First-order Decision-theoretic Planning in Structured Relational Environments

by

Scott Patrick Sanner

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

First-order Decision-theoretic Planning in Structured Relational Environments

Scott Patrick Sanner
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2008

We consider the general framework of first-order decision-theoretic planning in structured relational environments. Most traditional solution approaches to these planning problems ground the relational specification w.r.t. a specific domain instantiation and apply a solution approach directly to the resulting ground Markov decision process (MDP). Unfortunately, the space and time complexity of these solution algorithms scale linearly with the domain size in the best case and exponentially in the worst case. An alternate approach to grounding a relational planning problem is to lift it to a first-order MDP (FOMDP) specification. This FOMDP can then be solved directly, resulting in a domain-independent solution whose space and time complexity either do not scale with domain size or can scale sublinearly in the domain size. However, such generality does not come without its own set of challenges and the first purpose of this thesis is to explore exact and approximate solution techniques for practically solving FOMDPs. The second purpose of this thesis is to extend the FOMDP specification to succinctly capture factored actions and additive rewards while extending the exact and approximate solution techniques to directly exploit this structure. In addition, we provide a proof of correctness of the first-order symbolic dynamic programming approach w.r.t. its well-studied ground MDP counterpart.

Dedication

To my parents: for their love, support and encouragement of learning throughout my life; especially to my Dad for buying me a Commodore 64 (with a whopping 64 kB of memory!) and a Radio Shack electronics kit and sharing with me his joy of curiousity.

Acknowledgements

I have been quite fortunate to have Craig Boutilier as my PhD advisor; Craig's ability to clearly and cleanly formalize problems and intuitions is unsurpassed in my experience. Working with Craig has fundamentally changed the way that I think about and approach research; I only hope that I can have a fraction of this impact on my own future students.

I owe a debt of gratitude to Sheila McIlraith for her advice, inspiration, and encouragement as a committee member, but also as a friend. I first met Sheila while I was working on my Masters degree and her unselfish goodwill since that first serendipitous meeting has opened many doors in my life that may have been otherwise closed. The world needs more people like Sheila.

As my external committee member, Roni Khardon provided a thorough review of my thesis and I am indebted to him for the time and effort he spent doing this. I also thank my other committee members, Fahiem Bacchus and Hector Levesque, for their insights and suggestions over the course of my PhD.

And finally, I am grateful to all of my family and friends for putting up with me throughout my graduate career. Perhaps no one knows this difficulty more than Sonia — for her unconditional love and support (and her mom's cooking), I owe a debt of love that will require many fur coats to repay. ☺

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Decision-theoretic planning is the task of determining an optimal sequence of actions (or more generally, an action policy) that optimizes some reward criterion given state information and a stochastic action model of the environment. It generalizes classical deterministic planning by allowing for the uncertain specification of action outcomes and a utility-based specification of reward that permits one to view plan quality in a fully decision-theoretic paradigm rather than a more limited goal-oriented or cost-to-goal paradigm.

Planning with decision-theoretic notions is ubiquitous throughout the fields of artificial intelligence, operations research, control theory, and economics:

- Robots must optimize their actions in the face of uncertainty and must tradeoff the dangers of approaching obstacles with the need to accomplish their tasks.
- Factories must maximize production in their daily schedule of activities in consideration of process delays and potential equipment failures.
- Financial analysts must make long-term investment decisions with different levels of risk and uncertainty in order to maximize profit.
- Planning in logistics applications requires the minimization of resource usage and the maximization of goods delivered while taking into account the uncertainties inherent in various courses of action.

And these are just a few applications. Simply by definition, decision-theoretic planning is among the most critical components of agent-oriented AI. And if it is generalized to handle partial observability, multiple agents, and sampled model dynamics (i.e., reinforcement learning), this task subsumes almost any decision or control problem in AI. While we won't

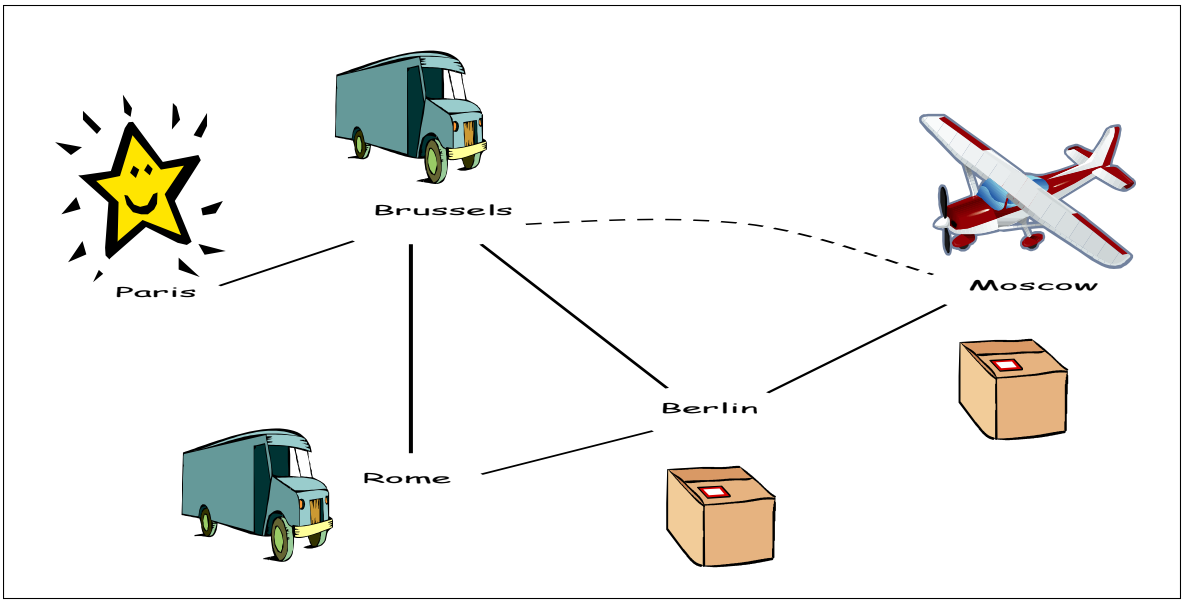


Figure 1.1: An example BOXWORLD problem. Trucks may drive along solid lines and planes may fly along dashed lines. The goal in this instance is to get all boxes in paris (indicated by the star).

consider the most general frameworks for decision-theoretic planning in this thesis, we will consider the general framework of first-order decision-theoretic planning in structured relational environments. This framework subsumes many planning problems for which the state is fully observable, the transition dynamics have a known probability distribution, and the state and action representation is discrete and relational in nature.

1.1 Motivating Examples

To motivate the decision-theoretic planning framework and to provide a sense of what types of problems we aim to solve, we provide two examples of problems that demonstrate rich structural regularities that we will exploit in the solution approaches introduced in the thesis. And while the simple specification of these problems may seem to suggest commensurate simplicity in the solutions, we note that good solutions for these problems are often non-trivial to specify and that the solution methods in this thesis often outperform expert hand-coded policies and other competing algorithms as we will demonstrate empirically.

1.1.1 Logistics Planning

The first problem we introduce is a standard logistics problem commonly referred to as BOXWORLD [Veloso, 1992]. Throughout all variants, the goal is to deliver boxes to their destination city by loading and unloading them from trucks and possibly planes that can move between cities via respective topologies of roads and air routes. In the basic setup, only one action can be executed at each time step, each action being to load or unload a box or to drive a truck or fly a plane to a destination. Each action typically has a pre-specified success probability that can depend on various state properties and can have an associated cost. A single reward is typically given for states where all boxes are delivered to their proper destination. Discounting of future rewards can be used to encourage optimal solutions that achieve this reward state while minimizing the total number of actions or the cost of actions required to achieve it. We provide a pictorial representation of one domain instance of this problem in Figure 1.1.

While the BOXWORLD problem specification may appear straightforward, its exact solution can become very complex as the number of boxes, trucks, planes, and cities grow. Thus, it is an ideal problem for exploring algorithms that can exploit relational and first-order structure to avoid scaling directly with the combinatorial aspects of the domain size.

1.1.2 System Administration

The second problem we introduce is motivated by an abstract hypothetical system adminstrator problem and is commonly referred to as SYSADMIN [Guestrin *et al.*, 2002]. In it, there are n computers $c_1, \ldots, c_n$ connected via a directed graph topology. In each state, a computer can be up and running (or not) and on every time step, a computer may be rebooted, thus causing it to be running in the subsequent state. If a computer is not explicitly rebooted then its probability of running in the next time step is conditioned on its current status and the proportion of computers with incoming connections that are also currently running. The reward is the count of computers that are running at any time step and rewards t time steps into the future are typically discounted exponentially in t. An optimal policy in this problem will reboot the computer that has the most impact on the expected future discounted reward over an infinite time horizon. An example for four computers is given in Figure 1.1.2.

SYSADMIN poses an interesting problem for decision-theoretic planning applications as it exhibits characteristic structure common to many planning problems. Additive rewards or utilities are perhaps one of the most commonly studied reward structures in decision-theory. And the exogenous effects in SYSADMIN that permit each computer to reboot or crash on each time

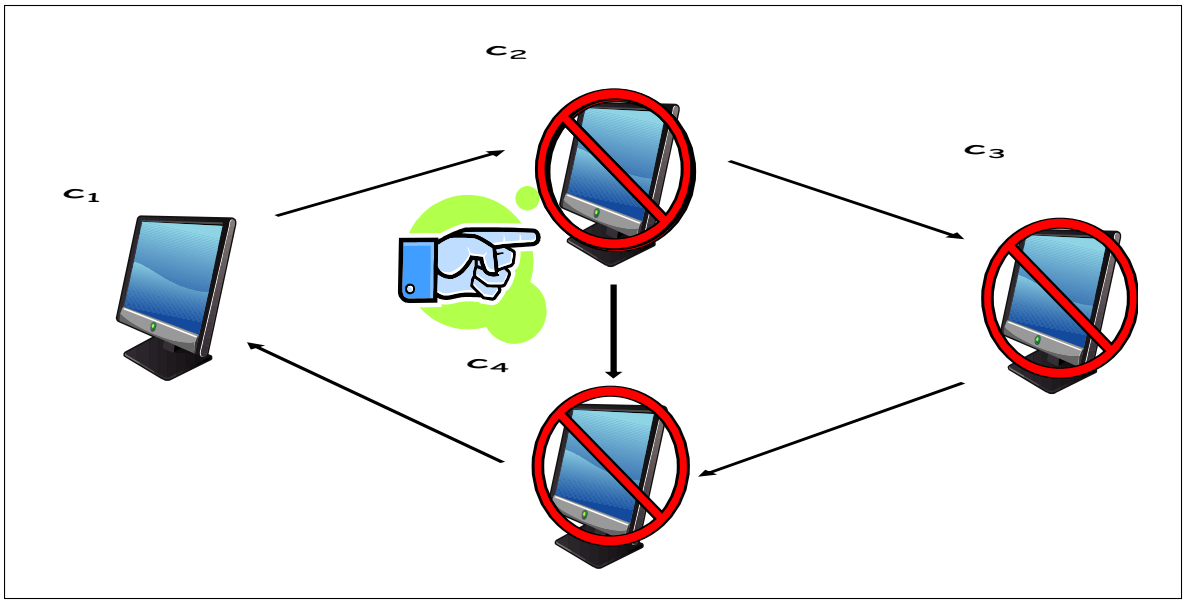


Figure 1.2: An example SYSADMIN problem with the network topology shown as a directed graph. One computer is up and running and three are not (indicated by the red circle with slash). A good action to take in this state would be to reboot c_2.

step are representative of a class of realistic world-models that do not require a strong frame assumption — the notion that the only state properties affected by an action directly result from the action itself. Thus, we focus on SYSADMIN as one of the main examples in this thesis because it is simple enough to warrant the complete exposition of our solution methods, yet representative of a much more general class of structured problems. And despite its deceptive simplicity, we note that the optimal policy may vary widely according to small changes in the network topology, making it very difficult to manually determine optimal policies.

1.2 Exploitation of Structure in Decision-theoretic Planning

Given the various types of structure that we observe in BOXWORLD and SYSADMIN, our aim in this thesis is to exploit this structure for efficiency in decision-theoretic planning. The groundwork for this thesis work follows in a line of extensive research over the years aimed at exploiting structure in order to compactly represent and efficiently solve decision-theoretic planning problems in the Markov decision process (MDP) framework [Boutilier *et al.*, 1999]. While traditional approaches to solving MDPs typically used an enumerated state and action model [Puterman, 1994], this approach has proved impractical for large-scale AI planning tasks where the number of distinct states in a model can easily exceed the limits of primary and

secondary storage on modern computers.

Fortunately, by switching to a factored state model, many MDPs can be compactly described by exploiting various independences in the reward and transition functions. For example, in the SYSADMIN problem, the state can be naturally represented with one binary variable per computer indicating whether the computer is up and running. Then the distribution over the next state of each computer is dependent on only the state variables of computers with direct incoming connections. Furthermore, while the reward in SYSADMIN is dependent upon all state variables, it can be expressed compactly in an additive format in terms of a sum of indicator functions for each state variable. And not only can this independence be exploited in the problem representation, it can often be exploited in exact and approximate solution methods as well. Such techniques have permitted the practical solution of MDPs that would not have been possible using an enumerated state and action model [Hoey *et al.*, 1999; St-Aubin *et al.*, 2000; Guestrin *et al.*, 2002]. And as we will show in this thesis, there is an opportunity to exploit even more structure in the factored MDP model than could be exploited by previous algorithms.

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 (e.g., the BOXWORLD logistics problem refers to object classes such as Box, $City$, $Truck$, and $Plane$) and relations between those domain objects that may change over time (e.g., $BoxIn(Box : b, City : c)$, $TruckIn(Truck : t, City : c)$, $BoxOnTruck(Box : b, Truck : t)$, $BoxOnPlane(Box : t, Plane : p)$, $PlaneIn(Plane : p, City : c)$).[1] Often, relational objectives abstract over objects using quantification as in "get some truck to Paris" or "get all boxes to their destination". And relational action templates such as loading or unloading a box from a truck or plane are likely to apply generically to domain objects and thus can be specified independently of any ground domain instantiation. This domain-independent specification allows very compact MDP specifications when compared to a corresponding grounded propositional representation. For instance, ten each of boxes, trucks, planes, and cities leads to a combined 500 state variables corresponding to all ground atoms of the above five binary relations.

Unfortunately, while relational specifications permit very compact descriptions of a variety of MDPs, this efficiency has not traditionally translated to the corresponding solution methods. Most traditional solution approaches to relational decision-theoretic planning problems ground the relational specification w.r.t. a specific domain instantiation and then apply a so-

[1] Throughout the thesis all predicates (including unary predicates denoting domain object classes) are capitalized and all variables and constants are lowercased. We use the notation $C : v$ to denote that variable v is restricted to domain object class C.

lution approach directly to the resulting ground Markov decision process (MDP) [Puterman, 1994]. Unfortunately, the resulting solution is domain-specific and the space and time complexity of these grounded solution algorithms scale linearly with the domain size in the best case and exponentially in the worst case.

