figure 4). Thus, to determine the value for a state, we can simply traverse the list from highest to lowest

value and take the value for the first case partition that is satisfied. The casemax operation guarantees that this value function will be a disjoint partitioning of the state space and our previous assumption that all actions are executable in all states ensures that this value function exhaustively assigns a value to all possible states (assuming $vCase^{t-1}$ was exhaustive).

3.4.3 First-order Value Iteration

One should note that the SDP equations given here are exactly the lifted versions of the classical dynamic programming solution to MDPs given previously in Equations 6 and 7 from Section 2. Since those equations were used in part to define a value iteration algorithm, we can use the lifted versions to define a ***first-order value iteration*** algorithm where ϵ is our error tolerance:

(1) Initialize $V^0(s) = R(s),\ t = 1$.
(2) Compute $V^t(s)$ given $V^{t-1}(s)$ using Equations 32 and 33.
(3) If the following Bellman error inequality holds

$$\|V^t(s) \ominus V^{t-1}(s)\|_\infty \leq \frac{\epsilon(1-\gamma)}{2\gamma} \tag{34}$$

then terminate and return $V^t(s)$, otherwise go to step 2.

Here, we define $\|V^t(s) \ominus V^{t-1}(s)\|_\infty$ as the maximal absolute value of any consistent partition in the case statement resulting from $V^t(s) \ominus V^{t-1}(s)$.

For example, applying first-order value iteration to the 0-stage-to-go value function (i.e., $vCase^0(s) = rCase(s)$, given previously in Equation 20) yields the following simplified 1- and 2-stage-to-go value functions in the BOXWORLD problem domain:

$vCase^1(s) =$

$\exists b.BoxIn(b, paris, s)$	: 19.0
$\neg$“ $\wedge\ \exists b, t.TruckIn(t, paris, s) \wedge BoxOn(b, t, s)$	: 8.1
$\neg$“	: 0.0

$vCase^2(s) =$

$\exists b.BoxIn(b, paris, s)$	: 26.1
$\neg$“ $\wedge\ \exists b, t.TruckIn(t, paris, s) \wedge BoxOn(b, t, s)$	: 15.4
$\neg$“ $\wedge\ \exists b, c, t.BoxOn(b, t, s) \wedge TruckIn(t, c, s)$	: 7.3
$\neg$“	: 0.0

After sufficient iterations of first-order value iteration, the t-stage-to-go value function converges, giving the optimal value function (and as we derive in a moment, an optimal policy) from Figure 4.

Boutilier et al. (2001) provide a proof that SDP and thus every step of value iteration produces a correct logical description of the value function. From this, we can lift the error bounds from the ground MDP case in Equation 8 to show domain-independent error bounds on the first-order abstracted value estimate:

Corollary 3.4.1 *Let $V^*(s)$ be the optimal value function for a FOMDP. Terminating according to the criteria given in Step 3 of first-order value iteration guarantees $\|V^t(s) - V^*(s)\|_\infty < \epsilon$ for* any *domain instantation (even infinite) of the FOMDP.*

More generally, as a direct result of this corollary, we can derive domain-independent error bounds for the first-order representation of the value function produced by *any* first-order MDP algorithm (see Section 6 for other first-order algorithms).

Corollary 3.4.2 *Let $\hat{V}(s)$ be an arbitrary first-order case representation of a value function. Let $\hat{V}'(s)$ be the result of applying Equations 32 and 33 to $\hat{V}(s)$ for a FOMDP. Let $\epsilon = \frac{2}{1-\gamma}\|\hat{V}'(s) \ominus \hat{V}(s)\|_\infty$. Then $\|\hat{V}(s) - V^*(s)\|_\infty < \epsilon$ for* any *domain instantiation of the FOMDP.*

The difference of γ between the bounds of Corollaries 3.4.1 and 3.4.2 occurs because the former refers to a bound on $V^t(s)$, while the latter refers to a bound on $\hat{V}(s) = V^{t-1}(s)$ and value iteration is known to contract the error by γ on each iteration.

3.4.4 Policy Representation

Given a value function, it is important to be able to derive a first-order greedy policy representation from it, just as we did in the ground case in Section 2. This policy can then be used to directly determine actions to apply when acting in a ground instantiation of the FOMDP, or it can be used to define first-order versions of (approximate) policy iteration (Sanner and Boutilier, 2006).

Fortunately, given a value function $V(s)$, it is easy to derive a greedy policy from it. Assuming we have m parameterized actions $\{A_1(\vec{x}), \ldots, A_m(\vec{x})\}$, we can formally derive the policy $\pi(s)[\cdot]$ using the $\cdot$ to denote the value representation from which the policy is derived as follows (taking into account a few modifications to the case operators that we discuss in a moment):

$$\pi(s)[V(s)] = \text{casemax}(\bigcup_{i=1\ldots m} \exists \vec{x}.\, FODTR[V(s), A_i(\vec{x})]) \tag{35}$$

We often refer to a specific instance of $\pi(s)$ with the case statement $\pi Case(s)$. For bookkeeping, we require that each partition $\langle \phi, t \rangle$ in $\exists \vec{x}\ FODTR[V(s), A_i(\vec{x})]$ maintain a mapping to the action A_i that generated it, which we denote as $\langle \phi, t \rangle \rightarrow A_i$. Then, given a particular world state s, we can evaluate $\pi Case(s)$ to determine which maximal policy partition $\langle \phi, t \rangle \rightarrow A_i$ is satisfied by s and thus, which action A_i should be applied. If we retrieve the bindings of the existentially quantified

action variables $\exists \vec{x}$ in that satisfying policy partition, we can easily determine the parameterization of action A_i that should apply according to the policy.

To make this concrete, we derive a simple greedy policy for the BOXWORLD FOMDP assuming the value function $V(s) = rCase(s)$ and that we only have two actions $unload(b^*, t^*)$ and $noop$. Noting that we have already computed $FODTR[rCase(s), unload(b^*, t^*)]$ in Equation 31 and that $FODTR[rCase(s), noop]$ will just be $rCase(s)$ with 10 replaced by 19, we obtain the following policy:

$$
\begin{aligned}
&\pi Case[rCase(s)] \\
&= \text{casemax}(\{\exists b^*, t^*.\, FODTR[rCase(s), unload(b^*, t^*)]\} \\
&\qquad \cup \{FODTR[rCase(s), noop]\})
\end{aligned}
$$

$$
= \begin{array}{|l|l|}
\hline
\exists b.\, BoxIn(b, paris, s) & : 19.0 \longrightarrow noop \\
\hline
\neg \text{“} \wedge [\exists b^*, t^*.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : \ 8.1 \longrightarrow unload(b^*, t^*) \\
\hline
\neg \text{“} & : \quad 0 \longrightarrow noop \\
\hline
\end{array}
$$

For a more interesting policy, we refer the reader back to the optimal value function and policy for BOXWORLD given in Figure 4.

Technically, we note that there may be an infinite number of actions that can be applied since there are an infinite number of ground instantiations of $unload(b^*, t^*)$ depending on the domain instantiation. Thus, this policy representation manages to *compactly* represent the selection of an optimal action amongst an infinite set.

4 Practical FOMDP Solution Techniques

The last section reviewed a symbolic dynamic programming (SDP) algorithm theoretically capable of producing an ϵ-optimal value function for a FOMDP that does not require theorem proving to detect inconsistent case partitions or logical simplification to maintain compact representations of case partition formulae. However, in practice, both theorem proving and simplification are needed to control the representational blowup of the value function occurring at each step of value iteration.

To this end, the first half of this section introduces a practical first-order extension of the algebraic decision diagram (ADD) (Bahar et al., 1993) data structure, the *first-order ADD (FOADD)*, for maintaining case statements in a simplified, non-redundant format that facilitates theorem proving for inconsistency detection. We show how FOADDs can be used to exploit structure in SDP for FOMDPs in much the same manner that ADDs have been used to exploit structure in dynamic programming for MDPs (Hoey et al., 1999). We conclude with an illustrative empirical results demonstrating that FOADDs enable an automated solution to basic FOMDPs. We will discuss related work on first-order decision diagrams (FODDs) (Wang et al., 2008), also applied to FOMDPs, in Section 6.

In the second half, we introduce an additive decomposition approach for approximately solving FOMDPs with universal reward specifications. This approach is motivated in part by previous decomposition methods and enables the application of FOMDP solution techniques to a reward specification that otherwise renders SDP solution approaches intractable.

4.1 Representation and Solution with First-order ADDs

An *algbraic decision diagram* (ADD) (Bahar et al., 1993) is a data structure for compactly representing a function from $\mathbb{B}^n \rightarrow \mathbb{R}$ using a directed acyclic graph. ADDs have been used to compactly model transition functions, rewards, and value functions in factored MDPs (Boutilier et al., 1999). Moreover, value iteration defined in terms of ADD operations has yielded substantial improvements in time and space complexity over enumerated state representations (Hoey et al., 1999).

To extend these ideas to the first-order framework, we define methods for breaking down first-order case partition formulae into their boolean propositional components and create a compact *first-order ADD* (FOADD) representation of case statements. Then we can apply known ADD algorithms to perform the $\otimes$, $\oplus$, and $\ominus$ case operations. Once we have shown how to do this, we end with a discussion of the practical use of FOADDs and a small example of a FOADD application to SDP.

4.1.1 FOADD Construction and Operations

The first aspect of FOADDs concerns how to construct them automatically from a case representation. Since ADDs are propositional, we need some method of finding propositional structure in first-order formulae. We can do this by permuting quantifers at the same level of nesting (e.g., $[\exists x, y.\phi] \equiv [\exists y, x.\phi]$) and by distributing quantifiers as deeply into case formulae as possible using the following rewrite rule templates ($\diamond$ indicates variables other than those explicitly quantified):

$$[\exists x.\, A(x,\diamond) \vee B(x,\diamond)] \longrightarrow [(\exists x.\, A(x,\diamond)) \vee (\exists x.\, B(x,\diamond))] \quad (36)$$
$$[\forall x.\, A(x,\diamond) \wedge B(x,\diamond)] \longrightarrow [(\forall x.\, A(x,\diamond)) \wedge (\forall x.\, B(x,\diamond))] \quad (37)$$
$$[\exists x.\, A(x,\diamond) \wedge B(\diamond)] \longrightarrow [(\exists x.\, A(x,\diamond)) \wedge (B(\diamond))] \quad (38)$$
$$[\forall x.\, A(x,\diamond) \vee B(\diamond)] \longrightarrow [(\forall x.\, A(x,\diamond)) \vee (B(\diamond))] \quad (39)$$

We also perform equality simplification using the non-empty domain assumption with the following two rules:

$$[\exists x.\, x = y \wedge A(x,\diamond)] \longrightarrow A(y,\diamond) \quad (40)$$
$$[\forall x.\, x \neq y \vee A(x,\diamond)] \longrightarrow A(y,\diamond) \quad (41)$$

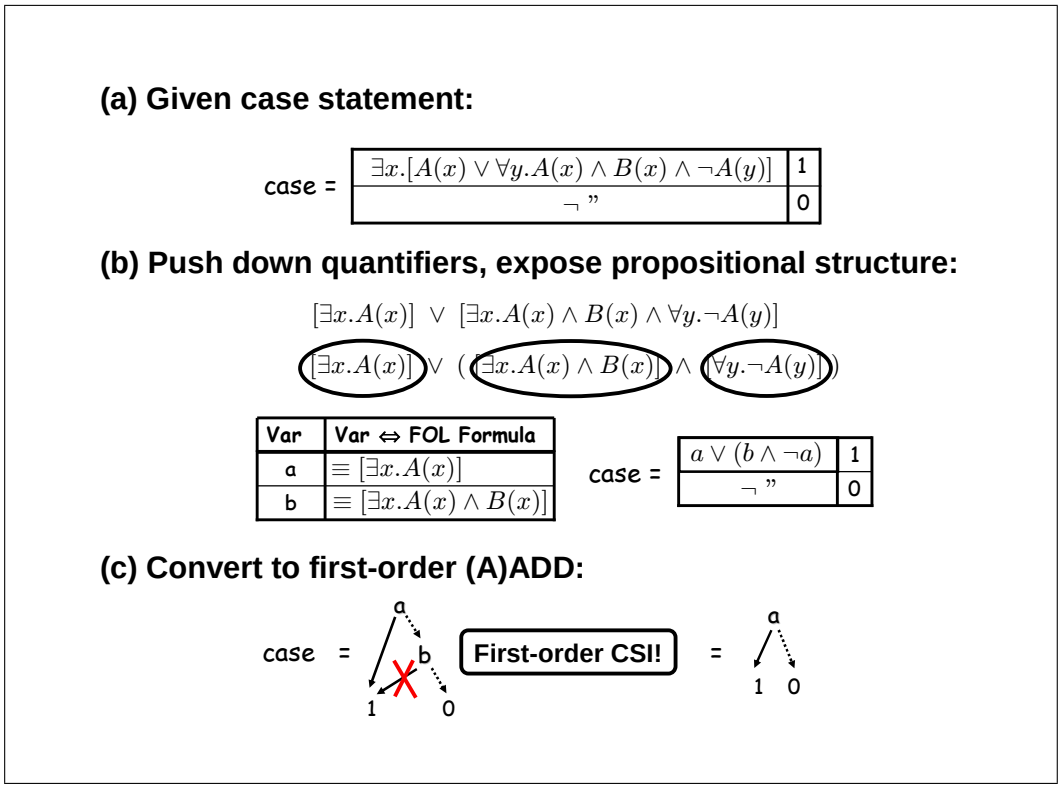

Fig. 5. An example conversion from a case statement to a compact FOADD representation demonstrating first-order CSI.

The first rule is fairly straightforward while the second rule follows simply from the negation of the first rule with renaming.

In practice, we iteratively apply simplification rules (40),(41) followed by rewrite rules (36)–(39) working from the innermost to the outermost quantifiers until no more rewrites can be applied. While other orders may give different (potentially smaller) results, we find that this deterministic approach is generally sufficient to expose most propositional structure in first-order formulae.

We provide the following example application of these rewrite and simplification rules to demonstrate their power:

$$
\begin{aligned}
& \exists x, z.\, [x = y \wedge A(x, \diamond) \wedge B(y, z)] \\
\equiv\ & \exists x.\, [x = y \wedge A(x, \diamond) \wedge (\exists z.\, B(y, z))] && \text{[Apply rewrite rule (38) for } z\text{]} \\
\equiv\ & (\exists x.\, x = y \wedge A(x, \diamond)) \wedge (\exists z.\, B(y, z)) && \text{[Apply rewrite rule (38) for } x\text{]} \\
\equiv\ & A(y, \diamond) \wedge (\exists z.\, B(y, z)) && \text{[Apply simplification rule (40) for } x\text{]}
\end{aligned}
$$

To build a FOADD, we first apply these rules to expose the propositional structure

of a first-order formula. Consider the example in Figure 5(a,b); we start with

$$\exists x.[A(x) \vee \forall y.A(x) \wedge B(x) \wedge \neg A(y)] \tag{42}$$

and apply rewrite rule (39) for y followed by (36) for x to obtain

$$[\exists x.A(x)] \vee ([\exists x.A(x) \wedge B(x)] \wedge [\forall y.\neg A(y)]). \tag{43}$$

Once we have pushed quantifiers as far down as possible, we extract the propositional structure of the formula by considering propositional connectives over quantified formulae as follows:

$$\boxed{\exists x.A(x)} \vee \left(\boxed{[\exists x.A(x) \wedge B(x)]} \wedge \boxed{\forall y.\neg A(y)} \right) \tag{44}$$

Each of these boxes represents a formula that we cannot further decompose into propositional components. Consequently, we treat each of these boxed formulae as propositions. To do this, we maintain a table of mappings from propositional variables p, naming each first-order formula, to first-order formulae ψ: $\{p \rightarrow \psi\}$. To convert a new formula ϕ in a case statement to a propositional variable, we examine each formula-to-proposition mapping in our table. If $\phi \equiv \psi$ for some ψ in the table, we return its corresponding proposition p; if $\phi \equiv \neg\psi$, we return $\neg p$; otherwise, we add a new proposition label q and add the mapping $q \rightarrow \phi$ to our table and return q. In our example, having built the table shown in Figure 5(b), we can convert the formula to its propositional counterpart:

$$a \vee (b \wedge \neg a) \tag{45}$$

At this point, we can build an ADD from a case statement whose formulae are purely propositional. What makes this ADD first-order is the additional proposition to first-order formula mapping that gives each proposition a first-order definition. Standard ADDs can exploit *context-specific independence (CSI)* (Boutilier et al., 1996) (i.e., where the value of a function is independent of an input variable given the assignment to other variables). There is, however, an additional form of CSI that we can exploit in FOADDs—*first-order CSI*. This first-order CSI follows from the structured and potentially overlapping nature of the propositional variables. For instance, in our example, $\neg a \supset \neg b$, so as we traverse its FOADD representation, we can force the decision node for b in the context of a. This is shown in Figure 5(c).

The options for detecting first-order CSI include:

(a) Do not perform any first-order CSI detection at all.
(b) Maintain information about all pairwise implications in the propositional mapping table and detect just this pairwise first-order CSI during the application of FOADD operations.

(c) Perform full simplification for all decision nodes in the context of the conjunction of all decisions made for parent nodes during all operations on the FOADD.

Obviously (a) requires no additional computation, but can give rise to FOADDs with potentially dead paths. In contrast, (c) requires substantial computation in return for extensive simplication. In practice, we find (b) to offer the most reasonable tradeoff between computation and simplification; time-limited theorem proving, although incomplete, suffices to identify many pairwise node implications that lead to substantial first-order CSI pruning. It is trivial to extend the ADD algorithm to do this additional consistency check in the presence of parent decisions when performing the standard ADD $Apply$ and $Reduce$ operations. However, if (b) or (c) are used, it is not sound to reorder the ADD nodes since the first-order context of these prunings may change and thus may no longer be valid after node reordering.

Once we convert a case statement to an FOADD, we can apply the $\otimes$, $\oplus$, and $\ominus$ case operations to FOADDs by making direct use of the ADD $Apply$ operations of multiplication, addition, and subtraction (Bahar et al., 1993). We can reuse standard ADD operations for FOADDs since they are just ADDs with augmented variable definitions in the propositional mapping table. Thus, the only practical difference between ADD and FOADD operations is that these augmented variable definitions may lead to additional pruning of structure due to first-order CSI.

In general, FOADDs may be treated as ADDs, except for the requirement to consult the propositional mapping table in the following circumstances:

(1) when constructing a FOADD;
(2) when converting a FOADD back to a case representation or evaluating a ground state; or
(3) when exploiting first-order CSI using method (b) or (c) above, we may consult this table during the ADD $Reduce$ and $Apply$ procedures.

4.1.2 Practical Considerations

Replacing case statements with FOADDs in the representation and solution of FOMDPs has the potential to exploit a great deal of structure that naturally occurs in these representations. First, the disjunctive nature of positive effects in the regression of FOMDP formulae introduces a number of disjunctions during the application of algorithms such as SDP. Second, the existential quantification of the action variables in these formulae introduce existential quantifiers that can be distributed through the disjunctions introduced by $Regr$. Consequently, every SDP step introduces structure that can be directly exploited by the previously described methods for exposing propositional structure of first-order formulae. As such, our approach to representing FOADDs is well-suited to FOMDPs as we demonstrate below with a small example.

However, if we were to define a complete SDP algorithm for FOMDPs that only uses FOADDs, we would need to define special unary FOADD operations such as $Regr$, casemax, and $\exists \vec{x}$ used in the SDP algorithm. While $Regr$ can be easily defined (note that a FOADD is just a compact representation of a case statement and thus $Regr$ can still be applied), it changes the logical meaning of the FOADD nodes since they have a first-order definition. In general, maintaining a canonical representation after performing $Regr$ on a FOADD requires expensive node reordering operations. The application of $\exists \vec{x}$ and casemax also generally require expensive node reordering operations. For these reasons, we do not apply $Regr$, casemax, or $\exists \vec{x}$ to FOADDs in practice, instead opting for a pragmatic use of FOADDs that exploits their strengths. [4]

The primary advantage of FOADDs is the provision of efficient binary operations and formula simplification through the breakdown of propositional structure and the elimination of redundancy that occurs during their construction. In doing this simplification, FOADDs remove a lot of burden from the theorem prover, which must otherwise detect inconsistency with highly redundant representations. Thus, in our SDP algorithms, we use FOADDs where they are most useful and efficient—binary operations and logical simplification—and revert to the case representation to perform the unary operations of $Regr$, casemax, and $\exists \vec{x}$ that can be expensive due to the need for internal node rotations. This approach leads to a viable SDP algorithm, to which we now turn.

4.1.3 Symbolic Dynamic Programming with FOADDs

The use of FOADDs in the somewhat hybrid manner discussed above allows the development of a practical SDP algorithm.

We have implemented a fully automated first-order value iteration algorithm and tested it on several examples to develop a sense of its effectiveness. One problem tested is the running BOXWORLD FOMDP example. The FOADDs for the reward, optimal value function and policy are given in Figure 6. For the variable ordering, we simply maintain the order of formulae as they were added to the variable mapping table in the FOADD during the SDP algorithm. We use the Vampire theorem prover (Riazanov and Voronkov, 2002) for detecting equivalence and inconsistency. The total running time for this solution until convergence within tolerance 1e-4 was 15.7s on a 2Ghz Pentium with 2Gb of RAM. Unsurprisingly, the final FOADD for this problem gives exactly the decision list structure that we would expect for the BOXWORLD problem as shown in Figure 4.

We have also used our FOADD value iteration algorithm to solve other variants

[4] While we do not discuss $Regr$, casemax, and $\exists \vec{x}$ for FOADDs further here, the reader is referred to Sanner (2008) for additional information on how one might perform these operations efficiently.

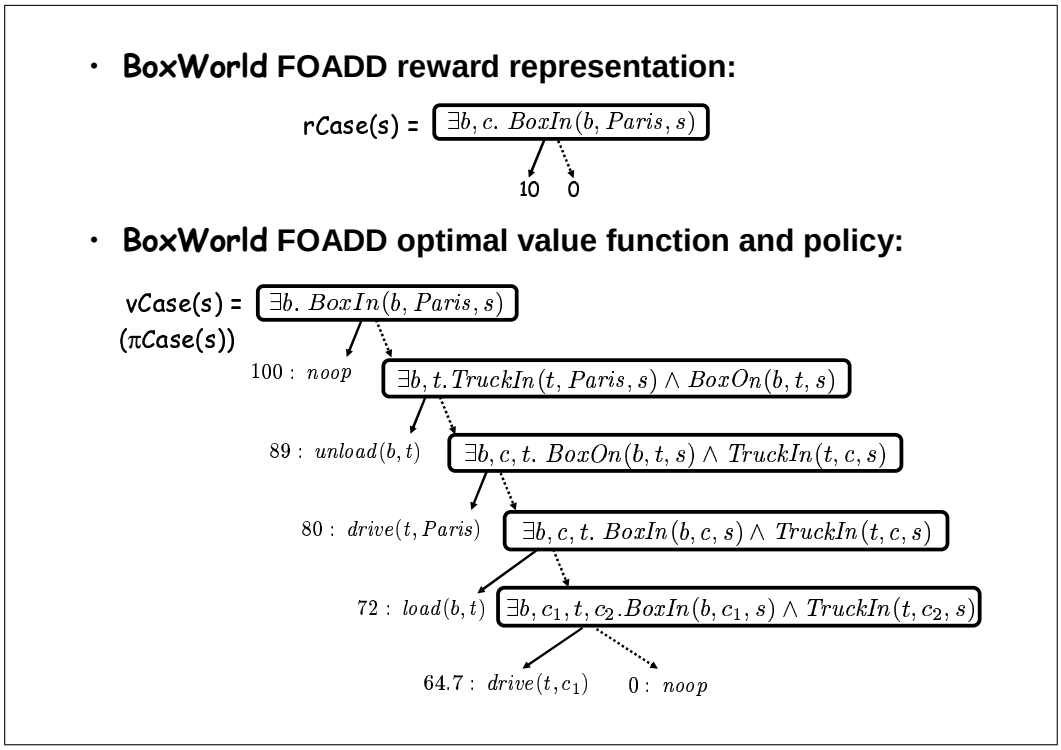

Fig. 6. An example FOADD representation of the reward in BOXWORLD and the FOADD representation of the optimal value function and policy for this domain.

of the BOXWORLD problem, including the version given in Boutilier et al. (2001) with an extra fluent for $Rain(s)$ and action probabilities conditioned on this fluent. We also used a BOXWORLD reward with the following structure:

$$R(s) = \begin{array}{|l|} \hline \exists b. BoxIn(b, paris, s) \wedge TypeA(b) \quad : 10 \\ \hline \neg \text{“} \wedge \exists b. BoxIn(b, paris, s) \wedge \neg TypeA(b) : \ 5 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \tag{46}$$

Here in addition to the $Rain(s)$ fluent, we have also added a non-fluent predicate $TypeA(b)$ to distinguish types of boxes and varying rewards for each type of box. The FOADDs for these solutions are too large to display, but we note that after a small number of steps of value iteration, the value function FOADD stopped growing indicating that all relevant state partitions had been identified. Value iteration continued with this quiesced FOADD until all values at the leaves converged. The respective solution times to convergence within tolerance 1e-4 for these more complex problems were 70.4s and 489s on a 2Ghz Pentium with 2Gb of RAM. For comparison, the ReBel algorithm (Kersting et al., 2004) produced the same solution for the first FOMDP variant with the $Rain(s)$ fluent in $<$6s on a 3.1Ghz machine. ReBel's specialization for a less expressive subset of FOMDPs (still capturing BOXWORLD, however) results in a substantial performance edge. We discuss differences between ReBel and the work in this article in Section 6.

There appear to be at least two general criteria for problem domains to demonstrate finitely-sized optimal value functions with the current case representation as occurred in these examples: (1) the non-zero reward case partitions must be existentially quantified and (2) the FOMDP dynamics must not introduce transitive structure that cannot be finitely bounded by domain axioms. As this last requirement is vague, we provide an example. In the BOXWORLD problem covered in this section, we implicitly assume that all cities are accessible from each other via the $drive$ action. If instead we had some underlying road topology indicated by $Conn(City\ :\ c_1, City\ :\ c_2)$ that restricted the $drive$ action *and* we did not know this topology in terms of prior knowledge specified as domain axioms, then the SDP algorithm would likely need to generate representations for all possible topologies, thus likely leading to a value function of infinite "size." Infinite-sized value functions can also occur when condition (1) is violated as we discuss in the next subsection. We discuss potential research directions to mitigate these observed deficiencies of the case representation in Section 7.1.

Unfortunately, the FOADD solution approach has failed to scale to more complex problems used in the planning community (particularly problems from the ICAPS 2004 and 2006 International Planning Competitions) since they typically use more complex rewards, including those with universal quantifiers. Whereas problems with existentially quantified rewards may exhibit a finite-size optimal value function, this is rarely the case with universal rewards. Thus additional techniques are required to handle this problem, as we discuss next.

4.2 Decomposing Universal Rewards

In first-order domains, we are often faced with *universal reward expressions* that assign some positive value to the world states satisfying a formula of the general form $\forall y\ \phi(y, s)$, and 0 otherwise. For instance, in our BOXWORLD problem, we may define a reward as having *all* boxes b at their assigned destination city c given by $Dst(b, c)$:

$$R(s) = \begin{array}{|l|} \hline \forall b, c.\ Dst(b, c) \supset BoxIn(b, c, s) : 1 \\ \hline \neg \text{“} \qquad\qquad\qquad\qquad\qquad\qquad : 0 \\ \hline \end{array} \tag{47}$$

One difficulty with such rewards is that our case statements provide a piecewise-constant representation of the value function. However, with universal rewards, the value function typically depends on the *number* of domain objects of interest. In our example, value at a state depends on the number of boxes not at their proper destination (since this can impact the minimum number of steps it will take to obtain the reward). So a t-stage-to-go value function in this case would have the following characteristic structure (where we use English in place of first-order logic

for readability):

$$V^t(s) = \quad = \begin{array}{|l|} \hline \forall b, c.\, Dst(b,c) \supset BoxIn(b,c,s) : \quad 1 \\ \hline \text{One box not at destination} \quad : \quad \gamma \\ \hline \text{Two boxes not at destination} \quad : \quad \gamma^2 \\ \hline \vdots \quad : \quad \vdots \\ \hline t-1 \text{ boxes not at destination} \quad : \gamma^{t-1} \\ \hline \end{array}$$

Obviously, since there are t distinct values in an optimal t-stage-to-go value function, the piecewise-constant case representation requires a minimum of t case partitions to represent this value function. And when we combine these counting dynamics with other interacting processes in the FOMDP, we often see an uncontrollable combinatorial blowup in the number of case partitions of value functions for FOMDPs with universally defined rewards. As noted by Gretton and Thiebaux (2004), effectively handling universally quantified rewards is one of the most pressing issues in the practical solution of FOMDPs.

To address this problem we adopt a decompositional approach, motivated in part by techniques for additive rewards in MDPs (Boutilier et al., 1997; Singh and Cohn, 1998; Meuleau et al., 1998; Poupart et al., 2002). We divide our solution into offline and on-line components where the on-line component requires a finite-domain assumption in order to execute the policy.

4.2.1 Offline Generic Goal Solution

Intuitively, given a goal-oriented reward that assigns positive reward if $\forall \vec{y}\ G(\vec{y}, s)$ is satisfied, and zero otherwise, we can decompose it into a set of ground goals $\{G(\vec{y_1}), \ldots, G(\vec{y_n})\}$ for all possible $\vec{y_j}$ in a ground domain of interest. If we reach a state where all ground goals are true, then we have satisfied $\forall y\ G(y, s)$.

Of course, our methods solve FOMDPs without knowledge of the specific domain, so the set of ground goals that will be faced at run-time is unknown. Thus, in the offline FOMDP solution, we assume a *generic* ground goal $G(\vec{y}^*)$ for a "generic" object vector $\vec{y}^*$. Assuming that our universal reward takes an implicative form as it does in our reworked BOXWORLD example, the conditions in the antecedent ($Dst(b,c)$) indicate the goal objects of interest (all pairs $\langle b, c\rangle$ satisfying $Dst(b,c)$) and the consequent of the implication indicates the specific goal $G(\vec{y}, s)$ to be achieved for these objects ($BoxIn(b,c,s)$).

It is easy to construct a generic instance of a reward function $R_{G(\vec{y}^*)}(s)$ given a single goal. In our BOXWORLD example we would introduce the distinguished

constants b^* and c^* to denote our goal objects of interest $G(b^*, c^*)$:

$$rCase_{G(b^*,c^*)}(s) = \boxed{\begin{array}{l} BoxIn(b^*, c^*, s) \quad : 1 \\ \hline \neg BoxIn(b^*, c^*, s) : 0 \end{array}} \tag{48}$$

Given this simple reward, it is easy to derive a value function $V_{G(\vec{y}^*)}(s)$ for this FOMDP using SDP or the approximate FOMDP solution algorithms that we introduce in subsequent sections. $V_{G(\vec{y}^*)}(s)$ and its corresponding policy assume that $\vec{y}^*$ is the only object vector of interest satisfying relevant type constraints and goal preconditions in the domain. In our running BOXWORLD example, the optimal $vCase_{G(b^*,c^*)}(s)$ would look very similar to Figure 4 (or 6) with some differences owing to the fact that our reward is defined in terms of constants b^* and c^* rather than existentially quantified variables b and c.

We next derive Q-functions for each action $A_i(\vec{x})$ from the value function $V_{G(\vec{y}^*)}(s)$ for the "generic" domain:

$$Q_{G(\vec{y}^*)}(A_i, s) = \exists \vec{x}.\, FODTR[V_{G(\vec{y}^*)}(s), A_i(\vec{x})] \tag{49}$$

For our running BOXWORLD example, we would derive $qCase_{G(b^*,c^*)}(A_i, s)$ for $A_i \in \{unload, load, drive\}$.

4.2.2 Online Policy Evaluation

With the offline solution (i.e., Q-function for each action) of a generic goal FOMDP in hand, we address the online problem of action selection for a specific domain instantiation given at run-time. We assume a set of ground goals $\{G(\vec{y_1}), \ldots, G(\vec{y_n})\}$ corresponding to a specific finite domain given at run-time. If we assume that (typed) domain objects are treated uniformly in the uninstantiated FOMDP, as is the case in many logistics and planning problems, then we obtain the Q-function for any goal $G(\vec{y_j})$ by replacing all ground terms $\vec{y}^*$ in $qCase_{G(\vec{y}^*)}(A_i, s)$ with the respective terms $\vec{y_j}$ to obtain $qCase_{G(\vec{y_j})}(A_i, s)$.