An alternate approach to grounding is to lift the relational planning problem to a first-order MDP (FOMDP) specification [Boutilier *et al.*, 2001]. This FOMDP can then be solved directly, resulting in a domain-independent solution whose space and time complexity do not scale with domain size. This approach is particularly attractive given that the FOMDP framework can be used to model many planning problems stated in PPDDL [Younes and Littman, 2004].[2] Furthermore, we can extend the set of FOMDP problems that can be succinctly specified to include factored FOMDPs with exogenous actions and additive rewards that both scale with the domain size.

Unfortunately, the expressivity and power that can be gained from converting a decision-theoretic planning problem to a FOMDP and obtaining a domain-independent solution does not come without its drawbacks. The introduction of first-order logical languages to describe FOMDPs introduces the need for logical simplification and theorem proving. Unfortunately, both of these tasks are difficult by themselves and they introduce significant complications that have been carefully worked around in the solution approaches covered in this thesis. For example, while the use of alternating existential and universal quantifiers may complicate the tasks of simplification and theorem proving, the solution methods used in this thesis exploit the fact that many practical problems do not make substantial use of this expressivity. As such, the high degree of regularity and structure inherent in many FOMDPs permits the application of solution methods that were not possible in purely ground approaches. Thus, along with the structural expressivity gains of the FOMDP representation comes the ability to efficiently exploit the structure laid bare by such expressivity in practice. To this end, this thesis continues the long-standing trends of exploiting structure in decision-theoretic planning tasks in the MDP framework by succinctly representing and (approximately) optimally solving relational decision-theoretic planning problems represented as FOMDPs.

[2]PPDDL is one of the the most popular probabilistic planning specification languages and incorporates elements of popular deterministic planning languages such as STRIPS [Fikes and Nilsson, 1971] and PDDL [McDermott *et al.*, 1998].

1.3 Major Contributions

Following are some of the major contributions of the thesis:

1. **Affine Algebraic Decision Diagrams:** The algebraic decision diagram (ADD) [Bahar *et al.*, 1993] is a data structure for representing functions from $\mathbb{B}^n \rightarrow \mathbb{R}$ and can be very compact when the underlying function demonstrates context-specific independence (CSI) [Boutilier *et al.*, 1996]. Furthermore, unary and binary operations on functions from $\mathbb{B}^n \rightarrow \mathbb{R}$ can often be computed efficiently by direct operations on the respective ADD representations. As such, ADDs provide an attractive alternative to the tabular representation of functions from $\mathbb{B}^n \rightarrow \mathbb{R}$ commonly used to represent factored MDPs and thus can be used in the solution of factored MDPs. These ADD-based solutions can be more efficient than direct manipulation of direct tabular representations if the factored MDP demonstrates CSI in the problem specification, the solution, or both. However, many MDPs exhibit additive and multiplicative structure as well as CSI. Prior to this thesis work, no data structure could generally simultaneously exploit all three types of structure.

   To remedy this deficiency, we specify a novel extension to the ADD data structure — the affine ADD (AADD) — for simultaneously exploiting additive, multiplicative and context-specific independence in factored MDP representation and solution methods. We prove that the AADD never performs more than a constant factor worse in time and space than an ADD and can lead to an exponential-to-linear reduction in time and space over the ADD. We present a variety of empirical results suggesting that AADDs are often as good as or better than ADDs or tabular representations in the solution of factored MDPs.

2. **First-order Decision Diagrams:** We specify a first-order extension of both the ADD and AADD data structures that can be used to replace the case representation and operations used for FOMDPs. In doing this, we present a first-order extension of CSI that can be exploited in the solution of FOMDPs. FOADDs and FOAADDS permit the compact representation of FOMDP value functions and policies and help maintain simplified representations that reduce the theorem proving burden on the solution algorithm. The use of these first-order decision diagrams combined with techniques for simplifying first-order formulae permit the fully automated solution of basic FOMDPs.

3. **Additive Decomposition of Universal Rewards:** Universally quantified rewards are known to make FOMDPs extremely difficult to solve [Gretton and Thiebaux, 2004]. As

a heuristic alternative to the direct solution of a FOMDP with universal reward, we show how to additively decompose universal reward specifications in a manner that leads to efficient FOMDP solutions and reasonable performance characteristics on a variety of test problems.

4. **Linear-value Approximation for FOMDPs:** We show how to generalize linear-value approximation techniques for factored MDPs [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001] to the case of FOMDPs, along with generalized loss-bounds on the approximation. We also define a linear program (LP) with first-order constraints and contribute an efficient constraint generation algorithm that exploits the constraint structure to efficiently solve the LP.

5. **Representation and Solution of Factored FOMDPs:** We contribute the factored FOMDP extension to model FOMDPs with factored actions and additive rewards that scale with the domain size. We also contribute some extensions to symbolic dynamic programming and linear-value approximation techniques to efficiently solve factored FOMDPs in special cases. While the linear-value approximation algorithms that we introduce are specific to a domain-instantiation, we demonstrate an example where a solution equivalent to those obtained by ground methods can be obtained in time and space that scales sub-linearly in the domain size — a result that is impossible to obtain with grounded solution techniques.

6. **Correctness of Symbolic Dynamic Programming:** We provide a formal proof of correctness of symbolic dynamic programming (SDP) for FOMDPs. The key to this proof is showing that when an SDP solution to FOMDPs is grounded w.r.t. a domain closure assumption, the result is equivalent to the solution obtained by first grounding the FOMDP and then applying standard ground MDP solution techniques.

   This is a different proof approach from the original given in [Boutilier *et al.*, 2001]. There the emphasis was on proving the correctness of the SDP algorithm at a purely logical level (including the case of infinite models). In our proof, we focus on making a domain closure assumption (and thus implicitly, a finite model assumption) and proving correspondence between the first-order and well-known ground MDP solutions.

1.4 Distinction from Related Work

We note there has been a great deal of recent work in relational forms of decision-theoretic planning [Hölldobler and Skvortsova, 2004; Karabaev and Skvortsova, 2005; Kersting *et al.*, 2004; Wang *et al.*, 2007; Wang and Khardon, 2007; Fern *et al.*, 2003; Gretton and Thiebaux, 2004; Guestrin *et al.*, 2003]. We will discuss these alternate approaches at the appropriate point in future chapters, but for now we simply note that all related work demonstrates one or both of the following limitations in comparison to this thesis work and its foundations [Boutilier *et al.*, 2001]:

1. No other exact solution algorithm applies to FOMDPs with *both* universal and existential quantifiers.

2. Other approximate solution approaches rely on domain instance sampling and must scale at least linearly with the size of these sampled domain instances. As a consequence, these algorithms cannot scale to arbitrarily large sampled domain sizes and can only provide error bounds (if any) that grow proportionally to the domain size.

As such, this thesis work proposes the only exact and approximate solution approaches that can handle FOMDPs with both existential and universal quantifiers while scaling independently of the domain size for the case of FOMDPs or potentially sublinearly in the domain size for factored FOMDPs. Furthermore the approximate solution techniques that we propose permit the computation of error bounds that apply *uniformly to all domain sizes*.

1.5 Outline

The thesis proceeds as follows. In Chapter 2, we review the basic MDP model and motivate the importance and generality of the MDP as a model for decision-theoretic planning. We also present a variety of standard and approximate solution techniques for MDPs.

In Chapter 3, we introduce background material on factored MDPs relevant to the thesis. This includes demonstrating how the structure of a factored MDP representation can be exploited to avoid full state enumeration and how a variety of exact and approximate solution algorithms can exploit this structure for purposes of space and computational efficiency. Next, we introduce the first contribution of this thesis, the Affine Algebraic Decision Diagram [Sanner and McAllester, 2005], which permits the simultaneous exploitation of context-specific, additive and multiplicative independence in factored MDPs.

In Chapter 4 we begin by introducing the first-order MDP (FOMDP) formalism and the symbolic dynamic programming solution approach as originally defined in [Boutilier *et al.*, 2001]. We then introduce a simple procedure for generalizing the propositionally-based ADDs and AADDs to first-order (FO) versions that we respectively denote as FOADDs and FOAADDs. We then show how these first-order decision diagrams can be used in place of case statements to exploit structure in the basic FOMDP solution algorithms and provide simple empirical results for this approach. Faced with the difficulty of solving problems with universally specified rewards, we conclude the chapter by proposing an additive decomposition solution to this problem.

Perhaps the greatest difficulty with the value iteration technique proposed in Chapter 4 is that the value function representations tend to involve extremely complex formulae that cannot be easily simplified. The inability to simplify often leads to a combinatorial explosion in the size of the value function or policy. This typically prohibits the exact solution of relatively simple FOMDPs so in Chapter 5 we seek alternate approaches based on linear-value approximation. In this paradigm, we reduce the task of solving a FOMDP to that of obtaining good weights for a set of basis functions that approximates the optimal value function. In this chapter, we describe the basic generalization of these techniques from the factored case to the first-order case and also provide a much-needed technique for automatic basis function generation based on the work of Gretton and Thiebaux [2004].

In Chapter 6, we extend the symbolic dynamic programming framework for first-order MDPs (FOMDPs) to handle sum/product aggregators and factored actions required to represent factored FOMDPs. To motivate the need for each of these extensions, we begin by describing various scenarios where each new construct is required along with the formal semantics of these constructs. Once we have specified the semantics, we proceed to generalize symbolic dynamic programming (SDP) to handle FOMDPs with these additional constructs. Noting that a number of intractability issues arise with SDP, we then introduce appropriately generalized approximate linear programming and approximate policy iteration algorithms for efficient linear-value approximation in the presence of sum/product aggregators and factored actions.

We conclude in Chapter 7 with a summary of the thesis and some interesting directions for future work.

Chapter 2

Markov Decision Processes

The Markov decision process (MDP) model was first introduced in the field of operations research [Bellman, 1957] and significantly developed in subsequent years [Howard, 1960]. An excellent recent text on MDPs is that of Puterman [1994]. 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].

In the MDP model we use in this thesis, an agent is allowed to fully observe the current state and choose an action to execute from that state. Based on that state and action, Nature then chooses a next state according to some fixed probability distribution and the agent receives a corresponding reward. This process repeats itself for some horizon of time steps, possibly infinite. The goal of the agent is to choose its actions so as to maximize the sum of expected discounted future rewards in this model.[1]

Given this high-level description of the MDP model, we now proceed to provide a more detailed mathematical definition of an MDP followed by a description of various algorithmic approaches for making optimal sequential decisions in this model. Except where otherwise noted, the following presentation derives from Puterman [1994].

2.1 MDP Representation

Formally, a finite state and action MDP is specified by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, h, \gamma \rangle$. We now describe each of these components in turn, noting that in practice, each must be specified by a domain expert or learned from data.

[1] While an agent may seek to maximize other objectives in a general MDP model, we focus on maximizing the sum of expected discounted reward in this thesis.

State space $\mathcal{S}$

The world is modeled by a set of distinct states $\mathcal{S}$. In the most general MDP models, $\mathcal{S}$ can be infinite or continuous, but throughout the thesis, we assume a discrete (possibly infinite) state space.

Action space $\mathcal{A}$

An agent in an MDP can effect changes to its state by executing actions from the set $\mathcal{A}$. In more general MDP models, $\mathcal{A}$ can be infinite or continuous, but again, we assume a discrete (possibly infinite) action space throughout the thesis. Actions are the only way that an agent can interact with the state and thus the choice of action to take in each state comprises the main decision-theoretic task of the agent.

Transition function $\mathcal{T}$

In an MDP model, the effects of actions can be uncertain such that for any action $a \in \mathcal{A}$ executed, the world has a fixed probability distribution over transitions to any state in $\mathcal{S}$. For the purpose of this thesis, the transition function $\mathcal{T}$ will be modeled as a probability distribution $\mathcal{T}(s, a, s') = P(s'|a, s)$, which denotes the probability that the world makes a transition from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ given that action a was executed in state s. We note that this representation of the transition function satisfies the Markovian assumptions of an MDP, which require that 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} for $i \geq 1$ given the state s_t and the action a_t taken at time t.

While we typically think of the transition function as dependent only upon the agent's action and the state from which it was taken, there can also be *exogenous events* that are not directly influenced by the agent. For example, as discussed in the SYSADMIN problem in Chapter 1, any computer not explicitly rebooted can independently fail according to some probability distribution. In order to model such exogenous events in this thesis, we will simply fold these implicit probabilistic effects into the transition distribution for each action.

Reward function $\mathcal{R}$

The preferences of the agent are encoded in a reward function, which for the purpose of this thesis will be restricted to a real-valued range, that is $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. This form of reward function is much more flexible than goal-oriented notions in classical planning; for example,

one can easily model multiple objectives and decision-theoretic reward tradeoffs using different reward values for different states and actions. In a classical planning model, one is typically restricted to specifying a set of equally preferred goal states with state-independent action costs.

Horizon h and discount factor γ

In an MDP, the objective of the agent will be to maximize expected utility accumulated over some time horizon h representing the number of decision steps until termination. While we cover the case for finite h in this chapter, for all subsequent chapters of the thesis, we will assume $h = \infty$ unless otherwise noted.

In the calculation of accumulated reward, we allow for the discounting of rewards t time steps into the future by a discount factor γ^t where $\gamma \in [0, 1]$. Throughout this thesis, we will assume that $\gamma < 1$ unless specifically noted. 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. Such a discounting assumption has both an economic justification as well as an implicit modeling justification for a process that has a $1 - \gamma$ probability of terminating at each step.

Practically, $\gamma < 1$ is required to ensure that the total expected reward is bounded in the case of infinite horizon MDPs. However, if we can make the assumption that the only non-zero reward states in our MDP model are a set of goal states and the system transitions into a zero-reward absorbing state after reaching a goal state, then we can use $\gamma = 1$ in the infinite horizon setting since the total future reward is guaranteed to be bounded.

2.2 Policy Representation

The goal of an agent is to take the action in each state that maximizes the expected accumulated discounted reward criterion over a specified time horizon h. A sequence of actions to be taken can be specified as $\langle \pi_h, \pi_{h-1}, \ldots, \pi_1 \rangle$ where each $\pi_t : \mathcal{S} \rightarrow A$ is a time-dependent action *policy* that specifies an action to take from each state s_t with t-stages-to-go. An important result following from the Markovian property of MDPs is that any policy conditioned on the state or action history from previous decision stages can be represented by an equivalent policy conditioned on only the current state. This follows from the fact that the fully observed state at any stage renders the previous history irrelevant.

An optimal policy $\langle \pi_h^*, \pi_{h-1}^*, \ldots, \pi_1^* \rangle$ is a sequence of action policies to be taken that max-

imize the agent's total expected discounted reward over horizon h. Conveniently, for the case of $h = \infty$, there always exists an optimal stationary policy [Howard, 1960]. Thus, no loss of expected discounted reward is incurred for infinite horizon MDPs by restricting our policy representation to a single policy π denoting the action to take from all states at all time stages.