Returning to our running example, from the value function $vCase_{G(b^*,c^*)}(s)$ we derived a Q-function $qCase_{G(\vec{y}^*)}(A_i, s)$ for each action A_i. If at run-time, we are given the three goals $Dst(b_1, paris)$, $Dst(b_2, berlin)$, and $Dst(b_3, rome)$, then we would substitute these goals into our Q-functions to obtain three goal-specific Q-functions for each action A_i:

$$\{qCase_{G(b_1,paris)}(A_i, s),\ qCase_{G(b_2,berlin)}(A_i, s),\ qCase_{G(b_3,rome)}(A_i, s)\} \tag{50}$$

Action selection requires finding an action that maximizes value with respect to the original universal reward. Following (Boutilier et al., 1997; Meuleau et al., 1998), we do this by treating the *sum of the Q-values* of any action in the subgoal MDPs as

Algorithm 1: $EvalPolicy(\{qCase_{G(\vec{y}^*)}(\cdot,\cdot)\}, \{G(\vec{y_1}),\ldots,G(\vec{y_n})\}, s) \rightarrow A_i(\vec{c})$

input : (1) For each action template $A_i(\vec{x})$ a set of (non-disjoint) Q-functions $qCase_{G(\vec{y}^*)}(A_i, s)$ for a specific ground instantiation $\vec{y}^*$ of a goal G.
(2) A set of n unsatisfied goals $\{G(\vec{y_1}),\ldots,G(\vec{y_n})\}$ to achieve.
(3) A ground state s to find the best action for.

output : The optimal ground action $A_i(\vec{c})$ to execute with respect to the given state and additive decomposition of unsatisfied goals: $A_i(\vec{c}) = \arg\max_{i,\vec{c}} \sum_{j=1}^{n} qCase_{G(\vec{y_j})}(A_i(\vec{c}), s)$

begin

// In hash table h, entries map ground actions to corresponding value: $A(\vec{x}) \rightarrow v$.
Initialize empty hash table h;

// Now, compute additive values for all matching ground actions
foreach *(action A_i)* **do**
  foreach *(goal $G(\vec{y_j})$)* **do**
    Replace all occurrences of $\vec{y}^*$ in $qCase_{G(\vec{y}^*)}(A_i, s)$ with $\vec{y_j}$;
    foreach *(case partition $\langle \exists \vec{x}\, \phi(\vec{x}), t \rangle \in qCase_{G(\vec{y_j})}(A_i, s)$)* **do**
      foreach *(ground binding $\vec{x} = \vec{c}$ satisfying $\exists \vec{x}\, \phi(\vec{x})$)* **do**
        if *($A_i(\vec{c}) \rightarrow v$ is already in h for some v)* **then**
          Update h to contain $A_i(\vec{c}) \rightarrow \left(v + \frac{t}{n}\right)$;
        else
          Update h to contain $A_i(\vec{c}) \rightarrow \frac{t}{n}$;

// Assume h tracks its maximal entry: $A_i(\vec{c}) \rightarrow v$.
Return the maximal $A_i(\vec{c})$ from h;

end

a measure of its Q-value in the joint (original) MDP. Specifically, we assume that each goal contributes uniformly and additively to the reward, so the Q-function for an entire set of ground goals $\{G(\vec{y_1}),\ldots,G(\vec{y_n})\}$ determined by our domain instantiation is just $\sum_{j=1}^{n} \frac{1}{n} qCase_{G(\vec{y_j})}(A_i, s)$. Action selection (at run-time) in any ground state is realized by choosing the action with maximum additive Q-value. Naturally, we do not want to explicitly create the joint Q-function, but instead use an efficient "scoring" technique that evaluates potentially useful actions by iterating through the individual Q-functions as described in Algorithm 1.

While this additive and uniform decomposition may not be appropriate for all domains with goal-oriented universal rewards (and certainly offers no performance guarantees on account of its heuristic nature), we have found it to provide reasonable results for domains such as BOXWORLD as we empirically demonstrate in the next section. While our approach only currently handles rewards with universal quantifiers, this reflects the form of many planning problems. Nonetheless, this technique could be extended for more complex universal rewards, the general open

question being how to assign credit among the constituents of such a reward.

5 Linear-value Approximation for FOMDPs

Perhaps the greatest difficulty with the symbolic dynamic programming (SDP) approach and practical extensions discussed in the last section is that the size of the value function case representation grows polynomially on each iteration and thus exponentially in terms of the number of iterations. [5] Similar growth can occur for the first-order formulae representing the state partitions themselves. Once these formulae become too large to practically detect equivalence or inconsistency, all hope of obtaining a compact representation of the value function is lost as the number of partitions in the case representation grow unboundedly with no practical means for simplification or pruning. Indeed, the SDP approaches above, using both FOADDs and universal reward decomposition, are incapable of producing value functions and policies competitive with other planners from the ICAPS 2004 and 2006 International Probabilistic Planning Competitions (Littman and Younes, 2004; Gerevini et al., 2006).

Given that approximate solution techniques such as linear value approximation (Guestrin et al., 2002; Schuurmans and Patrascu, 2001; de Farias and Roy, 2003) have allowed MDP solutions to scale far beyond the limits of exact algorithms, at the same time offering reasonable error guarantees, this suggests generalizing linear value approximation techniques to FOMDPs. In this section, we generalize the LP methods for ground MDPs, discussed in Section 2, to the first-order case. This reduces the task of solving an FOMDP to that of obtaining good weights for a set of basis functions that approximates the optimal value function. This requires the generalization of linear programs to handle first-order constraints and further requires efficient extensions of solution methods such as constraint generation and variable elimination in cost networks to exploit the first-order structure of these constraints.

To develop a completely automated linear-value approximation approach to FOMDPs we must address the issue of automatic basis function construction; to do this, we adapt techniques proposed by Gretton and Thiebaux (2004). With appropriate domain axioms defining legal states, our techniques provide fully first-order, non-grounded solutions to FOMDPs derived from PPDDL and can compete with planners from the ICAPS 2004 and ICAPS 2006 International Probabilistic Planning Competitions.

[5] In the worst case, a single case operation can yield a quadratic blowup in the number of case partitions in terms of the maximum number of case partitions in its operands.

5.1 Benefits of Linear-value Approximation

Linear-value approximation for FOMDPs is attractive for several reasons:

- Given that much of the computation in linear value approximation reduces to solving LPs, this reduces the algorithm design space to the setup and solution of linear programs.
- Since the size of linear-value approximations is fixed, it can be used to moderate the complexity of the resulting solution algorithm. This leads to a flexible solution approach that trades off approximation accuracy and computation.
- Linear value approximation does not require extensive logical simplification in practice, just weight projections that make use of a theorem prover. This is a tremendous advantage over exact techniques that require substantial simplification in order to maintain a compact representation.
- Linear value approximation have yielded reasonable empirical performance for ground and factored MDPs, suggesting promise for its application to FOMDPs.
- If we do not use additive reward decomposition techniques of Section 4.2 (which approximate the FOMDP model), then we can derive domain-independent error bounds on our resulting value function using Corollary 3.4.2.

5.2 First-order Linear-value Representation

We represent a value function as a weighted sum of k *first-order basis functions*, denoted $b_i(s)$, each ideally containing a *small* number of formulae that provide a first-order abstraction of state space:

$$V(s) = \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \tag{51}$$

Throughout this section, we assume that each individual basis function $b_i(s)$ is represented by a case statement that is an exhaustive and disjoint partitioning of state space. This property will be useful when we define the backup operators next. However, two basis functions may assign non-zero values to overlapping regions of state space; in fact this can be quite useful for representing additively decomposable values.

Such a linear value function representation can often provide a reasonable approximation of the exact value function, especially given the additive structure inherent in many real-world problems. For example, as argued in previous sections, many planning problems have additive reward functions or multiple goals, both of which lend themselves to approximation via linearly additive basis functions. Unlike exact solution methods where value functions can grow exponentially in size during the solution process and must be logically simplified, here we maintain the value

function in a compact form that requires no simplification, just discovery of good weights.

As an example, consider approximation of the value function for our BOXWORLD FOMDP from the last section, using the following basis functions (we refer to specific instances of $b_i(s)$ as $bCase_i(s)$):

$$bCase_1(s) = \begin{array}{|l|} \hline \exists b.\, BoxIn(b, paris, s) : 1 \\ \hline \neg \text{“} \qquad\qquad : 0 \\ \hline \end{array}$$

$$bCase_2(s) = \begin{array}{|l|} \hline \exists b, t.\, BoxOn(b, t, s) : 1 \\ \hline \neg \text{“} \qquad\qquad : 0 \\ \hline \end{array} \tag{52}$$

$$bCase_3(s) = \begin{array}{|l|} \hline \exists b, t.\, TruckIn(t, paris, s) \wedge BoxOn(b, t, s) : 1 \\ \hline \neg \text{“} \qquad\qquad : 0 \\ \hline \end{array}$$

Then each instance of $V(s)$ (denoted by $vCase(s)$) has the form:

$$vCase(s) = [w_1 \cdot bCase_1(s)] \oplus [w_2 \cdot bCase_2(s)] \oplus [w_3 \cdot bCase_3(s)] \tag{53}$$

Each basis function is relatively small and represents a portion of state space to which we would expect to assign some positive value in order to approximate the BOXWORLD value function.

5.2.1 Backup Operators

Suppose we are given a value function $V(s)$. Backing up this value function through an action $A(\vec{x})$ yields a case statement containing the logical description of states that would give rise to $V(s)$ after doing action $A(\vec{x})$, as well as the values thus obtained.

However, due to the free variables in action $A(\vec{x})$, there are in fact two types of backups that we can perform. The first, $B^{A(\vec{x})}[\cdot]$, regresses a value function through an action and produces a case statement with *free variables* for the action parameters. The second, $B^{A}[\cdot]$, existentially quantifies over the free variables $\vec{x}$ in $B^{A(\vec{x})}[\cdot]$. Thus, the application of $B^{A}[\cdot]$ results in a case description of the regressed value function indicating the values that could be achieved by *any* instantiation of $A(\vec{x})$ in the pre-action state.

The definition of $B^{A(\vec{x})}[\cdot]$ is almost the same as the *first-order decision theoretic regression* (FODTR) operator from Equation 30, except that we do not explicitly

add in the reward. Slightly modifying our definitions from Section 3.3.3, we let $n_1(\vec{x}), \ldots, n_q(\vec{x})$ be the set of Nature's deterministic actions for stochastic action $A(\vec{x})$. Then we define $B^{A(\vec{x})}[\cdot]$ as follows:

$$B^{A(\vec{x})}[V(s)] \\ = \gamma \left[\bigoplus_{j=1}^{q} \{P(n_j(\vec{x}), A(\vec{x}), s) \otimes Regr(V(\mathit{do}(n_j(\vec{x}), s)))\}\right] \tag{54}$$

Defining $B^{A(\vec{x})}[\cdot]$ in this way without the reward makes it a linear operator. Thus, if we apply this operator to our linear-value function representation, it distributes to each first-order basis function:

$$B^{A(\vec{x})}[V(s)] = B^{A(\vec{x})}\left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s)\right] \\ = \bigoplus_{i=1}^{k} w_i \cdot B^{A(\vec{x})}\left[b_i(s)\right] \tag{55}$$

Having defined $B^{A(\vec{x})}[\cdot]$, we now use it to define $B^{A}[\cdot]$: [6]

$$B^{A}[V(s)] = \exists \vec{x}.\ \left\{B^{A(\vec{x})}[V(s)]\right\} \tag{56}$$

Unfortunately, if we apply $B^{A}[\cdot]$ to our linear-value function representation, we see that $B^{A}[\cdot]$ is not necessarily linear:

$$B^{A}[V(s)] = B^{A}\left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s)\right] \\ = \exists \vec{x}. \left\{\bigoplus_{i=1}^{k} w_i \cdot B^{A(\vec{x})}\left[b_i(s)\right]\right\} \tag{57}$$

The difficulty is that the existential quantification of $B^{A}[\cdot]$ jointly constrains the backup of all basis functions that contain the existentially quantified variable as a free variable.

These problems can be mitigated, however. We begin with a few definitions.

Definition 5.2.1 *We say that a deterministic action $n_j(\vec{x})$* affects *a fluent F if there is a positive or negative effect axiom that contains $a = n_j(\vec{x})$ in the body of the axiom and F in the head (c.f., Section 3.2.2). We say that a stochastic action $A(\vec{x})$ affects a fluent F if at least one of Nature's choices $n_j(\vec{x})$ for $A(\vec{x})$ affects F. Finally, a formula ϕ is* affected by *a stochastic action $A(\vec{x})$ iff ϕ contains a fluent affected by $A(\vec{x})$. Since a case statement is defined as a logical formula, this definition extends to case statements in the obvious way.*

[6] For simplicity, we assume that the reward is independent of the action arguments $\vec{x}$, allowing us to exclude the reward from the $\exists \vec{x}$ operation of B^{A}. If required, such dependencies could be added with appropriate adjustments to our definitions.

Property 5.2.2 *When a basis function case statement $b_i(s)$ is affected by a stochastic action $A(\vec{x})$, $B^{A(\vec{x})}[b_i(s)]$ will contain the action arguments $\vec{x}$ as free variables. The inverse of this property is also true: if a stochastic action $A(\vec{x})$ does not affect a basis function $b_i(s)$, $B^{A(\vec{x})}[b_i(s)]$ will not contain the action arguments as free variables.*

To exploit this property, we let I_A^+ denote the set of indices i for basis functions $b_i(s)$ that are affected by an action $A(\vec{x})$ (so that for all $i \in I_A^+$, $B^{A(\vec{x})}[b_i(s)]$ contains at least one of the free variables $\vec{x}$). Likewise, we let I_A^- denote the set of indices of basis functions $b_i(s)$ not affected by an action (so that for all $i \in I_A^-$, $B^{A(\vec{x})}[b_i(s)]$ contains none of the free variables $\vec{x}$). We can exploit the fact that the $\exists \vec{x}$ is vacuous for case statements not containing free variables $\vec{x}$ and remove these terms from the scope of the $\exists \vec{x}$ quantification. This yields the following form for B^A:

$$B^A\left[\bigoplus_i w_i b_i(s)\right] = \left(\bigoplus_{i \in I_A^-} w_i B^{A(\vec{x})}\left[b_i(s)\right]\right) \oplus \exists \vec{x}.\ \left(\bigoplus_{i \in I_A^+} w_i B^{A(\vec{x})}\left[b_i(s)\right]\right) \tag{58}$$

Consequently, if no fluent occurs in more than a few basis functions and no action affects more than a few fluents then we can reasonably expect the result of applying B^A to retain some additive structure. The first property can be controlled by the appropriate design of basis functions. The second is true of typical planning domains.