2.3 Optimal Solution Criteria

If the agent's objective is to find the policy that maximizes the expected sum of discounted rewards over a specified time horizon, this objective can be formally expressed as

$$E_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t | s_0 \right] \tag{2.1}$$

where r^t is a reward obtained at time t, γ is a discount factor as defined above, π is a policy as defined previously, and s_0 is the initial starting state. Based on this reward criterion, we define the *value function* for a policy π as the following:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t \;\middle|\; s_0 = s \right]. \tag{2.2}$$

Intuitively, the value function for a policy π is the expected sum of discounted rewards accumulated while executing that policy when starting from state s.

A *greedy policy* π_V w.r.t. a value function V is simply the action policy that takes an action in each state that maximizes expected value w.r.t to V defined as follows:

$$\pi_V(s) = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right\} \tag{2.3}$$

Thus, from any value function, we can derive a corresponding greedy policy that represents the best action choice w.r.t. that value estimation.

An *optimal policy* π^* in an infinite horizon MDP maximizes the value function for all states. An optimal policy π^* is the greedy policy w.r.t. an optimal value function V^* and likewise the optimal value function is the value under an optimal policy, $V_{\pi^*}(s) = V^*(s)$. We note that V^*

satisfies the following fixed-point equality:

$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') \cdot V^*(s') \right\}. \tag{2.4}$$

2.4 Exact Solution Techniques

In this section we will discuss exact solution techniques primarily for the case of infinite horizon MDPs. Before we discuss these techniques though, we introduce an alternative matrix notation for MDPs that will simplify portions of the following presentation.

2.4.1 Vector and Matrix Notation

We sometimes write the MDP in vector and matrix form. For each $a \in \mathcal{A}$, we can represent the reward $R(s,a)$ as a column vector R_a indexed by state $s \in \mathcal{S}$. We can represent the value function $V(s)$ as a column vector V indexed by state s. And we can represent the transition function $T(s,a,s')$ for each action $a \in \mathcal{A}$ as a transition matrix T_a row-indexed by current state s and column-indexed by next state $s' \in \mathcal{S}$. In this case, equation 2.4 can be restated as the following:

$$V^* = \max_\pi \{R_\pi + \gamma T_\pi V^*\} \tag{2.5}$$

In some cases, we will refer to the reward vector and the transition matrix with respect to a policy π as R_π and T_π, respectively; here the reward value and transition probability for each state corresponds to the action choice indicated by π. Or we may refer to the reward vector and transition matrices restricted to a specific action $a \in \mathcal{A}$ as R_a and T_a, respectively. If needed, π itself can be represented as a vector of actions $a \in \mathcal{A}$ indexed by state and we let Π denote the set of all possible policy vectors.

2.4.2 Dynamic programming

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

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

$$V_\pi^t(s) = R(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s,\pi(s),s') \cdot V_\pi^{t-1}(s') \tag{2.6}$$

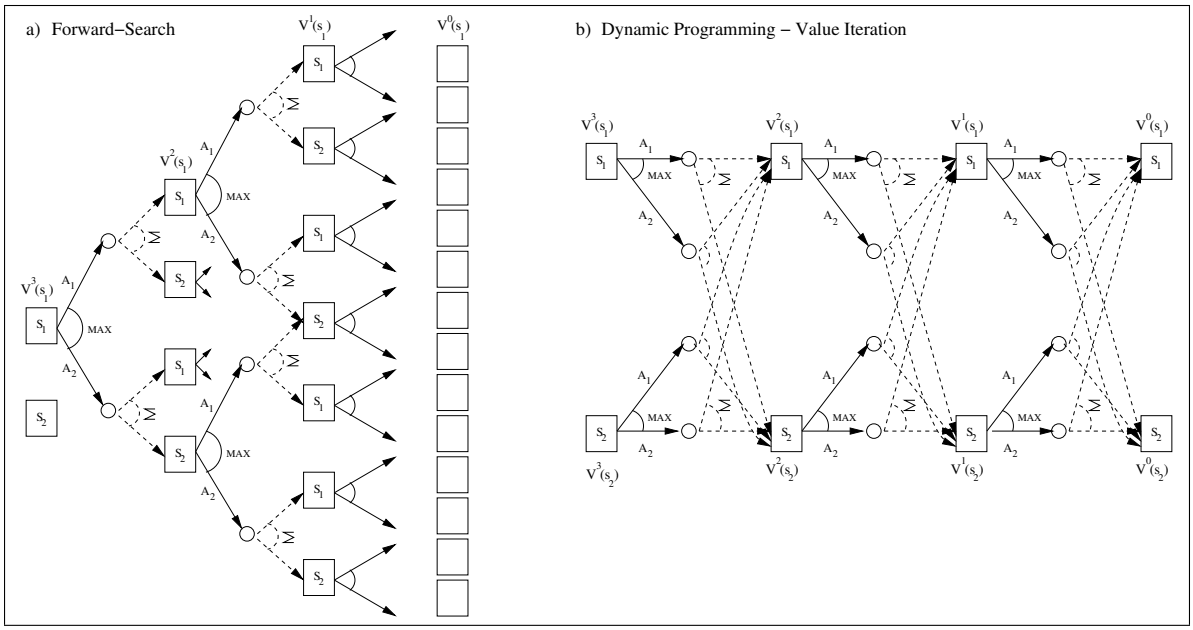


Figure 2.1: A diagram demonstrating a) forward evaluation of the MDP value function and b) dynamic programming regression evaluation of the MDP value function. Both methods return the same value for $V^3(s)$, but the forward evaluation requires exponential time in the search depth $O((|\mathcal{S}| \cdot |\mathcal{A}|)^d)$ and only calculates the value for one initial state whereas dynamic programming caches its results on each backup thus requiring only polynomial time in the search depth $O(|\mathcal{S}| \cdot |\mathcal{A}| \cdot d)$ and solving for the value function at *every* state.

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 $t-1$:

$$V^{t,*}(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') \cdot V^{t-1,*}(s') \right\} \tag{2.7}$$

Value iteration

We start with an algorithm known as value iteration that directly implements Equation 2.7. Here, we start with $V^0(s) = \max_a R(s,a)$ and perform the Bellman backup given in Equation 2.7 for each state $V^1(s)$ using the value of $V^0(s)$. We repeat this process for each stage t, producing the backed up value function for $V^t(s)$ from $V^{t-1}(s)$ until we have computed the intended t-stage-to-go value function. This algorithm is demonstrated graphically in Figure 2.1(b).

Often, the Bellman backup is rewritten in two steps to separate out the backup of a value function through a single action and the maximization of this value over all actions. In this

case, we first compute the t-stage-to-go Q-function for every action and state:

$$Q^t(s,a) = R(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} T(s,a,s') \cdot V^{t-1}(s') \tag{2.8}$$

Then we maximize over each action to determine the value of the regressed state:

$$V^t(s) = \max_{a \in A} \left\{ Q^t(s,a) \right\} \tag{2.9}$$

This is clearly equivalent to equation 2.7 but is in a form that we will refer to later since it separates the algorithm into its two conceptual components.

Puterman [1994] shows that terminating once the following condition is met

$$\|V^t - V^{t-1}\|_\infty < \frac{\epsilon(1-\gamma)}{2\gamma} \tag{2.10}$$

guarantees ϵ-optimality, i.e., $\max_s |V^t(s) - V^*(s)| < \epsilon$. Thus, the greedy policy derived from V^t iteration loses no more than ϵ in value over the infinite horizon in comparison to the optimal policy.

We note that the value iteration approach requires time polynomial in the search depth d, i.e., $O(|S| \cdot |A| \cdot d)$, and solves for the value function at *every* state. Puterman [1994] provides a proof that value iteration converges at a linear rate in terms of the number of iterations.

Policy iteration

At each step of the value iteration backup, we are implicitly performing a policy update, determining the best action to take from every state in order to maximize reward. Another approach to dynamic programming is known as policy iteration [Howard, 1960] and is summarized in the following algorithm:

1. *Initialization:* Pick an arbitrary initial decision policy $\pi_0 \in \Pi$ and set $i = 0$.

2. *Policy Evaluation:* Solve for V_{π_i} (see below).

3. *Policy Improvement:* Find a new policy π_{i+1} that is a greedy policy w.r.t. V_{π_i} (i.e., $\pi_{i+1} \in \arg\max_{\pi \in \Pi} \{R_\pi + \gamma T_\pi V_{\pi_i}\}$ with ties resolved via a total precedence order over actions).

4. *Termination Check:* If $\pi_{i+1} \neq \pi_i$ then increment i and go to step 2 else return π_{i+1}.

We note that the policy evaluation of a *fixed* policy π reduces to the solution of a linear system since the MDP reduces to a simple Markov chain. Thus, we can solve for V_π by computing the right-hand side of the following equation:

$$V_\pi = R_\pi (I - \gamma T_\pi)^{-1} \tag{2.11}$$

We note that a unique solution for V_π always exists since the Markovian properties of T_π guarantee that $I - \gamma T_\pi$ is invertible. We note that solving for V_π directly using matrix inversion takes time $O(|S|^3)$. Alternately, we can solve for V_π using *successive approximation*, which initializes $V_\pi^0 = R_\pi$ and iteratively computes V_π^t from V_π^{t-1} using Equation 2.6 until $V_\pi^t = V_\pi^{t-1}$ (where $V_\pi = V_\pi^t$).

Once policy iteration has terminated, the final policy returned is the optimal policy π^* and the value function corresponding to this policy is the optimal value function V^*. Puterman [1994] provides conditions and a proof of a superlinear rate of convergence for policy iteration.

So far, we have implicitly assumed that the above algorithms perform synchronous updates, that is, we are updating the value function in value iteration for all states and that we are improving the policy in policy iteration for all states. We additionally note that there are a number of asynchronous variants of value and policy iteration that do not update the value or improve the policy at every state on all iterations, yet still retain similar convergence properties. These algorithm variants are discussed by Puterman [1994] and Bertsekas and Tsitsiklis [1996] and are extremely useful for proving convergence properties of the reinforcement learning [Barto and Sutton, 1998] and real-time search [Barto *et al.*, 1993] approaches to solving MDPs. However, we do not discuss asynchronous methods further as they are not directly relevant to the methods we employ throughout the rest of the thesis.

Modified policy iteration

A comparison of the two previous algorithms reveals that they occupy two extremes in terms of policy updates: value iteration performs an implicit policy update in order to compute every intermediate value function whereas policy iteration performs an update only after solving directly for V_π.

If we interpolate between these two approaches, we arrive at an algorithm known as modified policy iteration [Puterman and Shin, 1978]. In this algorithm, we simply iterate between policy evaluation and policy improvement phases until our policy is ϵ-optimal using the same

terminating criteria as value iteration. The algorithm is very similar to policy iteration with the exception of the policy evaluation phase replaced by an approximate version:

1. *Initialization:* Pick an arbitrary initial decision policy vector $\pi_0 \in \Pi$ and set $i = 0$.

2. *Approximate Policy Evaluation:* Solve for V_{π_i} using some number of steps of successive approximation.

3. *Policy Improvement:* Find a new policy π_{i+1} that is the greedy policy w.r.t. V_{π_i}.

4. *Termination Check:* If $\pi_{i+1} \neq \pi_i$ then increment i and go to step 2 else return π_{i+1}.

Algorithm convergence requires only that the policy approximation phase does not increase the error of the value estimate from the previous iteration, i.e.,

$$\|V^* - V_{\pi_{i+1}}\| \leq \|V^* - V_{\pi_i}\| \tag{2.12}$$

Such a property holds, for example, by initializing the value estimate with V_{π_i} and then performing one or more steps of successive approximation under the policy π_{i+1}.

A proof of superlinear convergence rate for modified policy iteration under certain conditions is given by Puterman [1994]. Puterman also notes that modified policy iteration often empirically requires less computation time than both value and policy iteration.

2.4.3 Forward-search

If we reexamine Equation 2.7, we note that we could compute this recurrence in a forward-search manner by starting at an initial state and unfolding the recurrence to horizon h and then computing the expectation and maximization as we return to the initial state. A graphical representation of the unfolding of this computation is shown in Figure 2.1(a). We note that determining the value $V^h(s)$ for a *single* state using this method requires time exponential in the search depth h, that is, $O((|\mathcal{S}| \cdot |\mathcal{A}|)^h)$.

Since we are performing forward search to a fixed *a priori* search depth, we can determine the minimum horizon h to search if we want an ϵ bound on the maximum error of our value function, given knowledge of our discount factor γ and our maximum reward $R_{max} = \max_{s,a} R(s, a)$:

$$h \geq \log_\gamma \left(\frac{\epsilon(1-\gamma)}{R_{max}} \right) - 1 \tag{2.13}$$

2.4.4 Real-time dynamic programming

The real-time dynamic programming (RTDP) framework [Barto *et al.*, 1993] is a hybrid approach that combines real-time forward search with dynamic programming. This approach uses limited depth, forward-search backups to update the value function of the set of states visited during on-line trials, assuming that initial states were generated according to some fixed distribution. The policy used for the trials is the optimal policy for the current value function. Since backed-up and cached values from one step are used by other steps, this approach mixes the forward-search and dynamic programming paradigms. It is provably convergent and has the advantage that it only derives the value function for the set of states reachable from the initial state distribution. This can often be more efficient than synchronous dynamic programming approaches when the set of reachable states is small compared to the total number of states.

2.4.5 Linear programming

An MDP can also be solved by formulating it as the optimization of a linear program (LP). The fact that such a solution exists follows from the notion that the optimal policy and value function must satisfy the following inequalities for all states as implied by Equation 2.4:

$$V^*(s) \geq \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right) ; \ \forall s \in \mathcal{S} \tag{2.14}$$

This equality in turn implies the following conditions:

$$V^*(s) \geq R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s'); \ \forall a \in \mathcal{A}, s \in \mathcal{S} \tag{2.15}$$

While this latter set of inequalities only enforces one side of the optimal value function fixed-point equality given in Equation 2.4, it turns out that finding the minimal V^* under an $\mathcal{L}_1$ metric that satisfies these constraints suffices to enforce the other side of the inequality. Thus, the optimal value function can be computed by the following primal specification of a linear

program [Puterman, 1994]:

$$\begin{aligned} \text{Variables:} \quad & V \\ \text{Minimize:} \quad & \|V\|_1 \\ \text{Subject to:} \quad & 0 \geq R_a + \gamma T_a V - V, \ \forall a \in \mathcal{A} \end{aligned} \tag{2.16}$$

Puterman [1994] provides a proof that this formulation is guaranteed to produce an optimal value function for an MDP. Puterman also notes that solving the dual LP formulation is often more efficient than solving the primal LP formulation. However, we do not present the dual formulation here as we work directly with the primal formulation and its variants throughout the thesis.

2.5 Approximate Solution Techniques

As the number of states and actions in an MDP grows, it often becomes necessary to explore approximate solutions in the face of intractability of exact solutions. While approximation in MDPs can take many forms, it is frequently carried out by considering restricted representations of the value function. Some methods for restricting the value function representation will become relevant once we introduce structured descriptions of our MDP models and solution algorithms. However, a very 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, 1999a; Koller and Parr, 1999b; Schuurmans and Patrascu, 2001; Guestrin *et al.*, 2002], which we discuss at length in this section.

2.5.1 Linear-value Function Representation

Representing value functions as a linear combination of basis functions has many convenient computational properties, many of which will become evident as we incorporate factored and relational structure in our MDP model. However, perhaps one of the most important aspects for the work we present here is that linear-value function representations lead to MDP solution formulations using optimization w.r.t. linear objectives and linear constraints — that is, the well-studied case of linear program (LP) optimization. Since many robust off-the-shelf LP solvers are available, this makes such approaches attractive for practical implementation purposes.

If we have n states in our 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 function vectors denoted $b_i(s)$ as follows:

$$\tilde{V}_{\vec{w}}(s) = \sum_{i=1}^{k} w_i \cdot b_i(s) \tag{2.17}$$

The linear subspace spanned by the basis set might not include the actual value function, but one can use projection methods to minimize some error measure between the actual 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.*, 2002a; Mahadevan, 2005]). We will consider more structured forms of automated basis function construction as we introduce structured MDP representations in subsequent chapters.

On a final note, we mention that there are a variety of other general function approximation such as nonlinear functions or neural nets [Bertsekas and Tsitsiklis, 1996] but it is generally difficult to provide useful convergence properties for such approximation architectures so we do not discuss them further in this thesis.

2.5.2 Error Bounds on Approximate Value Functions

Once a set of basis functions has been specified, the problem of finding an approximate value function reduces to the problem of finding a good set of weights that closely approximates the optimal value function. One way of measuring the *a posteriori* quality of an approximated value function $\tilde{V}_{\vec{w}}$ is by evaluating the Bellman error β (i.e., the L_∞ norm of the Bellman residual) of the value function under the MDP dynamics:

$$\beta = \max_{s \in \mathcal{S}} \left| \tilde{V}_{\vec{w}}(s) - \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \tilde{V}_{\vec{w}}(s') \right) \right| \tag{2.18}$$

Of course we note that when the Bellman error is zero, this equation satisfies the fixed-point equation for the optimal value function given in Equation 2.4 and thus $\beta = 0$ indicates that $\tilde{V}_{\vec{w}} = V^*$.

Let $\tilde{\pi}$ be the greedy policy w.r.t. the value function approximation $\tilde{V}_{\vec{w}}$. Once β is known for $\tilde{V}_{\vec{w}}$, it is then easy to bound the max-norm ($\mathcal{L}_\infty$) error of $V_{\tilde{\pi}}$ w.r.t. the optimal value function

using the following inequality [Williams and Baird, 1994]:

$$\|V^* - V_{\tilde{\pi}}\|_\infty \leq \frac{2\gamma\beta}{1-\gamma} \tag{2.19}$$

Thus, in all of the following approximation techniques, we will have some way of determining a maximum bound on the loss of our approximation.

2.5.3 Approximate Dynamic Programming

Approximate dynamic programming techniques are simply extensions of the previous dynamic programming algorithms with additional approximation steps. While these approximation steps do not guarantee convergence, an *a posteriori* analysis of the Bellman error of a value function can show that the value function estimate has converged within some error bound.

Approximate Value Iteration

Approximate value iteration (AVI) is precisely the value iteration algorithm previously presented with the additional step that after each Bellman backup, the value function may be projected onto a more compact representation. Since we are focusing on linear-value function approximation in this section, we will cover the case of projecting the one-step Bellman backup onto a linear-value function representation.

In AVI using linear-value approximation, we begin by initializing the weights $\vec{w}^0$ of our initial linear-value function representation $\tilde{V}^0_{\vec{w}}$ in some way — perhaps with $\vec{w}^0 = \vec{0}$ or with $\vec{w}^0$ set so that $\tilde{V}^0_{\vec{w}} = \max_a R_a$ (if our linear-value function representation permits this). Then we perform the standard Bellman backup given in Equation 2.7 to obtain V^1. Since the dynamics of our MDP do not guarantee that our linear-value function representation spans the space of V^1, it will be necessary to project V^1 onto the space of $\tilde{V}^1_{\vec{w}}$, which we discuss in a moment. This process can proceed indefinitely in AVI, obtaining V^t from $\tilde{V}^{t-1}_{\vec{w}}$ and projecting V^t to obtain $\tilde{V}^t_{\vec{w}}$ until some predefined stopping criterion such as a maximum limit on iterations or Bellman error bound has been met.

Perhaps the most obvious choice for projecting V^t to obtain $\tilde{V}^t_{\vec{w}}$ in AVI is the following where $\vec{w}^*$ represents the weights for the optimal projection and n is the error norm $\mathcal{L}_n$ being minimized in the projection:

$$\vec{w}^* = \arg\min_{\vec{w}} \left\| V^t - \tilde{V}^t_{\vec{w}} \right\|_n \tag{2.20}$$

Tsitsiklis and Van Roy [1996] show that minimizing the Euclidean-distance ($\mathcal{L}_2$) error can diverge — even when $\tilde{V}^t_{\vec{w}}$ spans the space of the optimal value function. Likewise, Guestrin, Koller, and Parr [2001] discuss similar issues with the divergence of AVI for the case of the max-norm ($\mathcal{L}_\infty$) error minimizing projection. However, these divergence issues can be mitigated in practice if additional basis functions are introduced to minimize the projection error.

Approximate Policy Iteration

Approximate policy iteration (API) with linear-value function approximation is another variant of dynamic programming that uses a different projection step. The benefit of API is that under an $\mathcal{L}_\infty$ projection step, its error can be shown to be bounded for all iterations, thus avoiding the divergence issues of AVI [Guestrin *et al.*, 2001].

The API algorithm follows the policy iteration algorithm provided previously, except that the value determination step is now approximate rather than exact. After starting with an initial arbitrary policy π_0, policy iteration iterates between the following two steps where the projection is in terms of the $\mathcal{L}_n$ norm:

$$\vec{w}^i = \arg\min_{\vec{w}} \left\| R_{\pi_i} + T_{\pi_i} \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}} \right\|_n \tag{2.21}$$

$$\pi_{i+1} = \arg\max_{\pi \in \Pi} \left\{ R_\pi + \gamma T_\pi \tilde{V}_{\vec{w}^i} \right\} \tag{2.22}$$

Koller and Parr [1999b] provide an API algorithm based on minimizing a weighted $\mathcal{L}_2$ norm in the projection step. In subsequent work, Guestrin, Koller and Parr [2001] presented the following LP intended to directly minimize the $\mathcal{L}_\infty$ norm in the projection step:

$$\begin{aligned} &\text{Variables: } \vec{w} \\ &\text{Minimize: } \beta \\ &\text{Subject to: } \beta \geq \left| R_{\pi_i} + T_{\pi_i} \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}} \right| \end{aligned} \tag{2.23}$$

One nice advantage of directly minimizing the L_∞ norm in the projection step is that when API converges (i.e., $\pi_i = \pi_{i-1}$ or equivalently $\vec{w}^i = \vec{w}^{i-1}$), the objective β for the final LP solution of Equation 2.23 is the Bellman error of the approximated value function. Thus a bound on the error of the approximated value function is immediately available by plugging β directly into Equation 2.19 [Guestrin *et al.*, 2001].

2.5.4 Approximate Linear Programming

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 and thus the linear programming framework can still be used. Consequently, we arrive at the following variant of the LP in Equation 2.16 that simply takes into account the linear-value function representation:

$$\begin{aligned} \text{Variables:} \quad & \vec{w} \\ \text{Minimize:} \quad & \|\tilde{V}_{\vec{w}}\|_1 \\ \text{Subject to:} \quad & 0 \geq R_a + \gamma T_a \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}},\ \forall a \in \mathcal{A} \end{aligned} \tag{2.24}$$

2.6 Application to AI Planning Problems

We focus on MDPs as a model for decision-theoretic planning since they generalize many of the planning paradigms found in the literature. First we review some of these planning paradigms and then proceed to a discussion of two general classes of MDP problems, one oriented towards a decision-theoretic extension of classical task-oriented planning and the other oriented towards a non-terminating process model with a long-tern reward optimization objective, but no clear definition of a single task or goal.

2.6.1 Common AI Planning Paradigms

As mentioned previously, classical planning can be viewed as a restricted case of decision-theoretic planning in MDPs where all actions are deterministic and the reward is goal-oriented, that is, there is only one non-zero reward value that is specified for a set of absorbing goal states. Typically the initial state is known, thus making observability a moot issue — with a known initial state and deterministic actions, the state of the world after *any* action sequence will be known with certainty.

In classical planning the objective is simply to find a sequence of actions that will lead to a goal state from the initial state. There may be an emphasis placed on finding shorter plans, or more generally there may be costs associated with actions and the use of an objective criterion that minimizes cost-to-goal. Nonetheless, all of these variants can be modeled in the MDP

framework. However, this does not mean that standard MDP solution algorithms are particularly well-suited for classical planning; while standard exact MDP solution algorithms will provide an optimal policy in the case of classical planning, this optimal policy is provided for *all* states. However due to the known initial state and determinism of action effects, solutions to classical planning can be specified via straight-line sequences of actions that may touch on only a very small subset of the total state space. Thus, the full policy provided by exact MDP solution algorithms will be inefficient compared to deterministic planners in the classical planning paradigm that can exploit knowledge of the initial state and action constraints to avoid searching through all states. Weld [1999] provides an excellent overview of many recent advances in classical deterministic AI planning along these lines.

A related topic is that of optimal deterministic planning, which uses a similar framework as classical AI planning (i.e., known initial state and deterministic action effects), but relaxes the goal-oriented notions to a much richer set of preferences over goals and resource constraints (see e.g., [Haddawy and Hanks, 1998; Williamson and Hanks, 1994; Brafman and Chernyavsky, 2005]) and even temporally extended preferences (see e.g., [Bienvenu *et al.*, 2006; Baier *et al.*, 2007] for some recent work and a discussion of related approaches in this area). The task here is to find an optimal plan that takes into account the preferences and constraints. Since these approaches use a rich notion of preferences and assumptions, there does not necessarily exist a direct correspondence to the scalar-reward MDP framework discussed in this chapter. Nonetheless, notions of reward in the MDP framework defined here can capture some aspects of optimal deterministic planning.

A number of planning problems in AI involve partial observability and thus cannot be solved in the MDP framework presented here. Two notable problems are variants of conformant planning. In conformant planning [Cimatti and Roveri, 1999] the initial state is restricted, but strictly unknown and actions have non-deterministic effects with no (or in some variants, partial) observability. Probabilistic conformant planning is similar except that strict uncertainty in the initial state and action effects are replaced with known probability distributions [Kushmerick *et al.*, 1995]. Nonetheless, the partial observability assumptions of conformant planning and many other partially observable problems prevent them from being modeled or solved within the MDP framework presented here.

2.6.2 Task- vs. Process-oriented Planning

Most classical AI planning problems exhibit the characteristic of being goal-oriented, even when there are multiple goals and relative preferences over those goals. The BOXWORLD problem from Chapter 1 is a good example of a such a task-oriented problem: there are a number of boxes that need to be delivered to their destination and once this is achieved, the problem terminates. While many task-oriented decision-theoretic planning problems can be modeled as MDPs with some form of absorbing goal state, this is only one possible class of problems.

There are many problems that are continuous processes without a clearly defined notion of goal or termination, but rather a continuously accumulating reward over an infinite horizon. The SYSADMIN problem from Chapter 1 is an exemplar of this class of problems. Recalling the SYSADMIN description, the objective was to maximize the count of computers running per time step under an infinite horizon discounted reward criterion. However, given that any computer can independently fail at any time step if not rebooted due to exogenous events, the task has no clear criterion for termination since no state can persist indefinitely. Fortunately, this ongoing process-oriented problem is well-modeled as the optimization and solution of an infinite-horizon discounted reward MDP.

As mentioned in Boutilier *et al.* [1999], many real-world problems exhibit both task- and process-oriented behavior. And the beauty of the MDP framework is that it can accommodate both forms of MDP models and it can seamlessly combine them, if needed. Thus, we can not only accurately model decision-theoretic planning problems based on the classical task-oriented paradigm, but we can encapsulate these task-oriented problems in a more realistic ongoing optimization process with random exogenous events. These types of combined task- and process-oriented models more accurately reflect the problems than an agent would likely have to contend with while acting in a realistic world model.

2.7 Summary

We have motivated the decision-theoretic planning paradigm and cast the framework in an MDP setting. And we have covered all of the groundwork for the MDP solution techiques that we develop in this thesis. Among these solutions, there are two important choices to consider. The first choice is whether to use iterative dynamic programming methods or direct linear program optimization techniques. The second choice is whether to use exact or approximate

solution methods.

It is not entirely clear when to use dynamic programming algorithms vs. direct linear program optimization techniques. While Puterman [1994] cites Koehler [1976] in reporting that dynamic programming based modified policy iteration techniques can outperform direct linear programming techniques by as much as 10 times, Trick and Zin [1997] report exactly the opposite case, perhaps owing to their use of the more recently available and highly optimized ILOG CPLEX LP solver.

The second choice of exact vs. approximate is almost invariably determined by the size of the state space. If the state space is relatively small then one can easily resort to exact methods. However, if the state space is sufficiently large, approximate solution techniques are the only viable option. But this last statement depends critically on *how one measures the size of the state space*.

Looking ahead to future chapters, we note that there is only so much computational advantage that can be gained by using the approximate solution techniques in place of the exact techniques covered in this chapter. That is, all exact and approximate solution techniques mentioned here must represent the value function and policy (if required) as vectors or functions over an explicitly enumerated state space. As it turns out, there are many representations well suited to decision-theoretic planning tasks that do not require explicit state enumeration in the problem representation or in the solution. As such, the use and exploitation of structured representations is complementary to the choice of exact vs. approximate solution method or dynamic programming vs. direct linear program optimization. That is, the exploitation of structure can help all of these methods scale far beyond what is possible with approaches that rely on explicit state enumeration.

Thus, the modeling and exploitation of decision-theoretic planning structure in the MDP framework will be the core focus of the remainder of this thesis.

Chapter 3

Factored MDPs

In the MDP representation of the previous chapter we expressed the reward, transition distribution, policy, and value function all in terms of an explicitly enumerated state space. However, this is neither the most natural nor the most compact representation one can choose, nor can it be easily exploited in solution methods.

Intuitively, we often think of states in terms of various state properties. That is, a state representation can be factored into a number of properties that we will call *state variables* where each of these state variables can take assignments from a set of possible values. For example, a state variable may be the location of an object and it may take assignments from a small set of locations (e.g., office, hallway, or cafeteria). If there are a number of objects, we may choose to represent the location of each object with a different state variable. In this case of multiple state variables, states can be considered to be a joint configuration of all state variables. As we will show in the first half of this chapter, it is not only natural to represent MDPs in this factored manner, but state variable factoring can also result in compact representations that can be exploited by solution methods to avoid explicit state enumeration.

In the second half of this chapter, we will review a number of methods for exploiting factored MDP structure in extensions of solution algorithms from the previous chapter. We will also introduce the first contribution of this thesis, which is a compact data structure termed the affine algebraic decision diagram (AADD) that can compactly and simultaneously exploit multiple forms of independence in the representation and solution of factored MDPs.