As a concrete example to demonstrate the backup operators and the exploitation of additive structure, let us compute $B^{drive}[\cdot]$ for our previously specified linear-value function from Equation 53:

$$\begin{aligned}
&B^{drive}[vCase(s)] = \exists t^*, c^*\ B^{drive(t^*,c^*)}[vCase(s)] \qquad (59)\\
&\quad = \exists t^*, c^*\ B^{drive(t^*,c^*)}[w_1 \cdot bCase_1(s) \oplus w_2 \cdot bCase_2(s) \oplus w_3 \cdot bCase_3(s)]\\
&\quad = \exists t^*, c^*\ \Big\{w_1 \cdot B^{drive(t^*,c^*)}[bCase_1(s)] \oplus w_2 \cdot B^{drive(t^*,c^*)}[bCase_2(s)]\\
&\qquad\qquad \oplus w_3 \cdot B^{drive(t^*,c^*)}[bCase_3(s)]\Big\}\\
&\quad = \exists t^*, c^* \Bigg\{ w_1 \cdot \begin{array}{|l|r|}\hline \exists b.\ BoxIn(b, paris, s) & : 0.9 \\ \hline \neg\text{“} & : \ 0 \\ \hline \end{array} \oplus w_2 \cdot \begin{array}{|l|r|}\hline \exists b, t.\ BoxOn(b, t, s) & : 0.9 \\ \hline \neg\text{“} & : \ 0 \\ \hline \end{array}\\
&\qquad\qquad \oplus w_3 \cdot \begin{array}{|l|r|}\hline \exists b, t.\ [t = t^* \wedge c^* = paris \wedge \exists c_1\, TruckIn(t, c_1, s)] & \\ \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) & : 0.9 \\ \hline \neg\text{“} & : \ 0 \\ \hline \end{array} \Bigg\}
\end{aligned}$$

Here, we note that the first and second basis functions are not affected by the $drive(t^*, c^*)$ action and thus their backup through this action is equivalent to a backup through a $noop$. Since the third basis function is affected by the action $drive(t^*, c^*)$ and this introduces the action parameters t^* and c^* into the result of its

backup, we can push the quantifiers in to just this third case statement:

$$B^{drive}[vCase(s)] = \exists t^*, c^*.\ B^{drive(t^*,c^*)}[vCase(s)]$$

$$= w_1 \cdot \boxed{\begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}} \oplus w_2 \cdot \boxed{\begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}}$$

$$\oplus w_3 \cdot \exists t^*, c^* \left\{ \boxed{\begin{array}{|l|}\hline \exists b, t.\ [t = t^* \wedge c^* = paris \wedge \exists c_1\, TruckIn(t, c_1, s)] \\ \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) \quad : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}} \right\}$$

Finally, we carry out the explicit $\exists t^*, c^*$ operation on the third case statement where we distribute the quantifiers inside the case partitions and simplify. This allows us to remove the $\exists t^*, c^*$ by rewriting equalities and exploiting the non-empty domain assumption:

$$B^{drive}[vCase(s)] = \exists t^*, c^*.\ B^{drive(t^*,c^*)}[vCase(s)] \tag{60}$$

$$= w_1 \cdot \boxed{\begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}} \oplus w_2 \cdot \boxed{\begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}}$$

$$\oplus w_3 \cdot \boxed{\begin{array}{|l|}\hline \exists b, t.\ [(\exists c_1.\ TruckIn(t, c_1, s)) \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}}$$

This example demonstrates best case performance for $B^A[\cdot]$, where an action only affects one basis function thus allowing the other basis functions to be removed from the scope of the $\exists \vec{x}$ operator. Then the $\exists \vec{x}$ operator can be easily applied to a single case statement without incurring a representational blowup that would otherwise occur if the $\exists \vec{x}$ ranged over a sum of case statements and the explicit "cross-sum" $\oplus$ was required.

Of course, in many cases, more than one basis function will be affected by an action. For example, if we had computed $B^{unload}[vCase(s)]$, all three basis functions would have been affected by the action and we would have had to explicitly compute the "cross-sum" $\oplus$ of the backups of all three basis functions. While this effectively counteracts many of the benefits of linear-value approximation since additive structure can no longer be exploited, we will see that by generating our basis functions in a restricted manner, we can often manage to avoid computing the explicit $\oplus$, even when *all* basis functions are affected by an action. We will discuss this further when we discuss basis function generation.

5.3 First-order Approximate Linear Programming

We now generalize the approximate linear programming (ALP) approach for MDPs (see Equation 11) to first-order MDPs. If we simply substitute appropriate notation,

we arrive at the following formulation of first-order ALP (FOALP):

$$\begin{aligned}
&\text{Variables: } w_i \ ; \ \ \forall i \leq k \\
&\text{Minimize: } \sum_s \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \\
&\text{Subject to: } 0 \geq R(s) \oplus B^A \left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s) \right] \ominus \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \ ; \ \forall \ A, s
\end{aligned} \tag{61}$$

As with ALP, our variables are the weights of our basis functions and our objective is to minimize the sum of values over all states s. We have one constraint for each stochastic action A (e.g., in BOXWORLD, $A \in \{unload, load, drive\}$) and each state s. One advantage of FOALP over SDP is that it does not require a casemax, thus avoiding the representational blowup incurred by this step in SDP.[7] Unfortunately, while the objective and constraints in ALP for a ground MDP range over a finite number of states, this direct generalization to the FOALP approach for FOMDPs requires dealing with infinitely (or indefinitely) many states s.

Since we are summing over infinitely many states in the FOALP objective, it is ill-defined. Thus, we redefine the FOALP objective in a manner that preserves the intention of the original approximate linear programming solution for MDPs. In ALP (see Equation 11), the objective equally weights each state and minimizes the sum of the value function over all states. However, if we look at the case partitions $\langle \phi_i(s), t_i \rangle$ of each basis function $b_i(s)$ case statement, each case partition serves as an aggregate representation of ground states assigned equal value. Consequently, rather than count ground states in our FOALP objective—of which there will generally be an infinite number per partition—we suppose that each basis function partition is chosen because it represented a potentially useful partitioning of state space, and thus weight each case partition equally. Consequently, we rewrite the FOALP objective as follows:

$$\sum_s \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \ \ = \ \ \bigoplus_{i=1}^{k} w_i \sum_s b_i(s) \ \ \sim \ \ \bigoplus_{i=1}^{k} w_i \sum_{\langle \phi_j, t_j \rangle \in b_i} \frac{t_j}{|b_i|}$$

We use $|b_i|$ to indicate the number of partitions in the ith basis function. This approach can be seen as aggregating states within a basis function partition into one abstract state and then weighting each abstract state uniformly in importance. For the case of 0-1 indicator basis functions as in Equation 52, this yields a simple objective of $\sum_{i=1}^{k} w_i$. Of course, this solution requires approximating the original objective and thus FOALP does not represent an exact generalization of the ground ALP approach to the first-order case. Nonetheless, we show that this approximation still leads to reasonable results in our empirical evaluation.

[7] The reasons for this are the same as for the lack of a $\max$ in the ground case as discussed in Section 2.2.3.

With the issue of the infinite objective resolved, this leaves us with one final problem—the infinite number of constraints (i.e., one for every state s). Fortunately, we can work around this since case statements are finite. Since the value t_i for each case partition $\langle \phi_i(s), t_i \rangle$ is constant over all situations satisfying the $\phi_i(s)$, we can explicitly sum over the $case_i(s)$ statements in each constraint to yield a single case statement representation of the constraints. The key observation here is that the finite number of constraints represented in the single "flattened" case statement hold iff the original infinite set of constraints in Equation 61 hold.

To understand this, consider the constraints for the $drive$ action in FOALP, substituting our previously defined basis functions $bCase_i(s)$ from Equation 52 for $b_i(s)$, the results of the B^{drive} operator for these basis functions from Equation 60, and the reward definition for BOXWORLD given by $rCase(s)$ in Equation 20 for $R(s)$. We substitute all of these directly into the constraint of the form in Equation 61 above to obtain:

$$
\begin{aligned}
0 \geq & \begin{array}{|l|}\hline \exists b. BoxIn(b, paris, s) : 10 \\ \hline \neg\text{“} \quad : \ 0 \\ \hline \end{array} \oplus w_1 \cdot \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 0.9 \\ \hline \neg\text{“} \quad : \ 0 \\ \hline \end{array} \\
& \oplus w_2 \cdot \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 0.9 \\ \hline \neg\text{“} \quad : \ 0 \\ \hline \end{array} \\
& \oplus w_3 \cdot \begin{array}{|l|}\hline \exists b, t.\ [(\exists c_1.\ TruckIn(t, c_1, s)) \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) : 0.9 \\ \hline \neg\text{“} \quad : \ 0 \\ \hline \end{array} \\
& \ominus w_1 \cdot \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 1 \\ \hline \neg\text{“} \quad : 0 \\ \hline \end{array} \ominus w_2 \cdot \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 1 \\ \hline \neg\text{“} \quad : 0 \\ \hline \end{array} \\
& \ominus w_3 \cdot \begin{array}{|l|}\hline \exists b, t.\ TruckIn(t, paris, s) \wedge BoxOn(b, t, s) : 1 \\ \hline \neg\text{“} \quad : 0 \\ \hline \end{array} ; \ \forall\ s
\end{aligned}
\tag{62}
$$

Next we perform an explicit $\oplus$ and $\ominus$ for some of the case statements, simplify the resulting partitions, and distribute the weights into the partition values:

$$
\begin{aligned}
0 \geq & \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 10 - 0.1 \cdot w_1 \\ \hline \neg\text{“} \quad : \quad 0 \\ \hline \end{array} \oplus \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : -0.1 \cdot w_2 \\ \hline \neg\text{“} \quad : \quad 0 \\ \hline \end{array} \\
& \oplus \begin{array}{|l|}\hline \exists b, t.\ TruckIn(t, paris, s) \wedge BoxOn(b, t, s) \quad : -0.1 \cdot w_3 \\ \hline \neg\text{“} \wedge \exists b, t, c_1.\ TruckIn(t, c_1, s) \wedge BoxOn(b, t, s) : \quad 0.9 \cdot w_3 \\ \hline \neg\text{“} \quad : \quad 0 \\ \hline \end{array} ; \ \forall\ s
\end{aligned}
\tag{63}
$$

To maintain our representation in a compact and perspicuous form, we define the following propositional renamings for the first-order formulae in these case state-

ments: [8]

$$\begin{aligned}
\phi_1(s) &\equiv \exists b.\, BoxIn(b, paris, s) \\
\phi_2(s) &\equiv \exists b, t.\, BoxOn(b, t, s) \\
\phi_3(s) &\equiv \exists b, t.\, TruckIn(t, paris, s) \wedge BoxOn(b, t, s) \\
\phi_4(s) &\equiv \exists b, t, c_1.\, TruckIn(t, c_1, s) \wedge BoxOn(b, t, s)
\end{aligned}$$

Finally, we fully expand the $\oplus$ to obtain an explicit representation of *all* FOALP constraints for the $drive$ action in our BOXWORLD example:

$\phi_1(s) \wedge \phi_2(s) \wedge \phi_3(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 + -0.1 \cdot w_3$
$\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 + 0.9 \cdot w_3$
$\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2$
$\phi_1(s) \wedge \neg\phi_2(s) \wedge \phi_3(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_3$
$\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + 0.9 \cdot w_3$
$\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s)$	:	$0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2$
$\neg\phi_1(s) \wedge \phi_2(s) \wedge \phi_3(s)$	:	$0 \geq -0.1 \cdot w_2 + -0.1 \cdot w_3$
$\neg\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s)$	:	$0 \geq -0.1 \cdot w_2 + 0.9 \cdot w_3$
$\neg\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s)$	:	$0 \geq -0.1 \cdot w_2$
$\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \phi_3(s)$	:	$0 \geq -0.1 \cdot w_3$
$\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s)$	:	$0 \geq 0.9 \cdot w_3$
$\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s)$	:	$0 \geq 0$

; $\forall\, s$

(64)

Here, if we had detected that any partition formula had been inconsistent, we would have removed it and the corresponding constraint.

While we note that technically there are an infinite number of constraints (one for every possible state s), there are only a finite number of *distinct* constraints. In fact, the case representation conveniently partitions the state space into regions with the same constraint. Thus, to solve the FOALP problem, we could enumerate all consistent constraints for every action and then directly solve the resulting LP. In addition to the above constraints for the $drive$ action in BOXWORLD, this approach would require us to carry out a similar procedure for the $unload$, $load$, and $noop$ actions; however, once we did this, we would have all of the constraints necessary for solving the FOALP first-order linear program specification.

[8] One will note that the renaming of first-order formulae with "propositional" variables is in the same spirit as FOADDs. Consequently, we note that FOADDs prove to be an efficient method for representing and performing operations on the constraints that occur in FOALP.

However, as the number of basis functions increases, the number of constraints can grow exponentially in the number of case statements in the constraint. To tackle this problem, we examine the underlying optimization problem in the next section.

5.4 First-order Linear Programs

We can restate the FOALP problem as the optimal solution to a general ***first-order linear program*** (FOLP) for which we provide a generic solution. A FOLP is nothing more than a standard linear program where the constraints are written in terms of a sum of case statements whose case partition values may be specified as linear combinations of the weights. Efficiently solving FOLPs poses a number of difficulties—and we tackle these difficulties next.

5.4.1 General Formulation

A FOLP is specified as follows:

$$\begin{aligned} &\text{Variables: } w_1, \ldots, w_k \ ; \\ &\text{Minimize: } \sum_{i=1}^{k} c_i w_i \\ &\text{Subject to: } 0 \geq case_{1,1}(\vec{w}, s) \oplus \ldots \oplus case_{1,l(1)}(\vec{w}, s) \ ; \ \forall \ s \\ &\qquad\qquad \vdots \\ &\qquad\qquad 0 \geq case_{m,1}(\vec{w}, s) \oplus \ldots \oplus case_{m,l(n)}(\vec{w}, s) \ ; \ \forall \ s \end{aligned} \tag{65}$$

The k variables $\vec{w} = \langle w_1, \ldots, w_k \rangle$ and objective weights $\vec{c} = \langle c_1, \ldots, c_k \rangle$ are defined as in a typical LP, the main difference being the form of the constraints. Here we have m different constraints of varying length $l(j)$ (i.e., the number of case statements in constraint j, $1 \leq j \leq n$). We allow the t_i in each partition $\langle \phi_i, t_i \rangle$ of $case(\vec{w}, s)$ to be linearly dependent on the weights $\vec{w}$ (e.g., $t_i = 3w_1 + 2w_2$). We note that the first-order LP for FOALP can be cast in this general form. As previously discussed in our FOALP example, we could simply compute the explicit "cross-sum" $\oplus$ to flatten out each constraint j into a single case statement as in Equation 64. However, this could be inefficient as it scales exponentially in the number of summed case statements. Fortunately, we can extend constraint generation methods used in factored MDPs (Schuurmans and Patrascu, 2001) to the first-order case as we show next.

5.4.2 First-order Cost Network Maximization

In the constraint generation approach to solving a FOLP, the most important operation is to find a most-violated constraint given a current solution (i.e., setting

Algorithm 2: $FOMax(C, \langle R_1 \ldots R_n \rangle) \longrightarrow \langle S, v \rangle$

input : (1) A set $C = \{case_1, \ldots, case_n\}$.
(2) An ordering $\langle R_1 \ldots R_n \rangle$ of *all* relations in C.

output : (1) The maximum value v achievable.
(2) A set $S = \{\langle \phi_i, t_i \rangle \in case_i\}$ for $i = 1 \ldots n$ s.t. $v = t_1 + \ldots + t_n$.

begin

// Convert C into CNF

for *($i = 1 \ldots n$)* **do**

foreach $(\langle \phi_i, t_i \rangle \in case_i(s))$ **do**

Convert ϕ_i to a set of CNF formulae.

foreach *(relation $R \in \langle R_1 \ldots R_n \rangle$ (in order))* **do**

// Divide C into two sets of cases based on whether they contain R

$C_R^+ := \{case_i | case_i \in C \wedge \exists j.(\langle \phi_j, t_j \rangle \in case_i) \wedge \phi_j$ contains relation $R\}$

$C_R^- := C \setminus C_R^+$

// Build explicit "cross-sum" $\oplus$ of its cases & convert to CNF

$case_R^+ := \bigoplus_{case_i \in C_R^+} case_i$

foreach *($\langle \phi_j, t_j \rangle \in case_R^+$)* **do**

Convert ϕ_j to CNF.

foreach *($\langle \phi_j, t_j \rangle \in case_R^+$ in order from highest to lowest value)* **do**

Resolve all clauses in ϕ_j on relation R until quiescence or inference limit.

// All resolvents on R derived so further resolution on these clauses

// cannot lead to the empty clause — thus clauses can be removed

Remove all clauses in ϕ_j containing R

// Remove inconsistent partitions (i.e., those containing empty clause)

if *($\emptyset \in \phi_j$)* **then**

Remove $\langle \phi_j, t_j \rangle$ from $case_R^+$ and continue with next $\langle \phi_j, t_j \rangle$.

// Remove θ-subsumed partitions that are dominated

foreach *($\langle \phi_i, t_i \rangle \in case_R^+$ where $t_i > t_j$)* **do**

if *($\phi_j \preceq_\theta \phi_i$)* **then**

Remove $\langle \phi_j, t_j \rangle$ from $case_R^+$ and continue with next $\langle \phi_j, t_j \rangle$.