3.1 Factored MDP Representations

While the MDP solution techniques from the previous chapter all require time at least polynomial in $|\mathcal{S}|$ and $|\mathcal{A}|$, we note that $|\mathcal{S}|$ can be very large. To see this, recall the SYSADMIN problem from Chapter 1 where the state can be represented by n binary state variables $x_1, \ldots, x_n$ where each state variable $x_i \in \mathcal{X}_i$ (with $\mathcal{X}_i = \{true, false\}$) represents whether computer i is running or not. In this problem, the total number of states is 2^n (i.e., $|\mathcal{S}| = |\{\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n\}|$). This is Bellman's well-known curse of dimensionality [Bellman, 1957] and it unfortunately implies that the enumerated state solution methods discussed in the last chapter require time exponential in the number of state variables n. Obviously, such enumerated state solution approaches would be computationally prohibitive for as few as 50 computers.[1]

Consequently, efficient representations and algorithms are extremely important for the solution of MDPs for realistic problems. This is especially true for fields such as decision-theoretic planning where 50 binary state variables would be considered at most an intermediate-sized problem.[2] In the following sections, we describe structured representations and algorithms that mitigate the problems associated with enumerated state MDP representations and solution algorithms, thus vastly increasing the size of MDPs that can be practically solved exactly or approximately.

3.1.1 Factored Transition and Reward Dynamics

One of the major representational bottlenecks in MDPs stems from representing the transition matrices. For example, with a state in SYSADMIN formed from $n = 3$ binary variables, the joint transition distribution would be of the form $P(x'_1, x'_2, x'_3, x_1, x_2, x_3, a)$ (with the x'_i variables representing the next-state variables and $a \in \mathcal{A}$ representing the three actions to reboot each of the three computers). If this probability distribution was represented in an enumerated manner, it would require $|\mathcal{A}| = 3$ matrices of row and column dimension 2^3 for a total of 2^6 entries per matrix. Clearly, it would become prohibitively difficult to store these matrices as more variables were added as the number of matrix entries scales exponentially

[1] For reference, using one byte of storage per enumerated state in an MDP with 50 variables would require one petabyte of storage, far beyond what could reasonably be stored in primary or secondary storage on a modern desktop computer.

[2] For example, in the 2006 ICAPS International Probabilistic Planning Competition, the largest problems in the ELEVATOR domain had well over 350 binary state variables if a binary variable were instantiated for each ground relational fluent. This amounts to over 2^{350} distinct states.

with the number of state variables n.

However, from an intuitive standpoint, most actions affect only a small subset of state variables, which can be exploited in a factored representation of the transition distribution. A *dynamic Bayes net (DBN)* [Dean and Kanazawa, 1989] serves as an appropriate representation in this case. Using a DBN, we can specify the effects of an action on an individual computer conditioned on the relevant state variables. Let us assume that our three computers in SYSADMIN are connected in a unidirectional ring[3], thus having the network configuration and DBN transition function representation in Figure 3.1(a). We can then specify the *conditional probability tables (CPTs)* in the DBN where the next state of each computer x_i' is conditioned on the computer's previous state x_i, the computer x_{i-1} to which it has an upstream connection, and the action (specifically whether x_i was rebooted by the action $reboot(i)$):[4]

$$P(x_i' = true|\vec{x}_i, a) = \begin{cases} a = reboot(i): & 1 \\ a \neq reboot(i) \wedge x_i = true: & .475 \cdot (\mathbb{I}[x_{i-1}] + 1) \\ a \neq reboot(i) \wedge x_i = false: & .025 \cdot (\mathbb{I}[x_{i-1}] + 1) \end{cases} \tag{3.1}$$

In words, this states that a computer is running in the next state with probability 1 if it was rebooted, or otherwise with a probability that is most impacted by the computer's previous state and somewhat less by the previous state of its upstream connection. The exact numerical values chosen here are taken from the SYSADMIN specification in *Guestrin et al.* [2002].

We can use a factored representation in the spirit of influence diagram [Howard and Matheson, 1984] representations to model the state variables that influence the reward function. This is also shown in Figure 3.1(a).

For this DBN, we can then write the full conditional joint transition distribution in the following factored form:

$$P(x_1', x_2', x_3'|x_1, x_2, x_3, a) \;\; = \;\; P(x_1'|x_1, x_3, a) \cdot P(x_2'|x_1, x_2, a) \cdot P(x_3'|x_2, x_3, a)$$

We note that the full conditional joint distribution for a single action would take 192 entries to represent as a fully enumerated CPT while the factored representation requires tables with a total number of 72 entries given the conditional independence assumptions. As the number

[3]Formally, in a unidirectional ring, each computer x_i has one incoming connection from x_{i-1} where subtraction is modulo n.

[4]The notation $\mathbb{I}[\cdot]$ is an indicator function that takes the value 1 when its argument evaluates to true and 0 when it evaluates to false.

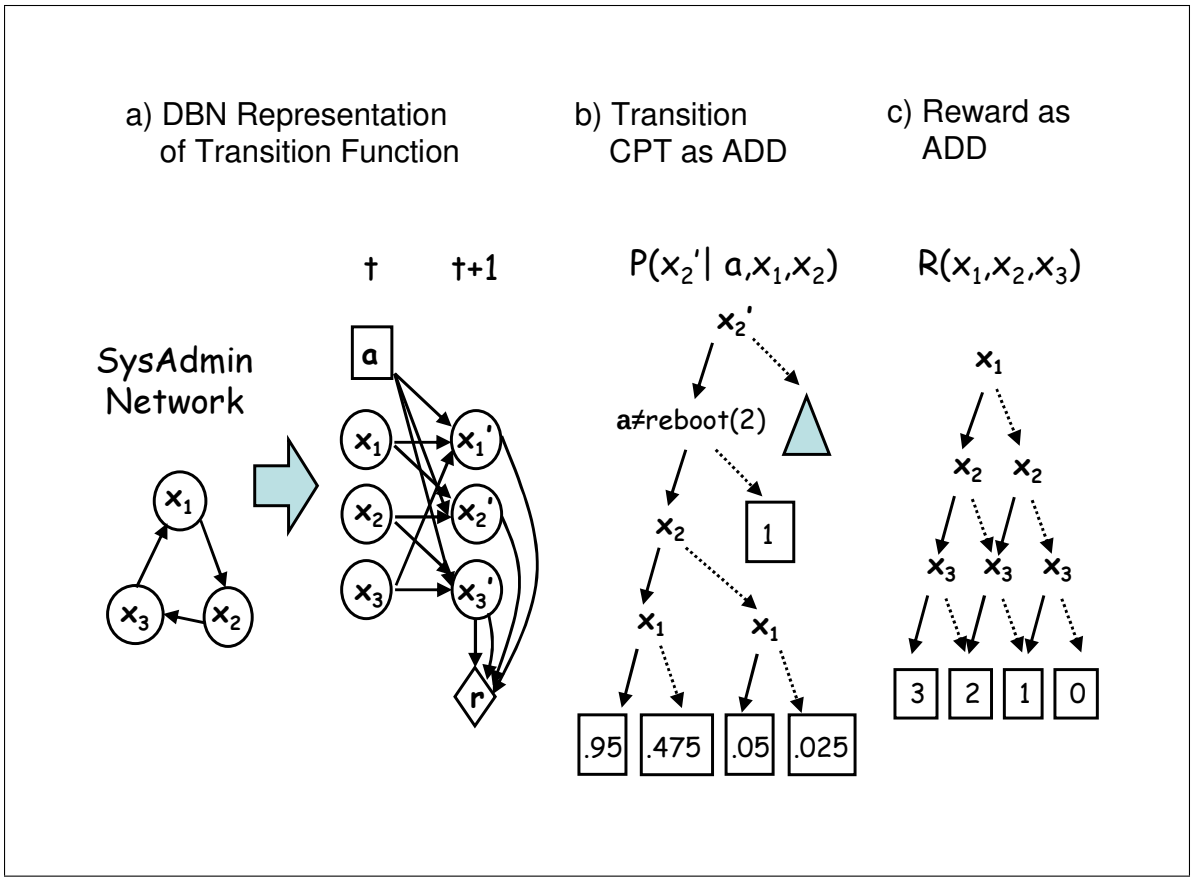


Figure 3.1: a) A dynamic Bayes network and decision diagram representing a transition function and a reward function for SYSADMIN with $n = 3$ and a unidirectional ring network topology. b) An compact encoding of the transition function CPT for the DBN as an ADD. Note that x_3' sums to one over all possible previous states. c) An ADD representation of the additive reward function for SYSADMIN. For all ADDs, the high (true) edge is solid, the low (false) edge is dotted.

of computers n in this unidirectional network topology increases, the size of the full joint representation will scale exponentially in n while the size of the DBN representation will scale only quadratically in n (requiring n CPTs each with $8n$ entries).

Throughout this exposition, we assume that the DBN representation of the transition function does not have synchronic arcs that specify dependences between post-action variables. However, if needed, it is easy to modify our DBN notation to permit such arcs and the forthcoming algorithms to take such arcs into account during inference. Or alternately, one may choose to modify the problem description to use joint variables in place of variables connected via synchronic arcs. This approach incurs a representational blowup exponential in the number of variables joined, but converts a DBN with synchronic arcs to an equivalent (but larger) DBN

without synchronic arcs.

We also note that there are alternative representations to the DBN transition representation such as probabilistic generalizations of STRIPS operators [Boutilier *et al.*, 1995a]. However, Littman [1997] proved that this representation can be converted to a dynamic Bayes net representation with only a polynomial blowup in size. This effectively demonstrates that both formalisms are representationally equivalent.

In the general case, using DBN and influence diagram structures to efficiently represent transition and reward dependencies often saves a considerable amount of space in these representations. Defining the *parents* of a next-state variable x_i' in the DBN representation as the set of current-state variables $\{x_j\}$ appearing in a CPT with x_i', we note that in the worst case, every x_i' has all $\{x_1, \ldots, x_n\}$ as parents, thus requiring a number of parameters exponential in n. In the best case, every state variable x_i' has only x_i as a parent, requiring a number of parameters linear in n. However, even in the typical case, if the number of parents of any state variable is bounded by some constant $k < n$, this requires $O(n \cdot 2^k)$ parameters in the case of binary state variables — still an exponential reduction over the worst case. While a factored transition and reward representation can yield substantial savings for the MDP representation, we note that this factoring cannot often be preserved in the value function due to the correlation of action effects over sufficiently extended periods of time [Boutilier *et al.*, 1995b]. Nevertheless, representing large MDPs is a first step toward solving them and subsequent techniques will take advantage of this factored structure for efficient computation and approximation.

3.1.2 Context-specific Independence and ADDs

Even if we can represent the joint transition probability as a Bayes net with a conditional probability table (CPT) for each next-state variable, we can often represent these tables more efficiently than by enumerating all state configurations of the variables in that table. Quite often, we find that certain values of variables in a CPT render the other values irrelevant. This is known as *context-specific independence (CSI)* [Boutilier *et al.*, 1996].

For the example DBN in Figure 3.1(a), given that the value of x_2' depends on x_1,x_2 and a in $P(x_2'|x_1, x_2, a)$ but that in the context of $a \neq reboot(2)$, the value of x_2' depends on no other variables, we say that in the context of $a \neq reboot(2)$, x_2' is independent of all other variables and thus $P(x_2'|x_1, x_2, a \neq reboot(2)) = P(x_2'|a \neq reboot(2))$. In order to represent this CSI compactly, we can use a decision tree or an algebraic decision diagram (ADD) [Bahar *et al.*, 1993], which is similar to a tree except that it is a canonical *directed acyclic graph (DAG)* with

all variable decision tests following a strict order from the root to the leaves. An example ADD for this probability distribution showing the above CSI is given in Figure 3.1(b). Effectively, CSI performs automatic state aggregation in that all possible state contexts under the condition $a \neq reboot(2)$ are effectively grouped together and assigned a common value. An example ADD for the reward is given in Figure 3.1(c), here there is no explicit CSI, but the reconvergent DAG structure of the ADD does allow sharing of common substructure that reduces what would be a tabular representation exponentially sized in n to an ADD representation quadratically sized in n.

In addition to the representational efficiency of state aggregation in ADDs, we note that computation with ADDs can also be very efficient. When we perform operations on factors represented as ADDs, we can just replace these operations with their ADD-based versions [Bahar *et al.*, 1993], allowing us to exploit CSI and shared substructure not only in the representation of factored MDPs, but also in the computations required for their solution.

Since the ADD will be a crucial data structure for our subsequent presentation of factored MDP solution algorithms, we provide a formal definition of ADDs and algorithms to construct and manipulate them in the following subsections. The following discussion draws on the work of Bahar *et al.* [1993], which is itself a slight variant of the original work on ordered *binary decision diagrams (BDDs)* of Bryant [1986].

Canonical Reduced ADDs

An ADD is a decision diagram with a fixed variable ordering of all decision tests on paths from the root to the leaves that is capable of representing functions from $\mathbb{B}^n \rightarrow \mathbb{R}$. We define ADDs with the following simple BNF grammar:

$$F ::= C \;\mid\; if\ (F^{var})\ then\ F_h\ else\ F_l \tag{3.2}$$

Here, $C \in \mathbb{R}$ is a constant-valued terminal node. Each internal decision node is represented as $if\ (F^{var})\ then\ F_h\ else\ F_l$ and is associated with a single variable var that indicates the high branch leading to node F_h should be taken when $var = true$ and the low branch leading to F_l should be taken when $var = false$.

Let $Val(F, \rho)$ be the value of ADD F under variable value assignment ρ. Then the valua-

tion of an ADD can be defined recursively by the following equation:

$$Val(F,\rho) = \begin{cases} F = C: & C \\ F \neq C \wedge \rho(F^{var}) = true: & Val(F_h,\rho) \\ F \neq C \wedge \rho(F^{var}) = false: & Val(F_l,\rho) \end{cases}$$

Formally, we define a *variable ordering* as a total ordering over all variables such that for all variable pairs x_i, x_j $(i \neq j)$ either $x_i \succ x_j$ or $x_j \succ x_i$. We say that F satisfies a given variable ordering if $F = C$ or F is of the form $if\ (F^{var})\ \ then\ F_h\ \ else\ F_l$ where (1) F^{var} does not occur in F_h or F_l, (2) F^{var} is the earliest variable under the given ordering occuring in F and (3) F_l and F_h satisfy the variable ordering. We discuss choices for variable order later in the context of variable reordering.

Then we obtain the following lemma where we define a *reduced* ADD to be the minimally-sized ordered decision diagram representation a function $f(x_1, \ldots, x_n)$.

Lemma 3.1.1. *Fix a variable ordering over $x_1, \ldots, x_n$. For any function $f(x_1, \ldots, x_n)$ mapping $\mathbb{B}^n \longrightarrow \mathbb{R}$, there exists a unique reduced ADD F over variable domain $x_1, \ldots, x_n$ satisfying the given variable ordering such that for all $\rho \in \mathbb{B}^n$ we have $f(\rho) = Val(F,\rho)$.*

Bryant [1986] provides a proof of this lemma for BDDs, which only have two distinct terminal values. The proof trivially generalizes to ADDs, which can have more than two distinct terminal values. This lemma shows that there is a unique canonical ADD representation of all functions from $\mathbb{B}^n \longrightarrow \mathbb{R}$.

Given that there exists a unique reduced ADD for any function from $\mathbb{B}^n \longrightarrow \mathbb{R}$, we next describe how this reduced ADD can be constructed from an arbitrary ordered decision diagram. All algorithms that we will define rely on the helper function $GetNode$ in Algorithm 1, which returns a canonical representation of a single internal decision node. Using $GetNode$, the $Reduce$ procedure in Algorithm 2 takes any ordered decision diagram and returns its reduced, canonical ADD representation (necessarily removing any redundant structure in the process). The control flow of $Reduce$ is very simple in that it uses the $GetNode$ procedure to recursively build a reduced ADD from the bottom up (i.e., from the terminal leaf nodes all the way up to the root node). An example application of the $Reduce$ algorithm is given in Figure 3.9.

Binary Operations on ADDs

Given functions $\mathbb{B}^n \longrightarrow \mathbb{R}$ represented as ADDs, we now want to apply operations to these functions that work directly on the ADD representation. Additionally, we would prefer that

Algorithm 1: $GetNode(v, F_h, F_l\rangle) \longrightarrow F_r$

```
input  : v, F_h, F_l : Var and node ids for high/low branches
output : F_r : Return values for offset,
         multiplier, and canonical node id
begin
    // If branches redundant, return child
    if (F_l = F_h) then
        return F_l;

    // Make new node if not in cache
    if (⟨v, F_h, F_l → id is not in node cache) then
        id := currently unallocated id;
        insert ⟨v, F_h, F_l⟩ → id in cache;

    // Return the cached, canonical node
    return id ;
end
```

Algorithm 2: $Reduce(F) \longrightarrow F_r$

```
input  : F : Node id
output : F_r : Canonical node id for reduced ADD
begin
    // Check for terminal node
    if (F is terminal node) then
        return canonical terminal node for value of F;

    // Check reduce cache
    if ( F → F_r is not in reduce cache) then
        // Not in cache, so recurse
        F_h := Reduce(F_h);
        F_l := Reduce(F_l);

        // Retrieve canonical form
        F_r := GetNode(F^var, F_h, F_l);

        // Put in cache
        insert F → F_r in reduce cache;

    // Return canonical reduced node
    return F_r;
end
```

these operations avoid enumerating all possible variable assignments whenever possible.

To do this, we first define the $Apply$ function that applies a binary operation to two operands represented as ADDs and returns the result as an ADD. We let op denote a binary operator on ADDs with possible operations being addition, substraction, multiplication, division, min, and max denoted respectively as $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\min(\cdot,\cdot)$, and $\max(\cdot,\cdot)$. We also define binary

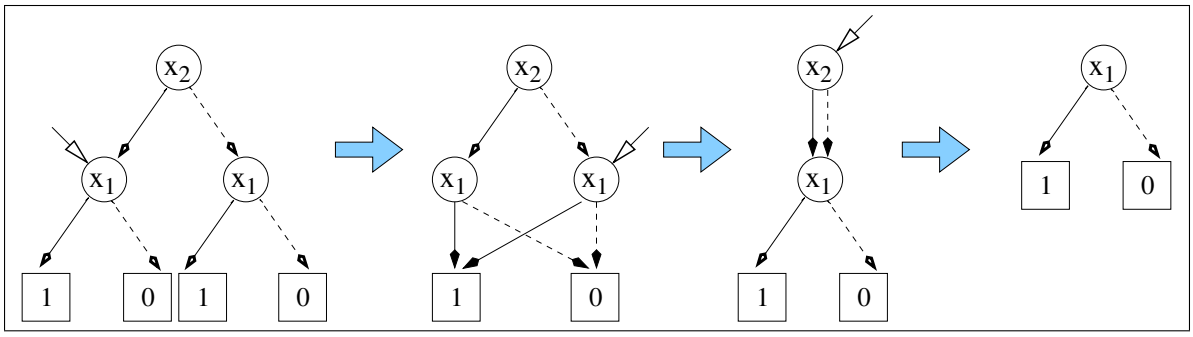


Figure 3.2: An example application of the $Reduce$ algorithm. The input is the leftmost diagram. From left to right, the hollow arrow shows the node F currently being evaluated by $Reduce$ just *after* the recursive $Reduce$ calls to the high branch F_h and low branch F_l but *before* $GetNode(F^{var}, F_h, F_l)$ is called and the canonical representation of F is returned (see Algorithm 2). The next diagram in the sequence shows the result after the previous $Reduce$ call. The rightmost diagram is the final canonical ADD representation of the input.

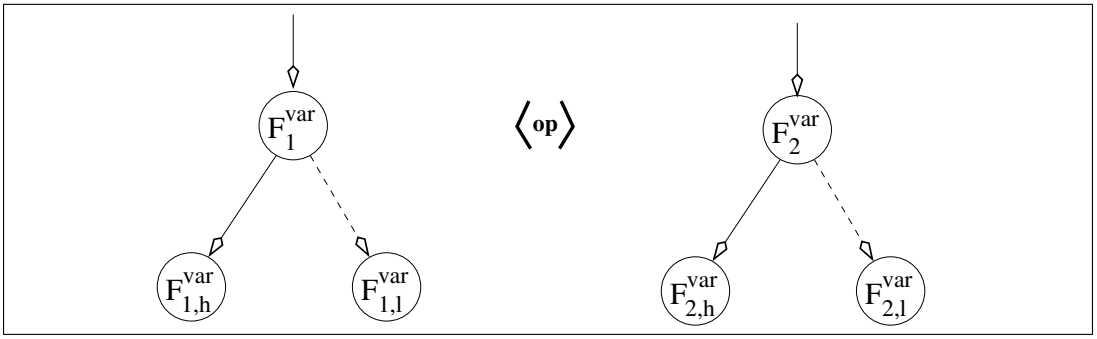


Figure 3.3: Two ADD nodes F_1 and F_2 and a binary operation op with the corresponding notation used in the presentation of the $Apply$ function.

comparison functions $\geq, >, \leq, <$ that return an indicator function represented as an ADD that takes the value 1 when the comparison is satisfied and 0 otherwise.

The high-level control flow of the $Apply$ routine in Algorithm 3 is straightforward: we first check whether we can compute the result immediately by calling $ComputeResult$, otherwise we check if we can reuse the result of a previously cached $Apply$ computation. If we can do neither of these, we then choose a variable to branch on and recursively call the $Apply$ routine for each instantiation of the variable. We cover these steps in-depth in the following sections and note that Figure 3.4 provides an example of the $Apply$ operation.

Terminal computation The function $ComputeResult$ given in Table 3.1, determines if the result of a computation can be immediately computed without recursion. The first entry in

Algorithm 3: $Apply(F_1, F_2, op) \longrightarrow F_r$

```
input  : F_1, F_2, op : ADD nodes and op
output : F_r : ADD result node to return
begin
    // Check if result can be immediately computed
    if (ComputeResult(F_1, F_2, op) → F_r is not null ) then
        return F_r;
    // Check if result already in apply cache
    if ( ⟨F_1, F_2, op⟩ → F_r is not in apply cache) then
        // Not terminal, so recurse
        if (F_1 is a non-terminal node) then
            if (F_2 is a non-terminal node) then
                if (F_1^var comes before F_2^var ) then
                    var := F_1^var;
                else
                    var := F_2^var;
            else
                var := F_1^var;
        else
            var := F_2^var;
        // Set up nodes for recursion
        if (F_1 is non-terminal ∧ var = F_1^var ) then
            F_l^v1 := F_1,l;  F_h^v1 := F_1,h;
        else
            F_l/h^v1 := F_1;
        if (F_2 is non-terminal ∧ var = F_2^var ) then
            F_l^v2 := F_2,l;  F_h^v2 := F_2,h;
        else
            F_l/h^v2 := F_2;
        // Recurse and get cached result
        F_l := Apply(F_l^v1, F_l^v2, op);
        F_h := Apply(F_h^v1, F_h^v2, op);
        F_r := GetNode(var, F_h, F_l);
        // Put result in apply cache and return
        insert ⟨F_1, F_2, op⟩ → F_r into apply cache;
    return F_r;
end
```

this table is required for proper termination of the algorithm as it computes the result of an operation applied to two terminal constant nodes. However, the other entries denote a number of pruning optimizations that immediately return a node without recursion. For example, we know that $F_1 \oplus 0 = F_1$ and $F_1 \otimes 1 = F_1$. If a result cannot be immediately determined in $ComputeResult$ then we must continue recursing on the substructure of the operands until a

$ComputeResult(F_1, F_2, op) \longrightarrow F_r$	
Operation and Conditions	**Return Value**
F_1 *op* F_2; $F_1 = C_1$; $F_2 = C_2$	C_1 *op* C_2
$F_1 \oplus F_2$; $F_2 = 0$	F_1
$F_1 \oplus F_2$; $F_1 = 0$	F_2
$F_1 \ominus F_2$; $F_2 = 0$	F_1
$F_1 \otimes F_2$; $F_2 = 1$	F_1
$F_1 \otimes F_2$; $F_1 = 1$	F_2
$F_1 \oslash F_2$; $F_2 = 1$	F_1
$\min(F_1, F_2)$; $\max(F_1) \leq \min(F_2)$	F_1
$\min(F_1, F_2)$; $\max(F_2) \leq \min(F_1)$	F_2
similarly for $\max$	
$F_1 \leq F_2$; $\max(F_1) \leq \min(F_2)$	1
$F_1 \leq F_2$; $\max(F_2) \leq \min(F_1)$	0
similarly for $<, \geq, >$	
other	$null$

Table 3.1: Input and output summaries of $ComputeResult$. If $ComputeResult$ receives two constant ADD nodes as input, the constant resulting from the direct evaluation of *any* possible binary operation is returned. In other cases where at least one node is non-terminal, special operand structure and specific operator properties sometimes permit the computation of the result without further recursion. Some computations rely on the unary $\min(F)$ and $\max(F)$ operators that are discussed directly following the $Apply$ algorithm.

result can be computed.

Recursive computation If a call to $Apply$ is unable to immediately compute a result or reuse a previously cached computation, we must recursively compute the result. For this we have two cases (the third case where both operands are constant terminal nodes having been taken care of in the previous section). These algorithms assume the notation given in Figure 3.3 for the structure of the operands.

- F_1 **or** F_2 **is a constant terminal node, or** $F_1^{var} \neq F_2^{var}$**:** For simplicity of exposition, we assume the operation is commutative and reorder the operands so that F_1 is the constant node or the operand whose variable comes *later* in the variable ordering so that we know to branch on F_2^{var} first.[5] Thus, we compute the operation applied separately to

[5]We note that the first case prohibits the use of the non-commutative $\ominus$ and $\oslash$ operations. However, a simple solution would be to recursively descend on either F_1 or F_2 rather than assuming commutativity and swapping operands to ensure descent on F_2. To accommodate general non-commutative operations, we have used this alternate approach in our specification of the $Apply$ routine.

F_1 and *each* of F_2's high and low branches. We then build an internal if decision node conditional on F_2^{var} and get its canonical representation for the result:

$$F_h = Apply(F_1, F_{2,h}, op)$$
$$F_l = Apply(F_1, F_{2,l}, op)$$
$$F_r = GetNode(F_2^{var}, F_h, F_l)$$

- F_1 **and** F_2 **are constant nodes and** $F_1^{var} = F_2^{var}$**:** Since the variables for each operand match, we know the result F_r is simply an if statement branching on F_1^{var} (= F_2^{var}) with the true case being the operator applied to the high branches of F_1 and F_2 and likewise for the false case and the low branches:

$$F_h = Apply(F_{1,h}, F_{2,h}, op)$$
$$F_l = Apply(F_{1,l}, F_{2,l}, op)$$
$$F_r = GetNode(F_1^{var}, F_h, F_l)$$

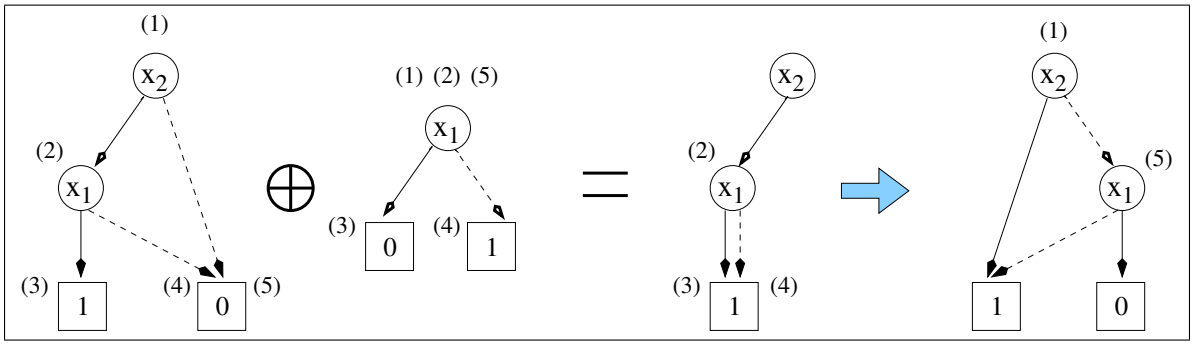


Figure 3.4: An example application of the $Apply$ algorithm. The indices (i) in the diagram correspond to successive (recursive) calls to the $Apply$ algorithm: for the operands the indices denote which node of each operand is passed as a parameter to the call to $Apply$ (the op is always $\oplus$); for the result the indices indicate the node that is returned by the call to $Apply$. For example, the initial call to $Apply$ takes the arguments corresponding to the node marked (1) x_2 on the LHS of the $\oplus$ and the node (1) x_1 on the RHS of the $\oplus$ (as well as the operation $\oplus$ itself) and returns the node marked (1) on the RHS of the equality.

Other Operations Above we covered binary operations on ADDs, but we will also need to perform a variety of unary operations on ADDs such as determining the $\min$ and $\max$ value of

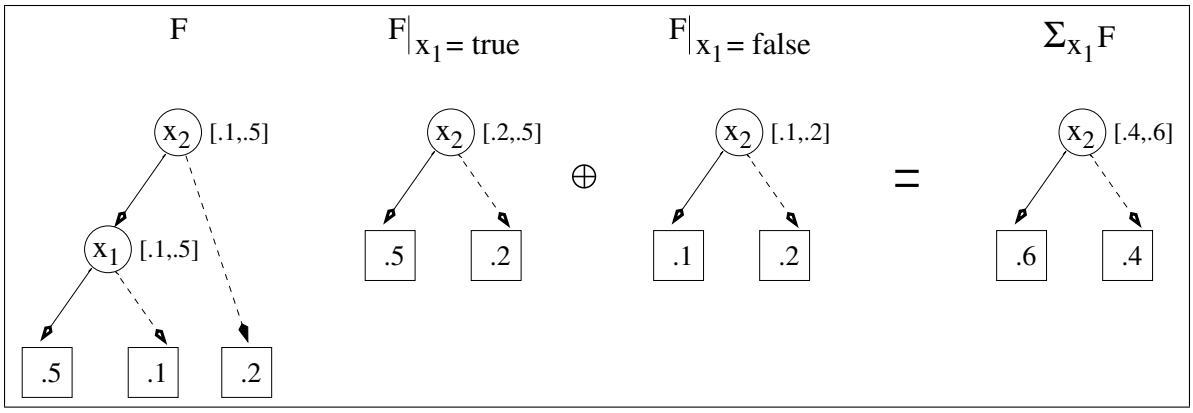


Figure 3.5: An example application of the unary *restriction* and *marginization* operations. Each ADD has all of its internal nodes annotated with $[\min, \max]$, which can be recursively computed from the children of each internal node.

an ADD and marginalization over variables. Here we cover some unary operations that can be performed (efficiently) on ADDs:

- **min and max computation:** During the $Reduce$ operation, it is easy to maintain the minimum and maximum values for each internal decision node. Exploiting the fact that an ADD is a DAG, $\min F = \min(F_l, F_h)$ and likewise for $\max$. A simple example of this annotation and its recursive relationship is shown in Figure 3.5.