$C := \{case_R^+\} \cup C_R^-$

$v := 0$; $S := \emptyset$

foreach *(maximal value partition $\langle \phi_j, t_j \rangle$ of each $case \in C$)* **do**

$v := v + t_j$; $S := S \cup$ all partitions from input C contributing to $\langle \phi_j, t_j \rangle$

Return $\langle v, S \rangle$.

end

of weights $\vec{w}$). In this section, we formulate this problem as maximization over a first-order generalization of a cost network (Dechter, 1999) represented as follows:

$$0 \geq \max_s [case_1(\vec{w}, s) \oplus \ldots \oplus case_n(\vec{w}, s)] \tag{66}$$

The use of $\max_s$ indicates that we are only interested in the single value (and corresponding case partitions contributing to this value) that maximizes the RHS. casemax would be less efficient here since it would exhaustively enumerate *all* values and constraints when we only require the single maximal value and constraint.

To determine the $\max_s$ with this form of the constraints, we define the $FOMax$ algorithm (see Algorithm 2) to carry out this computation. It is similar to *variable elimination* (Zhang and Poole, 1994) or *bucket elimination* (Dechter, 1999) (which makes a stronger connection to resolution), except that we use a simple ordered version of first-order resolution in place of propositional ordered resolution. Thus, we term this generalized variable elimination technique used by $FOMax$ to be *relation elimination*.

Ostensibly, relation elimination and the technique of *first-order variable elimination* (FOVE) (Poole, 2003; de Salvo Braz et al., 2005; de Salvo Braz et al., 2006) appear similar since they both deal with lifted versions of variable elimination. However, they fundamentally apply to different problems: FOVE does not permit quantified formulae in its representation, while relational elimination permits full first-order logic in its representation; furthermore, FOVE permits the representation of indefinite products and sums whereas relation elimination only permits finite products and sums. Here we require full first-order logic, but not indefinite products or sums. While it is beyond the scope of this article to delve into a detailed discussion, we note that both relation elimination and FOVE can be combined when required; this occurs, for example, in FOALP approaches to factored FOMDP solutions (c.f., Sanner and Boutilier (2007) and Chapter 6 of Sanner (2008)).

We provide a concrete example of $FOMax$ and relation elimination in Figure 7. Relation elimination proceeds analogously to variable elimination, except that we choose a relation R to eliminate at every step rather than a propositional variable. Elimination order can affect the time and space requirements of $FOMax$ since eliminating R requires the "cross-sum" $\oplus$ of all case statements containing R, incurring a polynomial blowup in the number of case statements being summed. In practice, we greedily eliminate the relation R at each step that minimizes this representational blowup, although this is not guaranteed to provide an optimal order.

On any elimination step of $FOMax$, once all of the case statements containing R have been explicitly "cross-summed," the next step is to determine whether any case partitions are inconsistent (via resolution) or θ-subsumed and dominated in value (using the generalized θ subsumption operator $\preceq_\theta$ (Buntine, 1988) with respect to our background theory, similar to the approach used by ReBeL (Kersting et al., 2004)); in both cases, these partitions may be removed since they will never contribute to the maximally consistent partition. Once all relations have been eliminated, maximal case partitions and their values extracted from the remaining sum of case statements are used to generate the maximal value (and case partitions contributing to this value).

We note that the ordered resolution strategy we use in $FOMax$ is not refutation-complete: it may loop indefinitely at an intermediate relation elimination step before finding a latter relation with which to resolve a contradiction. This is an unavoidable consequence of the fact that refutation resolution for general first-order theories is semi-decidable. From a practical standpoint, it is necessary to bound the number of resolutions performed at each relation elimination step (100 clauses per elimination step in our experiments) to prevent non-termination of $FOMax$ due to an infinite number of resolutions. This incomplete theorem proving approach may generate unnecessary constraints corresponding to unsatisfiable regions of state space; while these constraints serve to overconstrain the set of feasible solutions, this has not led to infeasibility problems in practice. Furthermore, we often omit the generalized θ-subsumption test $\preceq_\theta$ since the savings from this simplification does not outweigh its computational cost. This does not affect completeness since simplification is not required for inconsistency detection.

Finally, we remark that if the resolution procedure does finitely terminate before the inference limit is reached on every step of $FOMax$, then the conjunction of case partition formulae returned by $FOMax$ is guaranteed to be satisfiable as a consequence of the completeness of refutation resolution. Research on decidable resolution procedures for expressive subsets of first-order logic (Motik, 2006) may pave the way for stronger completeness guarantees for $FOMax$ in future work.

5.4.3 First-Order Constraint Generation

We can use the $FOMax$ algorithm to find the maximal constraint violation when we have constraints of the form in Equation 66. This allows us to define the following first-order constraint generation algorithm where we have specified some solution tolerance ϵ:

(1) Initialize LP with $i = 0$, $\vec{w}^i = \vec{0}$, and empty constraint set.
(2) For each constraint in the cost-network form of Equation 66, find the maximally violated constraint C (if one exists) using the $FOMax$ algorithm applied to the constraint instantiated with $\vec{w}^i$.
(3) If C's constraint violation is larger than ϵ, add C to the LP constraint set, otherwise return $\vec{w}^i$ as solution.
(4) Solve LP with new constraints to obtain $\vec{w}^{i+1}$, goto step 2

In first-order constraint generation, we initialize our LP with an initial setting of weights, but no constraints. Note that the initial weights $\vec{w}^0 = \vec{0}$ will violate at least one constraint in a FOMDP with non-zero reward. Then we alternate between generating constraints based on maximal constraint violations at the current solution and re-solving the LP with these additional constraints. This process repeats until no constraints are violated and we have found the optimal solution. In practice, this approach typically generates *far* fewer constraints than the full exhaustive enumer-

Suppose we are given the following hypothetical constraint specification for a first-order linear program:

$$0 \geq \max_s \left(\begin{array}{|l:r|} \hline \forall b, c.\ Dst(b,c) \supset BoxIn(b,c,s) & 10 \\ \hline \neg \text{“} & 0 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \exists b, c.\ Dst(b,c) \wedge \neg BoxIn(b,c,s) & w_1 \\ \hline \neg \text{“} & -w_1 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \exists t, c.\, TruckIn(t,c,s) & w_2 \\ \hline \neg \text{“} & 0 \\ \hline \end{array} \right)$$

Assume our last LP solution gave $w_1 = 2$ and $w_2 = 1$. We can compute the most violated constraint (if one exists) by evaluating the weights in the constraint and applying $FOMax$. We begin by converting all first-order formulae to CNF where $c_1, \ldots, c_6$ are Skolemized constants. Formulae are negated prior to Skolemization and once in CNF are only resolved with each other (i.e., never negated).

$$0 \geq \max_s \left(\begin{array}{|l:r|} \hline \{\neg Dst(b,c) \vee BoxIn(b,c,s)\} & 10 \\ \hline \{Dst(c_1,c_2), \neg BoxIn(c_1,c_2,s)\} & 0 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \{Dst(c_3,c_4), \neg BoxIn(c_3,c_4,s)\} & 2 \\ \hline \{\neg Dst(b,c) \vee BoxIn(b,c,s)\} & -2 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \{TruckIn(c_5,c_6,s)\} & 1 \\ \hline \{\neg TruckIn(t,c,s)\} & 0 \\ \hline \end{array} \right)$$

Assume elimination order $BoxIn, Dst, TruckIn$. First we eliminate $BoxIn$: we take the cross-sum $\oplus$ of case statements containing $BoxIn$, repeatedly resolve clauses in each partition on $BoxIn$ until quiescence, and remove all clauses containing $BoxIn$ (indicated by struck-out text):

$$0 \geq \max_s \left(\begin{array}{|l:r|} \hline \{\cancel{\neg Dst(b,c) \vee BoxIn(b,c,s)}, Dst(c_3,c_4), \cancel{\neg BoxIn(c_3,c_4,s)}, \neg Dst(c_3,c_4), \emptyset\ \} & 12 \\ \hline \{\ \cancel{\neg Dst(b,c) \vee BoxIn(b,c,s)}\} & 8 \\ \hline \{Dst(c_1,c_2), Dst(c_3,c_4), \cancel{\neg BoxIn(c_1,c_2,s)}, \cancel{\neg BoxIn(c_3,c_4,s)}\ \} & 2 \\ \hline \{\cancel{\neg Dst(b,c) \vee BoxIn(b,c,s)}, Dst(c_1,c_2), \cancel{\neg BoxIn(c_1,c_2,s)}, \emptyset\ \} & -2 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \{TruckIn(c_5,c_6,s)\} & 1 \\ \hline \{\neg TruckIn(t,c,s)\} & 0 \\ \hline \end{array} \right)$$

Because the partitions valued 12 and -2 contain the empty clause $\emptyset$ (i.e., they are inconsistent), we can remove them. And because the partition of value 8 dominates the partition of value 2 (i.e., $2 < 8$ and the empty clause set of the value 8 partition trivially θ-subsumes the clauses of the value 2 partition), we can remove it as well. This yields the following simplified result:

$$0 \geq \max_s \left(\begin{array}{|l:r|} \hline \{\ \} & 8 \\ \hline \end{array} \oplus \begin{array}{|l:r|} \hline \{TruckIn(c_5,c_6,s)\} & 1 \\ \hline \{\neg TruckIn(t,c,s)\} & 0 \\ \hline \end{array} \right)$$

From here it is obvious that the Dst elimination step will have no effect and the $TruckIn$ elimination step will yield a maximal consistent partition with value 9. Since this is a positive value and thus a violation of the original constraint, we can generate the new linear constraint $0 \geq 10 + -w_1 + w_2$ based on the original constituent partitions that led to this maximal constraint violation.

Fig. 7. An example use of FOMAX to find the maximally violated constraint during first-order constraint generation.

ation approach given by Equation 64. To provide intuitions for this, we refer back to the example of finding the most violated constraint in Figure 7.

Using first-order constraint generation, we now have a solution to the first-order LP from Equation 65, thus providing a general solution for FOALP. At this point, the only step for FOALP that we have not automated is the generation of basis functions, which we discuss next.

5.5 Automatic Generation of Basis Functions

The effective use of linear approximations requires a "good" set of basis functions, one that spans a space containing a good approximation to the true value function. Previous work has addressed the issue of basis function generation in ground MDPs (Patrascu et al., 2002; Mahadevan, 2005), while other work has addressed the inductive generation of first-order features or basis functions from sampled experience (Yoon et al., 2005; Wu and Givan, 2007). Here we consider a deductive first-order basis function generation method that draws on the work of Gretton and Thiebaux (2004). Specifically, they use regressions of the reward as candidate basis functions for learning a value function. This technique has allowed them to generate fully or t-stage-to-go optimal policies for a range of BLOCKSWORLD problems.

We leverage a similar approach for generating candidate basis functions using regression, except that rather than use these candidate basis functions to learn a value function, we fit their weights without sampling or grounding by using FOALP. Algorithm 3 provides an overview of our basis function generation algorithm. The motivation for this approach is as follows: if some portion of state space ϕ has value $v > \tau$ in an existing approximate value function for some nontrivial threshold τ, then this suggests that states that can reach this region (i.e., found by Regr(ϕ) through some deterministic action) should also have reasonable value. However, since we have already assigned value to ϕ, we want the new basis function to focus on the area of state space not covered by ϕ; thus we negate ϕ and conjoin it with $Regr(\phi)$.

As a small example, given the initial ***weighted*** basis function $bCase_1(s) = w_1 \cdot rCase(s)$ from BOXWORLD,

$$bCase_1(s) = w_1 \cdot \boxed{\begin{array}{|r|}\hline \exists b.\, BoxIn(b, paris, s) : 10 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}}, \tag{67}$$

we derive the following weighted basis function from $bCase_1(s)$ when considering

Algorithm 3: $BasisGen(\text{FOMDP}, \tau, \text{n}) \longrightarrow \text{B}$

input : (1) A FOMDP specification.
(2) a value threshold τ
(3) an iteration limit n

output : A set B of basis functions $bCase_i(s)$ and corresponding weights w_i.

begin

// Note: $rCase(s)$ may be a sum of cases, so we can start with many basis functions.

$B := \{rCase(s)\}$

for *($i = 1 \ldots n$)* **do**

foreach *($bCase_i(s) \in B$)* **do**

foreach *($\langle \phi_i(s), t_i \rangle \in bCase_i(s)$)* **do**

foreach *(deterministic action $n_j(\vec{x})$)* **do**

$$B := B \cup \begin{array}{|l|r|} \hline \neg\phi_i \wedge \exists \vec{x}\ Regr(\phi_i(do(n_j(\vec{x}), s))) & : 1 \\ \hline \neg \text{“} & : 0 \\ \hline \end{array}$$

Solve for the weights $\vec{w}$ using FOALP.

foreach *($bCase_i(s) \in B$)* **do**

if *($w_i < \tau$)* **then**

Discard $bCase_i(s)$ from B and ensure it is not regenerated.

if *(no new basis functions generated on this iteration)* **then**

Return B, $\vec{w}$.

Return B, $\vec{w}$.

end

deterministic action $A_i = unloadS(b^*, t^*)$ during basis function generation:

$$bCase_2(s) = \tag{68}$$
$$w_2 \cdot \begin{array}{|l|r|} \hline \neg[\exists b.\ BoxIn(b, paris, s)] \wedge [\exists c.\ BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : 1 \\ \hline \neg \text{“} & : 0 \\ \hline \end{array}$$

If one examines the form of these two basis functions, the inherent “orthogonality” between the new basis functions and the ones from which they were derived allows for significant computational optimizations. For example, since the top partition of $bCase_1(s)$ takes the form ϕ_1 and the top partition of $bCase_2(s)$ takes the form $\neg\phi_1 \wedge \phi_2$, these two partitions are mutually exclusive and could never jointly contribute to the value of a state. Thus, when two basis functions are orthogonal in this manner, we can efficiently perform an explicit “cross-sum” $\oplus$ on them to obtain a single

compact case statement representing both weighted basis functions:

$$bCase_{1,2}(s) = bCase_1(s) \oplus bCase_2(s) \tag{69}$$

$$= \begin{array}{|l|r|} \hline \exists b.\, BoxIn(b, paris, s) & : w_1 \cdot 10 \\ \hline \neg[\exists b.\, BoxIn(b, paris, s)] \wedge [\exists c.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : w_2 \\ \hline \neg \text{“} & : 0 \\ \hline \end{array}$$

This style of basis function generation also has many computational advantages for FOALP. To see this, we return to our original discussion concerning the fact that the $B^A[\cdot]$ operator as defined in Equation 58 will not be able to preserve additive structure when all basis functions in the linear-value function representation are affected by the stochastic action $A(\vec{x})$. Recalling Property 5.2.2, if all basis functions are affected by $A(\vec{x})$, then the backup $B^A[\cdot]$ of a sum of basis functions will require their explicit “cross-sum” since they will all have free variables $\vec{x}$ causing them to be summed with $\exists \vec{x}$ is applied. However, in the best case, if the explicit “cross-sum” was already pre-computed for orthogonal basis functions by merging them, then this blowup will not occur.

Of course, since different actions generate different non-orthogonal basis functions from the same “parent” basis function, it will not generally hold that all basis functions are pairwise orthogonal to each other. Nonetheless, if we can exploit the mutual orthogonality of *subsets* of the basis functions to efficiently carry-out their explicit “cross-sum”, then we can still achieve an exponential time speedup relative to the worst-case of the $B^A[\cdot]$ operator that requires the explicit computation of the “cross-sum”. To see how subsets of basis functions can be efficiently summed, we refer back to Equation 69, which provides an example sum of two orthogonal basis functions. In general, any mutually orthogonal subset of basis functions can be merged in this way.

As a consequence, we can exploit properties of orthogonal basis function generation in FOALP to mitigate exponential space and time scaling in the number of basis functions, where worst-case exponential scaling arises at various points due to the need to explicitly compute the “cross-sum” of the linear-value representation. While we do not claim this method of basis function generation will be appropriate for all domains, we will demonstrate that it works reasonably well for the stochastic planning problems evaluated in the next section.

5.6 Empirical Results

We evaluated FOALP on PPDDL planning problems from the ICAPS 2004 (Littman and Younes, 2004) and ICAPS 2006 (Gerevini et al., 2006) International Probabilistic Planning Competitions (IPPC). We divide the discussion of results according to

each competition in order to reflect the differences in the competition setup, the data collected, and the specific planners that entered each competition.

We used the Vampire theorem prover and the CPLEX 9.0 LP solver [9] in our FOALP implementation and applied $BasisGen$ (Algorithm 3) to our FOMDP translation of these PPDDL domains, generated as described in Section 3.2.2. We additively decomposed universal rewards using the technique described in Section 4.2; we note that doing so prevents us from obtaining any approximation guarantees on the solution generated by FOALP.

We provided FOALP with additional background theory axioms that were not encoded in the PPDDL source: if a fluent was intended to have functional arguments in PPDDL (PPDDL does not make provisions for specifying this property explicitly), we provide a background axiom stating this. So, for example, in our running BOXWORLD example, we would provide the following functional constraint axioms:

$$\forall b, c_1, c_2, s.\ BoxIn(b, c_1, s) \land BoxIn(b, c_2, s) \supset c_1 = c_2$$
$$\forall t, c_1, c_2, s.\ TruckIn(t, c_1, s) \land TruckIn(t, c_2, s) \supset c_1 = c_2$$
$$\forall b, t_1, t_2, s.\ BoxOn(b, t_1, s) \land BoxOn(b, t_2, s) \supset t_1 = t_2$$

In words, these axioms state that a box can only be in one city, a truck can only be in one city, and a box can only be on one truck. Any search-based or inductive planner that is given an initial state respecting these constraints (which was always the case in the competition instances) would never have to consider such erroneous states violating these constraints since they are unreachable from non-erroneous states satisfying these constraints. However, FOALP has no initial state knowledge in its offline solution phase and will produce extremely poorly approximated value functions if it cannot rule out such erroneous states as being inconsistent.

The need for these constraints may be viewed as a major drawback of the FOALP approach and was the reason that, although FOALP entered the ICAPS 2006 Probabilistic Planning Competition, it did not compete on 6 of the 10 problem domains (since these 6 problem domains were released at the start of the competition and rules prevented the planners from being modified beyond this point). On the other hand, we note that functional constraints on fluents represent a minimal type of problem knowledge often easily encoded by the person specifying a PPDDL problem; the constraints for BOXWORLD are a good example. As an aid to future non-grounding planners, we recommend that the capability to specify functional constraints on fluents be incorporated in future versions of the PPDDL specification. If such constraints are known to hold on all initial states, automated techniques based on reachability analysis could also be used to prove such constraints hold as well.

In the following sections, we present proof-of-concept results comparing FOALP

[9] *http://www.ilog.com/products/cplex/*

Problem	Competing Probabilistic Planners					
	NMRDPP	**mGPT**	**Humans**	**Classy**	**FF-Replan**	**FOALP**
bx c10 b5	438	184	419	376	425	433
bx c10 b10	376	0	317	0	346	366
bx c10 b15	0	–	129	0	279	0
bw b5	495	494	494	495	494	494
bw b11	479	466	480	480	481	480
bw b15	468	397	469	468	0	470
bw b18	352	–	462	0	0	464
bw b21	286	–	456	455	459	456

Fig. 8. Cumulative reward of 5 planning systems and FOALP (100 run avg.) on the BOXWORLD and BLOCKSWORLD probabilistic planning problems from the ICAPS 2004 IPPC (– indicates no data). BOXWORLD problems are indicated by a prefix of *bx* and followed by the number of cities *c* and boxes *b* used in the domain. BLOCKSWORLD problems are indicated by a prefix of *bw* and followed by the number of blocks *b* used in the domain.

to other planners across a sampling of problems where FOALP has been able to generate policies for IPPC problems.

5.6.1 ICAPS 2004 Probabilistic Planning Competition Problems

We applied FOALP to the BOXWORLD logistics and BLOCKSWORLD probabilistic planning problems from the ICAPS 2004 IPPC (Littman and Younes, 2004). In the BOXWORLD logistics problem, the domain objects consist of trucks, planes, boxes, and cities. The number of boxes and cities varied in each problem instance, but there were always 5 trucks and 5 planes. Trucks and planes are restricted to particular routes between cities in a problem instance-specific manner. The goal in BOXWORLD was to deliver all boxes to their destination cities and there were costs associated with each action. The transition functions allowed for trucks and planes to stochastically end up in destinations other than that intended by the execution of their respective drive and fly actions. BLOCKSWORLD is just a stochastic version of the standard domain where blocks are moved between the table and other stacks of blocks to form a goal configuration. In this version, a block may be dropped with some probability while picking it up or placing it on a stack.

We stopped our offline basis function generation algorithm after iteration 7 in $BasisGen$ (Algorithm 3) taking less than 2 hours for both problems on a 2Ghz Pentium with 2Gb of RAM; iteration 8 could not complete due to memory constraints. We note that if we were not using the "orthogonal" basis function generation described in Section 5.5, we would not get past iteration 2 of basis function generation

(the system does not terminate within 10 hours at iteration 3); thus, these optimizations have substantially increased the number of basis functions for which FOALP is a viable solution option.

We compared FOALP to the three other top-performing planners on these problems: *NMRDPP* is a temporal logic planner with human-coded control knowledge (Thiebaux et al., 2006); *mGPT* is an RTDP-based planner (Bonet and Geffner, 2004); *(Purdue-)Humans* is a human-coded planner, *Classy* is an inductive first-order policy iteration planner, and *FF-Replan* (Yoon et al., 2004) (2004 version) is a deterministic replanner based on FF (Hoffmann and Nebel, 2001). Results for all of these planners are given in Table 8.

Since FOALP was only able to complete 7 iterations of basis function generation, this effectively limits the lookahead horizon of our basis functions to 7 steps. A lookahead of 8 would be required to properly plan in the final BOXWORLD problem instance and thus FOALP failed on this instance. It is important to note that in comparing FOALP to the other planners, NMRDPP and Humans used hand-coded control knowledge. FF-Replan was a very efficient search-based deterministic planner that had a significant advantage because near-optimal policies in these specific goal-oriented problems can be obtained by assuming that the highest probability action effects occur deterministically and making use of classical search-based planning techniques. The only autonomous *fully stochastic* planners were mGPT and Classy (itself an inductive first-order planning approach), and FOALP performs comparably to both of these planners and outperforms them by a considerable margin on some problem instances.

5.6.2 ICAPS 2006 Probabilistic Planning Competition Problems

We now present results for FOALP on three problem domains from the ICAPS 2006 IPPC (Gerevini et al., 2006): BLOCKSWORLD, TIREWORLD, and ELEVATORS. [10] In BLOCKSWORLD, there are blocks and a table and the goal is to stack and unstack blocks from each other in an effort to achieve a goal configuration of the blocks with respect to the table. TIREWORLD is a relatively simple problem where the goal is to drive from a goal city to a destination city, while being able to pick up a spare tire in some cities. One stochastic outcome of driving between cities is that a tire may go flat and can only be fixed when a spare tire is present. Thus, routes with cities that contain spare tires are preferred to other routes that do not. Finally, ELEVATORS is a problem with a grid-like state space. The horizontal dimension of

[10] In the ICAPS 2006 IPPC, FOALP ran on the three problems reported here as well as EXPLODING-BLOCKSWORLD (not reported here). We do not report the EXPLODING-BLOCKSWORLD results since the competition version of the FOALP planner was restricted to use only the BLOCKSWORLD subset of the EXPLODING-BLOCKSWORLD problem description. In this section, we only show results for problems where FOALP was able to generate a policy for the full problem description.

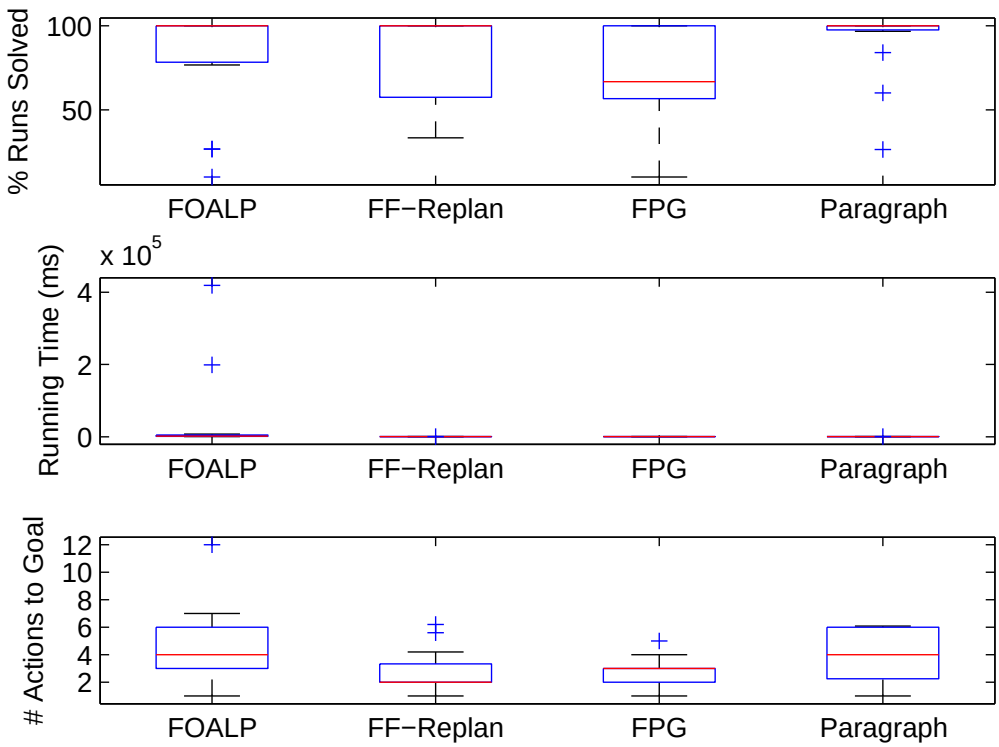

Fig. 9. A boxplot of performance of four planners on 15 instances of the TIREWORLD problem domain from the probabilistic track of the ICAPS 2006 IPPC. sfDP did not produce results for this problem; all other planners reported results for all instances.

the grid corresponds to positions on a floor and the vertical dimension corresponds to different floors. There may be elevators at each position that can move vertically between floors. An agent can occupy one position on one floor and can move left or right between positions or can move into or out of an elevator if it is at the appropriate floor or position. Any elevator can be moved up or down independently of whether the agent resides in it. There can be gates at certain positions, which probabilistically teleport the agent back to the start position of floor 1, position 1. Finally, there are a number of coins at different known positions and the goal is for the agent to retrieve them all.

In all of the following results, $BasisGen$ (Algorithm 3) was run for a four-hour fixed time limit on a 2Ghz Pentium with 2Gb of RAM to generate solutions for successively larger sets of basis functions. At the four-hour mark, we halted the solution process and used the largest (most recent) set of basis functions and weights for which FOALP had successfully terminated. Since the offline solution time of 4 hours can be amortized over an indefinite number of instances for a given problem, we do not report this in the online policy evaluation times in the following results.

In Figures 9, 10, and 11, we provide data for FOALP and competing planners that

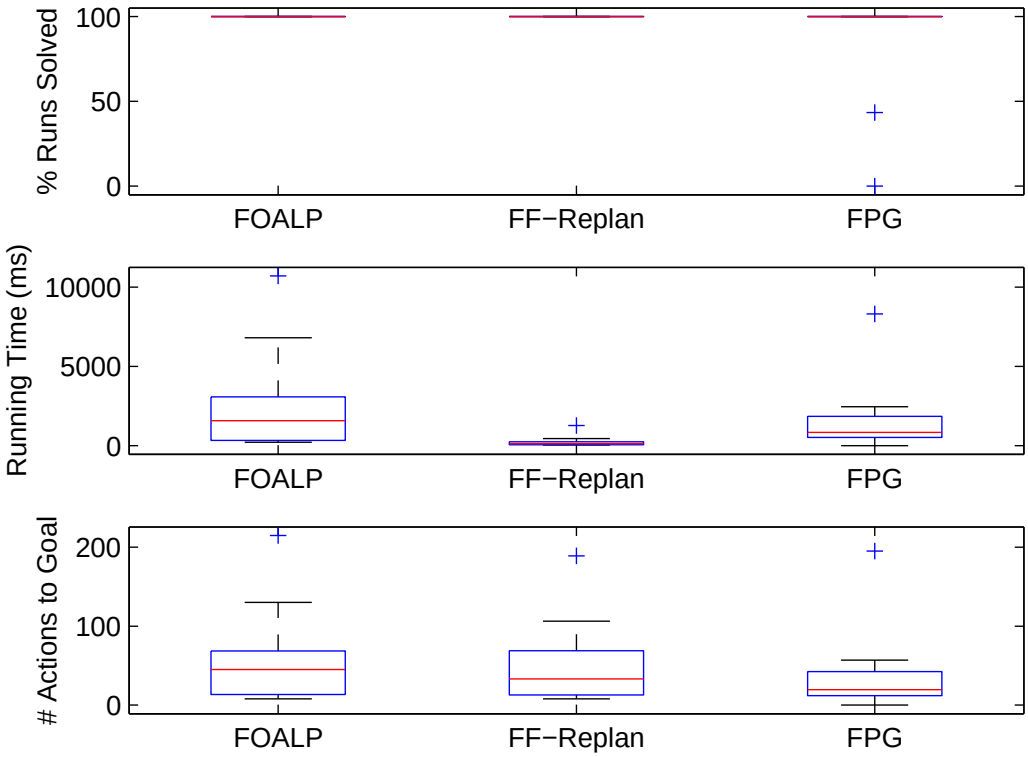

Fig. 10. A boxplot of performance of three planners on 15 instances of the ELEVATORS problem domain from the probabilistic track of the ICAPS 2006 IPPC. sfDP and Paragraph did not produce results for these problems; FF-Replan and FPG did not report results for 2 and 3 problem instances, respectively.

specifies the number of problem instances solved, the online solution generation time, and the average number of actions required to reach the goal in each successful problem. We compare to the following planners that entered the competition [11] : (1) *FPG* (Buffet and Aberdeen, 2006), which uses policy gradient search in a factored representation of the Q-functions; (2) *sfDP* (Teichteil and Fabiani, 2006), which uses ADD-based dynamic programming (Hoey et al., 1999) with reachability constraints based on initial state knowledge; (3) *Paragraph* (Little, 2006), which uses a probabilistic extension of Graphplan (Blum and Furst, 1995) for probabilistic planning; (4) *FF-Replan* (Yoon et al., 2007) (2006 version) is a deterministic replanner based on FF (Hoffmann and Nebel, 2001). We note that all planners in this competition aside from FOALP are ground planners in that they use a propositional representation of a PPDDL problem for a specific domain instantiation.

The results vary by problem, so we explain each in turn. In TIREWORLD, FOALP's policy allowed it to solve most problems although its policy was suboptimal in the number of actions and % problems solved in comparison to FF-Replan. In this case,

[11] Not all planners ran on all of the problems in the competition. Furthermore, some planners did not provide results on all problem instances, this is noted for each result plot.