- **Restriction:** The restriction of a variable x_i in an ADD F to either *true* or *false* (i.e. $F|_{x_i=true/false}$) can be computed by replacing all decision nodes for variable x_i with the branch corresponding to the variable restriction. Then $Reduce$ can be applied on the resulting decision diagram to convert it to a canonical ADD. Two examples of restriction are given in Figure 3.5.

- **Sum out/marginalization:** A variable x_i can be summed (or marginalized) out of a function F simply by computing the sum of the restricted functions (i.e. $\sum_{x_i} F = F|_{x_i=T} \oplus F|_{x_i=F}$). An example of this is given in Figure 3.5.

- **Negation/reciprocation:** Negation can be performed using the binary $Apply$ operation on $0 \ominus F$. Likewise, reciprocation (i.e., $\frac{1}{F}$) can be computed using the binary $Apply$ operation $1 \oslash F$.

- **Variable reordering:** Rudell [1993] provides an ADD variable reordering algorithm that casts a general variable reordering in terms of a sequence of pairwise reorderings

of neighboring variables. Then, the basic idea is that two variables x_i and x_j can be reordered locally (i.e., rotated) in the ADD DAG without requiring the modification of any internal nodes other than those involving x_i and x_j. Furthermore, Rudell describes how this can be done without requiring extra storage for backpointers from children to parents if $GetNode$'s canonical node cache is allowed to be modified.

As an addendum to this final operation, we note that the MDP solution algorithms based on ADDs (and their extensions) that we introduce in this chapter could dynamically reorder variables in an attempt to maintain even more compact representations than possible with a fixed variable ordering. However, we do not employ such dynamic variable ordering techniques in this thesis as they prevent the reuse of cached computations that underly one of the major sources of efficiency of ADDs when used in MDP solution algorithms. Furthermore, searching for compact ADD representations requires search and is computationally expensive. Such results are reflected in experiments using ADDs to perform value iteration in factored MDPs [St-Aubin *et al.*, 2000], which demonstrate that dynamic variable reordering does not pay off and that a natural fixed variable ordering derived from the MDP description tends to be compact and preserves structure. As a consequence of these observations, all of the algorithms used in this thesis use a natural fixed variable ordering derived from the order that variables appear in an MDP problem description, unless otherwise noted.

3.1.3 Additive Independence

Additive independence in reward structure is a common assumption in utility theory and related fields [Keeney and Raiffa, 1976; Bacchus and Grove, 1995]. In Figure 3.1(c), we note that we could represent the additive reward structure of SYSADMIN using an ADD whose size scales quadratically in the number of computers n. But if we can explicitly model additive rewards as sums of (potentially non-linear) factors, then we trivially note that the SYSADMIN reward can be expressed compactly in a form whose size scales linearly in n:

$$R(\vec{x}, a) = \sum_{i=1}^{n} \mathbb{I}[x_i] \tag{3.3}$$

Furthermore, if we permit the use of similar expressions in the CPTs that we specify for our transition DBN, we can also exploit additive independence in their representation. For example, letting $Conn(i, j)$ denote that there is an incoming network connection to computer j from computer i, we note that the CPTs for the transition function for any SYSADMIN network

topology can be specified in the following additive manner:

$$P(x_i' = true|\vec{x}_i, a) = \begin{cases} a = reboot(c_i): & 1 \\ a \neq reboot(c_i): & (0.05 + 0.9 \cdot \mathbb{I}[x_i]) \cdot \frac{\sum_j \mathbb{I}[j \neq i \wedge x_j \wedge Conn(j,i)]}{|\{x_j | j \neq i \wedge Conn(j,i)\}| + 1} \end{cases}$$

Here we see that the success probability of a computer running scales proportionally to the number of it's incoming connections that are also running. And we also note that the previous CPT we gave for the unidirectional ring in Figure 3.1(b) is just a special case of this CPT where computer i is connected only to computer $i + 1$ (where addition is modulo n).

In our subsequent discussion of solution methods for factored MDPs, we note that some recent approaches can exploit additive reward structure while others cannot. In fact, it will only be in the final part of this chapter when we introduce affine ADDs (AADDs) that we will be able to fully exploit CSI and additive independence in both the reward and transition functions, not to mention multiplicative independence as it happens to naturally occur in many value functions.[6]

3.1.4 Structured Policy Representation

Just as the reward and transition function may be represented in a factored manner in propositional MDPs, so can the policy. To do this, we adapt the following definition from Boutilier *et al.* [1995b]:

Definition 3.1.2. *A structured policy is any set of function-action pairs* $\pi = \{\langle \phi_a, a \rangle\}$ *such that* ϕ_a *is a structured representation of an indicator function and* $\{\phi_a\}$ *partitions the state space. This induces the explicit policy* $\pi_a(\vec{x}) = a$ *iff* $\phi_a(\vec{x}) = 1$.

To ensure that the policy partitions the state space, one must ensure that it is exhaustive and that all action indicator functions are pairwise disjoint. To ensure that the policy exhausts the entire state space, one can simply ensure that the sum of all indicator functions is the constant 1 (i.e.,$\sum_{a \in \mathcal{A}} \phi_a = 1$). To ensure that all action policies are pairwise disjoint, one can ensure $\phi_a \cdot \phi_b = 0$ for all action pairs $a \in \mathcal{A}$, $b \in \mathcal{A}$ such that $a \neq b$.

There are a variety of structured methods for representing the $\{\phi_a\}$ indicator functions ranging from decision lists [Koller and Parr, 1999a], to trees [Boutilier *et al.*, 1995b], to ADDs. Throughout our presentation here, we will use ADDs. Then policy evaluation is simply the task

[6] Multiplicative independence is just the multiplicative generalization of additive independence.

of evaluating each ADD ϕ_a under a given state assignment $\vec{x}$ to see if $\phi_a(\vec{x}) = 1$ (meaning do action a). This structured policy representation will play an important role in our description of structured policy iteration.

3.1.5 Putting it all Together

Before we cover exact solution methods in the factored MDP framework, let us quickly recapitulate the factored MDP representation. In a factored MDP, states will be represented by vectors $\vec{x}$ of length n, where for simplicity we assume all state variables $x_1, \ldots, x_n$ are binary-valued;[7] hence the total number of states is $N = 2^n$. We also assume a finite set of actions $A = \{a_1, \ldots, a_m\}$. As usual, we assume a discount factor $\gamma,\ 0 \leq \gamma \leq 1$ where appropriate steps have been taken to ensure bounded reward in the case of $\gamma = 1$.

To generalize the MDP model from the previous chapter, we specify a propositionally factored MDP by the following:

1. *Factored Transition Function:* A DBN-factored state transition model which specifies the probability of the next state $\vec{x}'$ given the current state $\vec{x}$ and action a. The transition function can be factored as a dynamic Bayes net (DBN) with CPTs $P(x_i'|\vec{x}_i, a)$ where each next state variable x_i' is only dependent upon the action a and its direct parents $\vec{x}_i$ in the DBN. Then the transition model can be compactly specified as $P(\vec{x}'|\vec{x}, a) = \prod_{i=1}^{n} P(x_i'|\vec{x}_i, a)$.

2. *Factored Reward Function:* An additive reward function $\sum_{i=1}^{r} R_i(\vec{x}_i, a)$ over r reward factors $R_i(\vec{x}_i, a)$ dependent on action a and relevant state $\vec{x}_i$, which specifies the immediate reward obtained by taking action a in state $\vec{x}$.

The individual factors can be expressed as tabular representations, or as trees and ADDs that exploit CSI, or even as additive expressions that exploit additive independence. Finally, when needed, a structured policy $\pi = \{\langle \phi_a, a \rangle\}$ uses indicator functions ϕ_a to specify the states where action a should be taken.

[7] However, all of the methods here can be easily generalized to non-binary variables through known transformations [Rossi *et al.*, 1990; Stergiou and Walsh, 1999].

3.2 Exact Solution Methods

In our specification of our solution methods, it will be notationally useful to define a *backup* operator B^a for action a as follows:[8]

$$B^a[V(\vec{x})] = \gamma \sum_{\vec{x}'} \prod_{i=1}^{n} P(x_i'|\vec{x}_i, a) V(\vec{x}') \tag{3.4}$$

This is essentially the factored representation of the Q-function computation for action a in Equation 2.8 from Chapter 2 without adding in the reward. We note that the backup $B^a[\cdot]$ operator can exploit both additive structure since it is a *linear operator* as well as efficient factored computation due to the transition DBN structure.

If π^* denotes the optimal policy and V^* its value function, then we have the following factored representation of the fixed-point Equation 2.4 from Chapter 2:

$$V^*(\vec{x}) = \max_{a \in \mathcal{A}} \left\{ \sum_{i=1}^{r} R_i(\vec{x}_i, a) + B^a[V^*(\vec{x})] \right\}. \tag{3.5}$$

Having done this, we first present the basic factored variants of the relevant MDP equations from the previous thesis chapter and proceed to show that the previous MDP solution methods can be easily redefined in terms of these factored equations. This allows us to exploit the factored structure and any CSI therein during the application of the MDP solution algorithms.

3.2.1 Structured Value Iteration

ADD-based Value Iteration

The value iteration algorithm from Chapter 2 can be easily extended to exploit factored MDP structure in a structured value iteration setting. Initializing $V^0(\vec{x})$ to some value, we generalize Equation 2.7 from Chapter 2 to the factored form:

$$V^{t+1}(\vec{x}) = \max_{a \in \mathcal{A}} \left\{ \sum_{i=1}^{r} R_i(\vec{x}_i, a) + B^a[V^t(\vec{x})] \right\}. \tag{3.6}$$

It will be extremely important to use a compact data structure such as a tree or ADD to exploit CSI in the representation of the value function in structured value iteration. If we were

[8] Technically, this should be written $(B^a V)(\vec{x})$, but we abuse notation for readability when V itself is structured and for consistency with subsequent first-order MDP notation.

to simply use a tabular representation, we would find that in typical MDPs, all variables in the value function become correlated after some number of backups *if* the graphical model underlying the DBN cannot be decomposed into disjoint components [Boutilier *et al.*, 1995b]. Thus, a tabular representation will typically need to represent a value function over all state variables and in the absence of some method for compactly representing value function structure, this representation will require full state enumeration.

Fortunately, as described previously, ADDs are ideal for exploiting CSI and functions with shared substructure, both of which may occur in the value functions of highly structured factored MDPs. As such, representing all factors in Equation 3.6 using ADDs and carrying out its computation in terms of ADD operations as done in the SPUDD algorithm of Hoey *et al.* [1999] has proved to be a promising method in comparison to the enumerated state value iteration approach of the previous chapter. While SPUDD may scale comparably to enumerated state value iteration in the worst case (e.g., when all states have distinct values), the authors demonstrate that there is much potential for computational and space savings using the SPUDD algorithm to perform value iteration on many factored MDPs.

Decomposition-based Value Iteration

In a different vein of research, there are alternate (but not incompatible) approaches to structured value iteration that exploit decomposable task structure in MDPs [Meuleau *et al.*, 1998a; Singh and Cohn, 1998]. If a problem domain consists of many independent subprocesses that only interact via their dependence on globally shared resources and/or constraints on joint action choices, one can often factor these MDPs into tasks represented as independent subMDPs with global resource and action constraints. We could take the cross-product of all the subMDP state spaces and solve the resulting joint MDP, but this would discard a lot of the structure inherent in the task decomposition of the initial problem. Alternately we can focus on algorithms that directly exploit the decomposed structure of the MDP directly.

An exact structured value iteration approach for a subclass of MDPs with highly decomposable structure is provided by Singh and Cohn [Singh and Cohn, 1998]. In this model, an MDP must decompose into a set of subMDPs where each subMDP has its own independent state space but an action set that is globally constrained. The reward objective is to maximize the sum of rewards for each subMDP. The solution approach they advocate is a value iteration method based on maintaining upper and lower bounds on the value function. The upper bounds simply come from assuming that actions are unconstrained across subMDPs (which

can be achieved in the best case) and the lower bounds come from taking the maximal reward for an individual subMDP (which could be achieved in the worst case). These upper and lower bounds allow various actions to be pruned from consideration during value iteration and with enough iterations will provably converge on an optimal solution. This decomposed value iteration algorithm is empirically found to be more efficient than value iteration in the joint cross-product MDP.

3.2.2 Structured Policy Iteration

Structured policy iteration (SPI) in factored MDPs was first defined in Boutilier *et al.* [1995b] using trees as a method of state aggregation. Here we describe a similar version using ADDs.[9] Recalling the definition of modified policy iteration from the previous chapter, first we initialize a random policy $\pi_0 = \{\langle \phi_a, a \rangle\}$ and then we iterate between approximate policy evaluation and policy improvement steps. For approximate policy evaluation, we can simply use the following factored extension of the successive approximation update (c.f. Section 2.4.2):

$$V_{\pi_i}^{t+1}(\vec{x}) = \sum_{a \in \mathcal{A}} \phi_a(\vec{x}) \cdot \left\{ \sum_{j=1}^{r} R_j(\vec{x}_j, a) + B^a[V_{\pi_i}^t(\vec{x})] \right\}. \tag{3.7}$$

Here, the policy indicator function ϕ_a ensures that the value for a state is only updated for action a if the policy indicates that action a should be taken from that state. We note that this entire computation can be carried out in terms of efficient operations on ADDs. Correctness follows from the fact that π_i is a partitioning of the state space.

Then, given $V^{\pi_i}(\vec{x})$, we need only produce a new policy π_{i+1} that is greedy w.r.t. V^{π_i}. In order to break ties for actions having equal value, we require a total preference ordering (perhaps random) over actions, that is, for all actions a and b such that $a \neq b$, either $a \succ b$ or $b \succ a$. Recalling the definition of the ADD "$>$" and "$\geq$" comparison functions that produce an ADD taking the value 1 in states where the LHS operand is greater than (or equal to) the RHS operand and 0 otherwise, we can produce the ADD representation of ϕ_a for all $a \in \mathcal{A}$ in the following iterative fashion:

1. Initialize $\phi_a = 1$ (the constant 1 ADD)
2. For each $a \in \mathcal{A}$, let $Q(\vec{x}, a) = \sum_{j=1}^{r} R_j(\vec{x}_j, a) + B^a[V_{\pi_i}(\vec{x})]$

[9] We will implicitly assume throughout the text that all operations in the following equations such as $+, -, \times$, *etc.* are performed on ADDs in terms of their corresponding operations $\oplus, \ominus, \otimes$, *etc.*

3. For each $b \in \mathcal{A}$ s.t. $b \neq a$ update ϕ_a as follows:

$$\phi_a := \begin{cases} a \succ b: & \phi_a \cdot (Q(\vec{x}, a) \geq Q(\vec{x}, b)) \\ b \succ a: & \phi_a \cdot (Q(\vec{x}, a) > Q(\vec{x}, b)) \end{cases}$$

Thus we have defined a structured version of policy iteration. While our algorithm presentation here differs from the original presentation [Boutilier *et al.*, 1995b], it is consistent with the overall structured approach to policy iteration. Furthermore, we will build on this approach when we extend policy iteration to exploit other types of structure in future chapters.