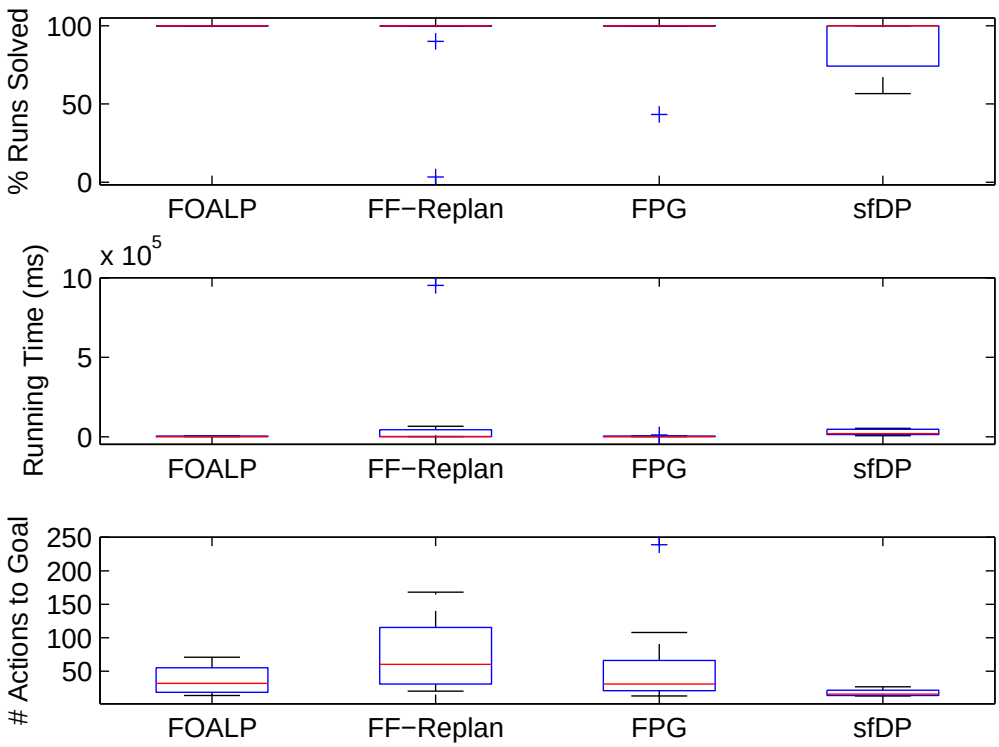

Fig. 11. A boxplot of performance of four planners on 15 instances of the BLOCKSWORLD problem domain from the probabilistic track of the ICAPS 2006 IPPC. Paragraph did not produce results for these problems; FF-Replan, FPG, and sfDP did not report results for 1, 5, and 10 problem instances, respectively.

it appears that the approximation inherent in the FOALP approach fared poorly in comparison to a deterministic replanner like FF-Replan that could perform nearly optimally on this problem. FOALP's slow policy evaluation on this problem is due to the transitive nature of the road connection topology and the lack of optimization in FOALP's logical policy evaluator. In ELEVATORS, the top three planners including FOALP all performed comparably with the deterministic replanner performing consistently faster than the others, again due to the suitability of this domain for deterministic replanning and the relative speed of that approach. The goals in this domain are highly decomposable and FOALP thus benefited substantially from its additive goal decomposition approach. In BLOCKSWORLD, FOALP shows the best performance, solving more problems, taking less time on the hard instances (FPG did not report results for the 5 hardest instances, thus skewing its results), and reaching the goal with the fewer actions (sfDP did not report results for the 10 hardest instances, thus skewing its results). In this case, FOALP's performance owes to two advantages: (1) first-order abstraction in BLOCKSWORLD considerably helps the system avoid much of the combinatorial complexity that the ground planners face, and (2) the additive goal decomposition, although not optimal for all BLOCKSWORLD problems, performed very well on these problem instances.

5.6.3 Summary of Results

In summary, the first-order representation of FOALP seems to offer robust performance across a range of domain instance sizes and problems. However, as discussed at the end of Section 4.1.3, the case representation used by FOALP is a limiting factor in its performance due to its inability to exploit value structure in problems requiring reasoning about universal rewards (for which suboptimal additive reward decomposition techniques were used) or transitive reachability (for which the deficiency is quite clear from the TIREWORLD results). We discuss potential research directions to mitigate these observed deficiencies in Section 7.1.

6 Related Work

In this section, we review work related to that presented in this article across two important dimensions: deductive first-order decision-theoretic planners based on symbolic dynamic programming (SDP), and inductive lifted decision-theoretic planners based on learning first-order representations of value functions, control knowledge, or policies from grounded domain instantiations.

6.1 Variants of Symbolic Dynamic Programming

There have been a variety of alternative exact approaches to solving relationally specified MDPs without grounding in the spirit of SDP. Each of these approaches apply an SDP-like algorithm to their own first-order MDP representation. Like SDP, these algorithms all have guarantees on domain-independent error bounds for the value functions they produce and can produce exact domain-independent value functions when they exist. However, all of these approaches are restricted to solve less expressive variants of relational MDPs than SDP as we describe below.

First-order value iteration (FOVIA) (Karabaev and Skvortsova, 2005; Hölldobler et al., 2006) and the Relational Bellman algorithm (ReBel) (Kersting et al., 2004) are value iteration algorithms that solve a restricted subclass of relational MDPs, most notably disallowing combined *universal conditional* effects (as defined in Section 3.1.1). Since universal conditional effects are a powerful planning formalism underlying the ADL extension to STRIPS, it can be argued that this is a significant limitation of these alternate SDP approaches. Both have provided fully automated proof-of-concept results; we were able to directly compare SDP with FOADDs and ReBel on the BOXWORLD problem in Section 4.1.3. ReBel's specialization for a less expressive subset of FOMDPs (still capturing BOXWORLD, however) results in a substantial performance edge for this problem although both produce the same, exact solution. Results for ReBel and FOVIA are not available

for the specific versions of the planning competition domains that we examined in Section 5.

First-order decision diagrams (FODDs) (Wang et al., 2008) have been introduced to compactly represent case statements and to permit efficient application of symbolic dynamic programming operations to another restricted class of relational MDPs via value iteration (Wang et al., 2007) and policy iteration (Wang and Khardon, 2007).

Since FODDs are very similar in spirit to the FOADDs we defined in Section 4.1, we enumerate some of the major differences between these two formalisms:

(1) FODDs disallow explicit universal quantification. This prevents FODDs from being applied to relational MDPs with universal preconditions or alternating quantifiers in their effects, although importantly, they *can* handle universal conditional effects.
(2) Unlike FOADDs, which are maintained in a canonical form, FODDs are maintained in a sorted format, but are not guaranteed to be in a canonical form. As such, they rely on a range of simplification rules to maintain compact representations. This approach has the advantage that some diagrams without a strict order can be exponentially more compact than diagrams with a strict order (Wang et al., 2007). However, rather than having a well-defined simplification algorithm leading to a canonical form, simplification in FODDs is somewhat open-ended and heuristic.
(3) There is no need to reorder internal decision nodes after $Regr$ in FODDs in order to maintain a canonical form. In this way, $Regr$ is more efficient in FODDs than in FOADDs. This results in value and policy iteration algorithms that can be performed completely in terms of FODDs, unlike the current FOADD representation.
(4) FODDs assume an implicit semantics where the maximal value is assumed for all instantiations of the free variables, thus precluding the need to perform explicit $\exists x$ and casemax. In FOADDs, such operations would need to be performed explicitly. As such, the use of FODDs can lead to very compact representations for decision-theoretic planning, but this semantics may interfere with extensions of FODDs to handle universally quantified formulae.

Consequently, FODDs represent an interesting alternative in the design space of data structures for the compact representation of case statements. Nonetheless, the major limitation with respect to the work we present in this article is their limitations w.r.t. representing some forms of universal quantification. Ideally the best approach would be to combine the advantages of FOADDs with those of FODDs. This is a non-trivial problem, however, and an interesting future research direction.

There are many alternative approaches to first-order decision-theoretic planning that reason inductively about sample domain instances and sample trajectories to produce lifted value functions or policies. This stands as an alternative to reasoning symbolically about actions and rewards directly at a first-order level without grounding as done in this article.

In one class of approaches, sampled experience from grounded domain instantiations is used to directly induce relational representations of value or Q-functions in a reinforcement learning approach. This can be done with pure reinforcement learning using relational decision or regression trees to learn a value or Q-function (Dzeroski et al., 2001), combining this with supervised guidance (Driessens and Dzeroski, 2002), or using Gaussian processes and graph kernels over relational structures to learn a value or Q-function (Gartner et al., 2006).

A second approach uses experience sampled from ground domain instantiations to induce first-order policy representations. In one version, policies can be learned directly from sampled experience trajectories generated using other planners (Yoon et al., 2002). In a different vein, policies can be learned in an approximate policy iteration framework (Yoon et al., 2006) that combines trajectory sampling with policy updates derived from these trajectories. In this approach, sample experience trajectories can be generated using planning heuristics (Fern et al., 2003) and/or random walks on problem sizes that are adaptively scaled as planner performance improves (Fern et al., 2004).

A third inductive approach (that could also be used in conjunction with FOALP) allows first-order features to be learned from experience rather than symbolically deriving them directly from the relational MDP specification as described in Section 5.5. In one approach, heuristic control knowledge represented in a first-order taxonomic syntax can be learned from solution trajectories on a given problem (Yoon et al., 2005). In another recent approach, relational basis functions can be learned from sampled trajectories and then used in an approximate value iteration framework (Wu and Givan, 2007).

Since the approaches in this subsection also produce first-order value functions or policies, it is important to compare and contrast them with the symbolic deductive approach we adopt. In this approach, our *ideal* objectives are threefold:

(1) Obtaining domain-independent exact or bounded approximate solutions where possible while exploiting natural relational and first-order planning structure.
(2) Avoiding potential pitfalls of value functions and policies specific to biases from (small) sampled domain instantiations.
(3) Avoiding an intractable representational blowup by grounding in the solution algorithm.

In *practice*, the approaches advocated in this article are unable to effectively achieve objective (1): the heuristics (such as universal reward decomposition from Section 4.2) required to apply our techniques to planning competition problems prevent the derivation of bounds. Objective (2) may be met in practice, although the approximations required for practical applications introduce their own representational biases. Finally, objective (3) may also be satisfied in practice, although the domain-independent approach introduces its own representational blowup by effectively planning for every possible domain instantiation.

In comparison, inductive first-order approaches outlined above share a goal similar to (1) in exploiting natural relational planning structure in a domain-independent manner, but cannot claim to support (2) since they must sample. Theoretical complexity results by Khardon (1999a,b) indicate that (3) is indeed possible to achieve for inductive approaches in some settings. We further note that in practice, the bias and computational complexity inherent in sampling a small set of possible ground domain instantiations of an MDP is not generally problematic since policies that work on one domain instantiation often generalize to similar or larger domains given an appropriate representation language (Yoon et al., 2006).

So we may then ask: which first-order approach is better, inductive or deductive? [12] Empirically, recent results (Wu and Givan, 2007) show that inductive first-order approaches outperform FOALP. Is this the final answer? Hopefully not; but clearly there is still a great deal of work to be done in order to make first-order deductive approaches fully competitive with recent state-of-the-art first-order inductive approaches. Perhaps even more promising though is the potential to combine advances among both approaches; Gretton and Thiebaux (2004) do this in work that combines inductive logic programming with first-order decision-theoretic regression, showing that optimal policies can be induced from few training samples if using deductive methods to generate candidate policy structure. Such approaches offer the hope of combining the best of both worlds while sharing the goal of exploiting first-order structure in relational decision-theoretic planning problems.

7 Future Directions and Concluding Remarks

In this article, we have motivated the need to exploit relational structure in decision-theoretic planning problems. To this end, we have provided a thorough review of

[12] To clarify, we use the term inductive to refer to any algorithm with an inductive component. However, it should be noted that all of the inductive approaches mentioned above incorporate some form of deduction by sampling from the Bellman equations then using induction to obtain a symbolic representation from these samples. In contrast the SDP and FOALP approaches advocated in this article can be viewed as pure symbolic deduction since they deduce their value representations from a lifted version of the Bellman equation.

the FOMDP representation of Boutilier et al. (2001) and showed how to translate an expressive subset of PPDDL to this particular FOMDP representation. We reviewed the solution of FOMDPs via symbolic dynamic programming and contributed additional practical solution techniques based on the use of first-order ADDs (FOADDs), additive value decomposition of universal rewards, and first-order approximate linear programming (FOALP). Combining all of these ideas, we have provided proof-of-concept results from the probabilistic track of the ICAPS 2004 and 2006 International Planning Competitions.

We outline some interesting directions for future work, and offer some concluding remarks on decision-theoretic planning in the framework of FOMDPs.

7.1 Future Directions

There are a number of open issues raised by our work that merit further exploration. We enumerate a few of them:

(1) An interesting approach for the practical application of FOMDPs to decision-theoretic planning is to combine their approximate offline solution with online methods for enhancing their performance. We need only look at the range of successful planners used in planning competitions for ideas. Perhaps one of the most useful approaches would be to use offline methods for solving FOMDPs to generate a first-order approximated value function. Then we could use such a value function as a heuristic seed for online search methods such as RTDP (Barto et al., 1993; Dearden and Boutilier, 1997). Another approach would be to consider domain-specific control knowledge encoded as temporal logic constraints as in TLPlan (Bacchus and Kabanza, 2000), program constraints as in Golog (Levesque et al., 1997) (both TLPlan and Golog are deterministic planners) or decision-theoretic extensions such as DT-Golog (Boutilier et al., 2000). We discuss the use of program constraints further in a moment.

(2) We did not explore approximate extensions of value iteration for FOMDPs. Given the success of the APRICODD planner (St-Aubin et al., 2000) that performs approximate value iteration using ADDs, this approach is quite appealing for first-order approximate value iteration using FOADDs. When the FOADD representing the value function becomes too large, we can simply prune out nodes in the FOADD in an effort to reduce the size of the value function while minimizing the approximation error.

(3) One promising use of FOMDPs is at the highest level of an abstraction hierarchy for agent-based decision-theoretic planning. Dearden and Boutilier (1997) demonstrate that an MDP model can be approximated to a structure that is efficiently solvable and that error bounds can be obtained on the resulting optimal policy in the abstracted model with respect to the optimal policy in the non-abstracted version. If we lift such results to FOMDPs, then this offers a

very appealing paradigm for their use: we can approximate a general FOMDP model to a level that we know we can solve efficiently while obtaining error bounds on the performance of the optimal policy in this approximated model. Or, further afield, we can use a solution to this approximated model as guidance for other more computationally expensive algorithms like ground heuristic search or as seed values (Dearden and Boutilier, 1997) or shaped rewards (Ng et al., 1999) for value iteration in the non-abstracted MDP model.

In addition to these immediate open problems posed by our techniques, we have only touched on the surface of FOMDPs and the vast array of stochastic decision processes and symbolic solution methods that are possible. There remain a number of promising directions for the exploitation of structure in relationally-specified decision-theoretic planning problems that we briefly describe here:

(1) One of the original goals in the FOMDP and symbolic dynamic programming frameworks (Boutilier et al., 2001) was to allow for very general symbolic representations. While most current FOMDP research has assumed a constant numerical representation of the values in case statement partitions, there are many situations where we might obtain non-constant values in our case statements, e.g., compactly representing value functions in FOMDPs with universal rewards that depend on the count of objects satisfying a property in a given situation, or in the context of modeling continuous state properties, perhaps combined with discrete state properties in a first-order generalization of *hybrid MDPs* (Hauskrecht and Kveton, 2004; Guestrin et al., 2004). However, as the case statement is generalized to handle non-constant numerical representations, case operators like the casemax must be appropriately generalized to efficiently handle such value representations (see Section 6.2.3 of Sanner (2008) for one example of such a casemax generalization). Furthermore, theorem provers must also be capable of reasoning about counting properties or (constrained) continuous variables in such symbolic case statement enhancements in order to detect the inconsistency of state partitions.
(2) In many FOMDPs there is an element of underlying topological graph structure. For example, in logistics planning, this graph structure may involve the accessibility of different cities via roads and flight routes. Currently, this graph structure is not exploited by our solution methods. Yet its regularity, if known *a priori*, could likely be exploitable by solution methods that could "compile" out this graph structure. This approach would be far more advantageous than relying on the first-order case representation to extract relevant graph properties using the cumbersome specification of transitively composed relations (i.e., $\exists c_1, c_2.\, Road(c_1, c_2) \wedge \exists c_3.\, Road(c_2, c_3) \wedge \exists c_4. Road(c_3, c_4) \wedge \ldots$).
(3) We often have a predefined set of constraints on the behavior of an agent and we need to optimize the agent's policy with respect to those constraints. If we can specify the program constraints in the form of a Golog program (Levesque et al., 1997), then we can generalize the hierarchy of abstract machines (HAM) architecture (Parr and Russell, 1998; Andre and Russell, 2001) to the case of

solving FOMDPs with respect to Golog program constraints. Such a solution would permit the (approximately) optimal execution of an incompletely specified program over all possible domain-instantiations. Various approaches in the decision-theoretic DT-Golog framework (Boutilier et al., 2000; Ferrein et al., 2003) have provided an initial investigation into these ideas.

The above suggestions are but a few of the many possible extensions to the work presented in this article and first-order decision-theoretic planning in general.

7.2 Concluding Remarks

For a few years immediately succeeding the publication of the symbolic dynamic programming solution (Boutilier et al., 2001) to relationally specified MDPs, this domain-independent non-grounding approach was disparaged as being unrealistic for practical applications due to the complexity of value functions or due to the need for logical simplification and theorem proving (Yoon et al., 2002; Gardiol and Kaelbling, 2004; Guestrin et al., 2003). While these are all in fact significant obstacles to be overcome in the practical application of first-order MDPs to decision-theoretic planning, this article has aimed to show that these obstacles are not insurmountable. It has provided a substantial step in the direction of demonstrating that with careful attention paid to the first-order representation and algorithms specifically designed to exploit that representation, non-grounded lifted solutions are viable in practice as we demonstrated with our proof-of-concept results from the ICAPS 2004 and 2006 International Planning Competitions. Our hope is that this article lays the foundations for further exploration of these non-grounding approaches and permits the integration of these ideas with other lines of research in decision-theoretic planning.

Acknowledgements

We are grateful to the three anonymous reviewers for their extensive comments and suggestions: these have vastly improved the presentation and discussion of our work. We are also grateful to Kee Siong Ng who provided many suggestions and corrections. This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada. This research was conducted while the first author (now with NICTA) was at the Department of Computer Science, University of Toronto. NICTA is funded by the Australian Government's Backing Australia's Ability and the Centre of Excellence programs.

References

Andre, D., Russell, S., 2001. Programmable reinforcement learning agents. In: Advances in Neural Information Processing Systems (NIPS-01). Vol. 13. pp. 78–85.

Bacchus, F., Halpern, J. Y., Levesque, H. J., 1995. Reasoning about noisy sensors in the situation calculus. In: International Joint Conference on Artificial Intelligence (IJCAI-95). Montreal, pp. 1933–1940.

Bacchus, F., Kabanza, F., 2000. Using temporal logics to express search control knowledge for planning. Artificial Intelligence 116 (1-2), 123–191.

Bahar, R. I., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F., 1993. Algebraic Decision Diagrams and their applications. In: IEEE /ACM International Conference on CAD. pp. 428–432.

Barto, A. G., Bradtke, S. J., Singh, S. P., , 1993. Learning to act using real-time dynamic programming. Tech. Rep. UM-CS-1993-002, U. Mass. Amherst.

Bellman, R. E., 1957. Dynamic Programming. Princeton University Press, Princeton, NJ.

Bertsekas, D. P., 1987. Dynamic Programming. Prentice Hall, Englewood Cliffs, NJ.

Bertsekas, D. P., Tsitsiklis, J. N., 1996. Neuro-Dynamic Programming. Athena Scientific, Belmont, MA.

Blum, A. L., Furst, M. L., 1995. Fast planning through graph analysis. In: IJCAI 95. Montreal, pp. 1636–1642.

Bonet, B., Geffner, H., 2004. mGPT: A probabilistic planner based on heuristic search. In: Online Proceedings for The Probablistic Planning Track of IPC-04: *http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*.

Boutilier, C., Brafman, R. I., Geib, C., 1997. Prioritized goal decomposition of Markov decision processes: Toward a synthesis of classical and decision theoretic planning. In: International Joint Conference on Artificial Intelligence (IJCAI-97). Nagoya, pp. 1156–1162.

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

Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D., 1996. Context-specific independence in Bayesian networks. In: Uncertainty in Artificial Intelligence (UAI-96). Portland, OR, pp. 115–123.

Boutilier, C., Reiter, R., Price, B., 2001. Symbolic dynamic programming for first-order MDPs. In: International Joint Conference on Artificial Intelligence (IJCAI-01). Seattle, pp. 690–697.

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., 2000. Decision-theoretic, high-level agent programming in the situation calculus. In: AAAI-00. Austin, TX, pp. 355–362.

Brachman, R., Levesque, H., 2004. Knowledge Representation and Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Buffet, O., Aberdeen, D., 2006. The factored policy gradient planner (ipc-06 version). In: Proceedings of the Fifth International Planning Competition.

Buntine, W., 1988. Generalized subsumption and its application to induction and redundancy. Artificial Intelligence 36, 375–399.

de Farias, D., Roy, B. V., 2003. The linear programming approach to approximate dynamic programming. Operations Research 51:6, 850–865.

de Salvo Braz, R., Amir, E., Roth, D., 2005. Lifted first-order probabilistic inference. In: 19th International Joint Conference on Artificial Intelligence (IJCAI-2005). Edinburgh, UK, pp. 1319–1325.

de Salvo Braz, R., Amir, E., Roth, D., 2006. MPE and partial inversion in lifted probabilistic variable elimination. In: National Conference on Artificial Intelligence (AAAI-06). Boston, USA.

Dearden, R., Boutilier, C., 1997. Abstraction and approximate decision-theoretic planning. Artificial Intelligence 89 (12), 219–283.

Dechter, R., 1999. Bucket elimination: A unifying framework for reasoning. In: Artificial Intelligence. Vol. 113. pp. 41–85.

Driessens, K., Dzeroski, S., 2002. Integrating experimentation and guidance in relational reinforcement learning. In: International Conference on Machine Learning (ICML). pp. 115–122.

Dzeroski, S., DeRaedt, L., Driessens, K., 2001. Relational reinforcement learning. Machine Learning Journal (MLJ) 43, 7–52.

Fern, A., Yoon, S., Givan, R., December 2003. Approximate policy iteration with a policy language bias. In: Advances in Neural Information Processing Systems 16 (NIPS-03).

Fern, A., Yoon, S., Givan, R., June 2004. Learning domain-specific control knowledge from random walks. In: International Conference on Planning and Scheduling (ICAPS-04). pp. 191–199.

Ferrein, A., Fritz, C., Lakemeyer, G., 2003. Extending DTGolog with options. In: 18th International Joint Conference on Artificial Intelligence (IJCAI-2003). Acupulco, Mexico, pp. 144–151.

Fikes, R. E., Nilsson, N. J., 1971. STRIPS: A new approach to the application of theorem proving to problem solving. AI Journal 2, 189–208.

Gardiol, N. H., Kaelbling, L. P., 2004. Envelope-based planning in relational MDPs. In: Advances in Neural Information Processing Systems 16 (NIPS-03). Vancouver, CA, pp. 1040–1046.

Gartner, T., Driessens, K., Ramon, J., 2006. Graph kernels and gaussian processes for relational reinforcement learning. Machine Learning Journal (MLJ) 64, 91–119.

Gerevini, A., Bonet, B., Givan, B. (Eds.), 2006. Online Proceedings for The Fifth International Planning Competition IPC-05: *http://www.ldc.usb.ve/ bonet/ipc5/docs/ipc-2006-booklet.pdf.gz*. Lake District, UK.

Gretton, C., Thiebaux, S., 2004. Exploiting first-order regression in inductive policy selection. In: Uncertainty in Artificial Intelligence (UAI-04). Banff, Canada, pp. 217–225.

Guestrin, C., Hauskrecht, M., Kveton, B., 2004. Solving factored MDPs with continuous and discrete variables. In: 20th Conference on Uncertainty in Artificial

Intelligence. pp. 235–242.
Guestrin, C., Koller, D., Gearhart, C., Kanodia, N., 2003. Generalizing plans to new environments in relational MDPs. In: 18th International Joint Conference on Artificial Intelligence (IJCAI-2003). Acapulco, Mexico, pp. 1003–1010.
Guestrin, C., Koller, D., Parr, R., Venktaraman, S., 2002. Efficient solution methods for factored MDPs. Journal of Artificial Intelligence Research (JAIR) 19, 399–468.
Hauskrecht, M., Kveton, B., 2004. Linear program approximations for factored continuous-state Markov decision processes. In: Advances in Neural Information Processing Systems 16. pp. 895–902.
Hoey, J., St-Aubin, R., Hu, A., Boutilier, C., 1999. SPUDD: Stochastic planning using decision diagrams. In: Uncertainty in Artificial Intelligence (UAI-99). Stockholm, pp. 279–288.
Hoffmann, J., Nebel, B., 2001. The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intellence Research (JAIR) 14, 253–302.
Hölldobler, S., Karabaev, E., Skvortsova, O., 2006. FluCaP: A heuristic search planner for first-order mdps. Journal of Artificial Intelligence Research (JAIR) 27, 419–439.
Howard, R. A., 1960. Dynamic Programming and Markov Processes. MIT Press.
Karabaev, E., Skvortsova, O., 2005. A heuristic search algorithm for solving first-order MDPs. In: Uncertainty in Artificial Intelligence (UAI-05). Edinburgh, Scotland, pp. 292–299.
Kersting, K., van Otterlo, M., de Raedt, L., 2004. Bellman goes relational. In: International Conference on Machine Learning (ICML-04). ACM Press, pp. 465–472.
Khardon, R., 1999a. Learning action strategies for planning domains. Artificial Intelligence 113 (1-2), 125–148.
Khardon, R., 1999b. Learning to take actions. Machine Learning 35 (1), 57–90.
Koller, D., Parr, R., 1999. Computing factored value functions for policies in structured MDPs. In: International Joint Conference on Artificial Intelligence (IJCAI-99). Stockholm, pp. 1332–1339.
Koller, D., Parr, R., 2000. Policy iteration for factored MDPs. In: Uncertainty in Artificial Intelligence (UAI-00). Stockholm, pp. 326–334.
Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R., 1997. GOLOG: a logic programming language for dynamic domains. Journal of Logic Programming 31 (1-3), 59–83.
Little, I., 2006. Paragraph: A Graphplan-based probabilistic planner. In: Proceedings of the Fifth International Planning Competition.
Littman, M. L., Younes, H. L. S. (Eds.), 2004. Online Proceedings for The Probablistic Planning Track of IPC-04: *http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*. Vancouver, Canada.
Mahadevan, S., 2005. Samuel meets Amarel: Automating value function approximation using global state space analysis. In: National Conference on Artificial Intelligence (AAAI-05). Pittsburgh, pp. 1000–1005.

McCarthy, J., 1963. Situations, actions and causal laws. Tech. rep., Stanford University, reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pages 410-417.

Meuleau, N., Hauskrecht, M., Kim, K.-E., Peshkin, L., Kaelbling, L. P., Dean, T., Boutilier, C., 1998. Solving very large weakly coupled Markov decision processes. In: National Conference on Artificial Intelligence (AAAI-98). Madison, WI, pp. 165–172.

Motik, B., January 2006. Reasoning in Description Logics using Resolution and Deductive Databases. Ph.D. thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany.

Ng, A. Y., Harada, D., Russell, S., 1999. Policy invariance under reward transformations: theory and application to reward shaping. In: Proc. 16th International Conf. on Machine Learning. Morgan Kaufmann, San Francisco, CA, pp. 278–287.

Parr, R., Russell, S., 1998. Reinforcement learning with hierarchies of machines. In: M. Jordan, M. K., Solla, S. (Eds.), Advances in Neural Information Processing Systems 10. MIT Press, Cambridge, pp. 1043–1049.

Patrascu, R., Poupart, P., Schuurmans, D., Boutilier, C., Guestrin, C., 2002. Greedy linear value-approximation for factored Markov decision processes. In: National Conference on Artificial Intelligence (AAAI-02). Edmonton, pp. 285–291.

Pednault, E. P. D., 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In: KR. pp. 324–332.

Poole, D., 1997. The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94 (1-2), 7–56.

Poole, D., 2003. First-order probabilistic inference. In: IJCAI. pp. 985–991.

Poupart, P., Boutilier, C., Patrascu, R., Schuurmans, D., 2002. Piecewise linear value function approximation for factored MDPs. In: National Conference on Artificial Intelligence (AAAI-02). Edmonton, pp. 292–299.

Puterman, M. L., 1994. Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York.

Reiter, R., 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In: Lifschitz, V. (Ed.), Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy). Academic Press, San Diego, pp. 359–380.

Reiter, R., 2001. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press.

Riazanov, A., Voronkov, A., 2002. The design and implementation of vampire. AI Communications 15 (2), 91–110.

Rintanen, J., 2003. Expressive equivalence of formalisms for planning with sensing. In: 13th International Conference on Automated Planning and Scheduling. pp. 185–194.

Sanner, S., March 2008. First-order Decision-theoretic Planning in Structured Relational Environments. Ph.D. thesis, University of Toronto, Toronto, ON, Canada.

Sanner, S., Boutilier, C., 2005. Approximate linear programming for first-order MDPs. In: Uncertainty in Artificial Intelligence (UAI-05). Edinburgh, Scotland,

pp. 509–517.
Sanner, S., Boutilier, C., 2006. Practical linear evaluation techniques for first-order MDPs. In: Uncertainty in Artificial Intelligence (UAI-06). Boston, Mass.
Sanner, S., Boutilier, C., 2007. Approximate solution techniques for factored first-order MDPs. In: 17th International Conference on Automated Planning and Scheduling (ICAPS-07). pp. 288 – 295.
Schuurmans, D., Patrascu, R., 2001. Direct value approximation for factored MDPs. In: Advances in Neural Information Processing 14 (NIPS-01). Vancouver, pp. 1579–1586.
Schweitzer, P., Seidmann, A., 1985. Generalized polynomial approximations in markovian decision processes. Journal of Mathematical Analysis and Applications 110, 568–582.
Shapley, L. S., 1953. Stochastic games. Proceedings of the National Academy of Sciences 39, 327–332.
Singh, S. P., Cohn, D., 1998. How to dynamically merge Markov decision processes. In: Advances in Neural Information Processing Systems (NIPS-98). MIT Press, Cambridge, pp. 1057–1063.
St-Aubin, R., Hoey, J., Boutilier, C., 2000. APRICODD: Approximate policy construction using decision diagrams. In: Advances in Neural Information Processing 13 (NIPS-00). Denver, pp. 1089–1095.
Teichteil, F., Fabiani, P., 2006. Symbolic stochastic focused dynamic programming with decision diagrams. In: Proceedings of the Fifth International Planning Competition.
Thiebaux, S., Gretton, C., Slaney, J., Price, D., Kabanza, F., January 2006. Decision-theoretic planning with non-markovian rewards. Journal of Artificial Intelligence Research 25, 17–74.
Tsitsiklis, J. N., Van Roy, B., 1996. Feature-based methods for large scale dynamic programming. Machine Learning 22, 59–94.
Veloso, M., August 1992. Learning by analogical reasoning in general problem solving. Ph.D. thesis, Carnegie Mellon University.
Wang, C., Joshi, S., Khardon, R., 2007. First order decision diagrams for relational MDPs. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). Hyderabad, India, pp. 1095–1100.
Wang, C., Joshi, S., Khardon, R., 2008. First order decision diagrams for relational MDPs. Journal of Artificial Intelligence Research (JAIR) 31, 431–472.
Wang, C., Khardon, R., 2007. Policy iteration for relational MDPs. In: Uncertainty in Artificial Intelligence (UAI-07). Vancouver, Canada.
Wu, J., Givan, R., 2007. Discovering relational domain features for probabilistic planning. In: 17th International Conference on Automated Planning and Scheduling (ICAPS 2007). pp. 344–351.
Yoon, S., Fern, A., Givan, R., 2002. Inductive policy selection for first-order Markov decision processes. In: Uncertainty in Artificial Intelligence (UAI-02). Edmonton, pp. 569–576.
Yoon, S., Fern, A., Givan, R., 2004. Learning reactive policies for probabilistic planning domains. In: Online Proceedings for The Probablistic Planning Track

of IPC-04: *http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*.

Yoon, S., Fern, A., Givan, R., July 2005. Learning measures of progress for planning domains. In: 20th National Conference on Artificial Intelligence. pp. 1217–1222.

Yoon, S., Fern, A., Givan, R., 2006. Approximate policy iteration with a policy language bias: Learning to solve relational markov decision processes. Journal of Artificial Intelligence Research (JAIR) 25, 85–118.

Yoon, S., Fern, A., Givan, R., 2007. FF-Replan: A baseline for probabilistic planning. In: 17th International Conference on Automated Planning and Scheduling (ICAPS-07). pp. 352–359.

Younes, H. L. S., Littman, M. L., Weissman, D., Asmuth, J., 2005. The first probabilistic track of the international planning competition. Journal of Artificial Intelligence Research (JAIR) 24, 851–887.

Zhang, N. L., Poole, D., 1994. A simple approach to bayesian network computations. In: Proc. of the Tenth Canadian Conference on Artificial Intelligence. pp. 171–178.

Zhang, N. L., Poole, D., 1996. Exploiting causal independence in bayesian network inference. Journal of Artificial Intelligence Research (JAIR) 5, 301–328.