3.2.3 Difficulty of Structured Linear Programming

We do not present a structured variant of the exact linear programming solution to factored MDPs as this method requires *a priori* knowledge of the structure of the value function and in this case we are talking about the exact value function. Typically, we cannot determine the structure of an optimal value function from the structure of a factored MDP. Consequently, for the exact case, we would have no choice but to use a fully enumerated state representation of the value function, thus preventing the exploitation of factored structure. However, as we will see shortly, the approximate variant of linear programming is in fact quite useful for solving factored MDPs and there is much opportunity to exploit factored structure in that case.

3.3 Approximate Solution Methods

While some factored MDPs do exhibit considerable structure in their optimal value functions or policies, sometimes these representations are still too large for practical representation or computation as the size of the problem scales. Thus, in this section we focus on approximate variants of previously described solution algorithms.

3.3.1 Approximate Value Iteration Methods

ADD-based Approximation

One additional benefit of the use of ADDs to specify factored MDPs is that it allows one to prune internal nodes in an ADD and replace these nodes with the minimum and maximum value of the ADD rooted at that node [Dearden and Boutilier, 1997]. An example of this is

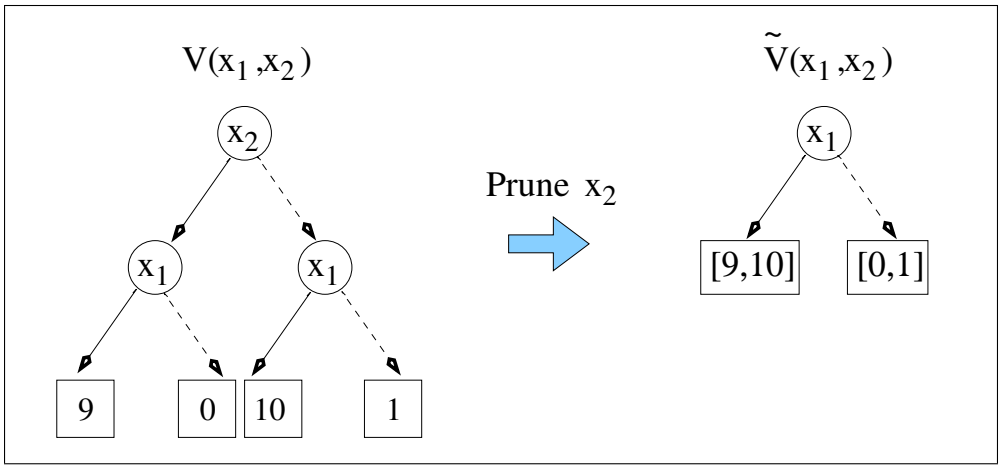


Figure 3.6: An example of approximating an ADD representation of a value function $V(x_1, x_2)$ as $\tilde{V}(x_1, x_2)$ by pruning out the decision node for variable x_2 and replacing leaf values with their respective ranges.

shown in Figure 3.6. One can then perform value iteration maintaining these upper and lower bounds. Since the Bellman backup is a known contraction operator [Puterman, 1994], this algorithm will still converge, albeit within some error bound of the optimal value function. This is the idea behind the APRICODD [St-Aubin *et al.*, 2000] algorithm that is essentially the SPUDD value iteration algorithm with an extra step for approximation by pruning the value function ADD. We note that APRICODD represents a completely automated approach to approximate value iteration that autonomously derives the approximated value function. It should be noted that this contrasts sharply with the linear-value approximation approach to approximate value iteration discussed in Section 2.5.3 that relies on the pre-specification of a fixed set of basis functions.

Decomposition-based Approximation

In the vein of exploiting structure in decomposable MDPs, Mealeau *et al.* [1998a] describe an approximate value iteration technique for solving weakly coupled subMDPs having global resource and action constraints. Their algorithm is referred to as Markov Task Decomposition (MTD) and is an approximately optimal approach to solving the joint MDP that divides the solution into local and global optimization steps. MTD first determines the optimal value function for each subMDP. Following this local optimization, a global optimization phase then chooses a joint action at each time step that enforces the global resource constraint while trading off local action choices for each task in order to maximize the expected reward. Since an

optimal sequential solution in this case would be equivalent to solving the full joint MDP, a heuristic resource allocator is used in this work.

While we don't exploit the same approximate decomposition ideas in the contributions of this thesis, we do note that the general framework of additive value decomposition and approximate solution methods within this framework serves as motivation for our work in future chapters.

3.3.2 Linear-value Approximation Solution Methods

We next introduce three efficient approximate solution methods for factored MDPs based on linear-value approximation [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001]. These methods are effectively factored extensions of the approximate policy iteration and approximate linear programming techniques from the previous chapter. The key to the efficiency of these approaches over their enumerated state counterparts will be to show how the structure of the factored representation can be exploited by algorithms such as *variable elimination* [Zhang and Poole, 1996] that scale exponentially in the induced tree-width of the representation rather than exponentially in the total number of state variables. This exploitation of structure will be most apparent when solving the linear programs for error-minimizing max-norm projections that are at the heart of these techniques.

In a linear-value function representation, we represent V as a linear combination of k basis functions $b_j(\vec{x})$ where the b_j are typically dependent upon small subsets of $\vec{x}$:

$$V(\vec{x}) = \sum_{j=1}^{k} w_j b_j(\vec{x}) \tag{3.8}$$

Our goal is to find weights that approximate the optimal value function as closely as possible. In doing this, all of our solution methods will need to compute the backup of the value function through an action a. To compute this, we recall that the backup operator $B^a[\cdot]$ previously defined is a linear operator such that it distributes into a sum:

$$B^a[V(\vec{x})] = B^a[\sum_{j=1}^{k} w_j b_j(\vec{x})] \tag{3.9}$$

$$= \sum_{j=1}^{k} w_j B^a[b_j(\vec{x})] \tag{3.10}$$

Thus, if the basis functions are defined over small sets of variables, and the backup introduces an additional small set of variables that causally affect this basis function according to the DBN representation of the transition distribution, this sum will be over factors of small sets of variables. This factored structure will be exploited in the methods we define in this section.

Next, we explore factored extensions of approximate policy iteration and linear programming. However, we do not cover approximate value iteration approaches due to their possibility of divergence as noted in the last chapter.

Approximate Policy Iteration

We can generalize policy iteration (API) to the factored linear-value approximation case by calculating successive iterations of weights $w_j^{(i)}$ that represent the best approximation of the fixed point value function for policy $\pi^{(i)}$ at iteration i. The method we present here is a slight variant of that given in Guestrin *et al.* [2002][10] and is an approach that we will generalize in the next chapter. To apply this approach, we need to introduce the $B^\pi[\cdot]$ operator which is just the backup operator under a fixed policy:

$$B^\pi[V(\vec{x})] = \gamma \sum_{\vec{x}'} \prod_{i=1}^{n} P(x_i'|\vec{x}_i, \pi(\vec{x}))V(\vec{x}') \tag{3.11}$$

In the context of the following algorithm, we will discuss how $B^\pi[\cdot]$ can be efficiently computed in structured cases.

We perform API by carrying out the following two steps on each iteration i: (1) derive the greedy policy: $\pi^{(i+1)} \leftarrow \pi_{gre}(\sum_{j=1}^{k} w_j^{(i)} b_j(s))$ using the approach outlined in Section 3.2.2 and (2) use the following LP to determine the weights for the L_∞ minimizing projection of the

[10]Other than the ordering of action comparisons in the greedy policy derivation method of Section 3.2.2, this presentations of API follows that of Guestrin *et al.* [2002]. For greedy policy derivation, they use a special ordering of action comparisons that first compares all actions to a *noop* action and then to each other, arguing that this approach is advantageous for domains such as SYSADMIN.

approximate value function for policy $\pi^{(i+1)}$:

$$\begin{aligned}
&\text{Variables: } w_1^{(i+1)}, \ldots, w_k^{(i+1)} \\
&\text{Minimize: } \beta^{(i+1)} \\
&\text{Subject to: } \beta^{(i+1)} \geq \left| \sum_{i=1}^{r} R_i(\vec{x}_i, \pi(\vec{x})) + \sum_{j=1}^{k} (w_j^{(i+1)} B^{\pi^{(i+1)}}[b_j(\vec{x})]) \right. \\
&\qquad\qquad \left. - \sum_{j=1}^{k} [w_j^{(i+1)} b_j(\vec{x})] \right| \; ; \; \forall \vec{x},
\end{aligned} \tag{3.12}$$

We note that this LP is just the factored form of the LP defined for approximate policy iteration in Equation 2.23 from the previous chapter where we have exploited linearity of the backup operator. Consequently, when the policy converges (i.e., $\pi^{(i+1)} = \pi^{(i)}$ or equivalently $\vec{w}^{(i+1)} = \vec{w}^{(i)}$), we can derive an error bound on the approximated value function by plugging the projection error β of the final LP solution directly into Equation 2.19 since β is the Bellman error of the approximated value function in this case.

However, we note that in the factored framework, $B^{\pi^{i+1}}[\cdot]$ cannot easily be computed according to Equation 3.11 since our structured policy $\pi(\vec{x})$ takes the form of indicator functions. However, we need only enforce that an LP constraint for an action a is satisfied in the states where ϕ_a^{i+1} takes the value 1. To do this, we can ensure that the constraint for action a is trivially satisfied when ϕ_a is 0. So we introduce the following policy factor as a summand in our constraint:

$$\hat{\phi}_a^{i+1} = (\phi_a^{i+1} - 1) \cdot \infty \tag{3.13}$$

Clearly, $\hat{\phi}_a^{i+1}$ will take the value 0 in states where a should be taken according to π^{i+1} and the value $-\infty$ otherwise.

To see how this allows us to perform the backup under a policy, let us rewrite the constraints we have expressed above:

$$\beta^{(i+1)} \geq \hat{\phi}_a^{i+1} + \left| \sum_{i=1}^{r} R_i(\vec{x}_i, a) + \sum_{j=1}^{k} (w_j^{(i+1)} B^a[b_j(\vec{x})]) - \sum_{j=1}^{k} [w_j^{(i+1)} b_j(\vec{x})] \right| \; ; \; \forall \vec{x}, a \in A \tag{3.14}$$

Effectively, the constraint for action a will be trivially satisfied when the policy factor $\hat{\phi}_a^{i+1}$ should not be applied and takes the value $-\infty$. Otherwise, $\hat{\phi}_a^{i+1}$ takes the value 0 in states where the policy should be applied and then the remainder of the constraint must be satisfied.

In a subsequent section, we discuss efficient methods for solving the above form of LP with

a factored $\max$-$\sum$ form of the constraints.

Approximate Linear Programming

In the extension of approximate linear programming (ALP) to factored models, we simply replace the enumerated state representation from the previous chapter in Equation 2.24 with the factored representation introduced in this chapter following Schuurmans and Patrascu [2001] where we have again exploited linearity of the backup operator:

$$\begin{aligned} &\text{Variables: } w_1, \ldots, w_k \\ &\text{Minimize: } \sum_{\vec{x}} \sum_{j=1}^{k} w_j b_j(\vec{x}) \\ &\text{Subject to: } 0 \geq \sum_{i=1}^{r} R_i(\vec{x}_i, a) + \sum_{j=1}^{k} (w_j B^a[b_j(\vec{x})]) - \sum_{j=1}^{k} w_j b_j(\vec{x}) \; ; \; \forall a, \vec{x} \end{aligned} \tag{3.15}$$

We can exploit the factored nature of the basis functions to simplify the objective to the following compact form where we assume each basis function explicitly depends on the subset of state variables in $\vec{x}_j$:

$$\sum_{\vec{x}} \sum_{j=1}^{k} w_j b_j(\vec{x}_j) = \sum_{j=1}^{k} w_j y_j \tag{3.16}$$

where $y^j = 2^{n-|\vec{x}_j|} \sum_{\vec{x}_j} b_j(\vec{x}_j)$.

Finally, we note that exploiting linearity of the backup operator again provides us with a factored $\max$-$\sum$ form of the ALP LP constraints from Equation 3.15 as it did similarly for the final form of the API LP constraints in Equation 3.14. We discuss an efficient solution to LPs with such $\max$-$\sum$ factored constraints in the next section.

Constraint Generation

In the above LP, both forms for the constraints take on the generic form of a sum of m factors $F_i(\vec{x})$ over (ideally) small sets of variables:

$$0 \geq w_1 \cdot F_1(\vec{x}) + \ldots + w_n \cdot F_m(\vec{x}) \; ; \; \forall \vec{x} \tag{3.17}$$

Not every factor must have a weight w_i, but we note that each factor has at most one linear weight owing to the structure of the original basis functions and the properties of the backup

operators.

To view the constraints in a more concrete form, we note that for every possible instantiation $\vec{x}^*$ of the state, we could simply instantiate the factor F_i to its constant value $c_i = F_i(\vec{x}^*)$ under that state assignment and come up with a corresponding linear constraint:

$$0 \geq w_1 \cdot c_1 + \ldots + w_n \cdot c_m \tag{3.18}$$

We could generate constraints for *all* possible state assignments $\vec{x}^*$ and solve our LP in this manner. However, we would obviously lose the benefits of our factored representation in that we would have to specify a number of constraints that scales exponentially in the number of state variables.

However, if we rewrite the constraints from Equation 3.17 in the following equivalent form where we enforce all constraints simultaneously with one maximization then we can see how to exploit the factored constraint structure:

$$0 \geq \max_{\vec{x}} [w_1 \cdot F_1(\vec{x}) + \ldots + w_n \cdot F_m(\vec{x})] \tag{3.19}$$

This *cost network* [Dechter, 1999] form of these constraints lends itself to very efficient evaluation methods such as variable elimination. The question is how to exploit this property in our LP solution. Fortunately, as it turns out, there are at least two approaches to exploiting this structure.

The first solution, due to Guestrin *et al.* [2002] is to directly simulate variable elimination in the LP encoding of the constraints of Equation 3.19. This leads to a total number of constraints $O(\exp(TW))$ where TW is the induced tree-width of the cost-network under the variable elimination order that was used. This is an attractive method because the structure of the factored MDP and the basis functions should lead to $TW << n$ when (a) the basis functions range over small sets of variables with little or no overlap and (b) the backups of each basis function have similar characteristics due to the property that only small sets of variables affect each other causally in the Bayes net. Thus, simulating variable elimination in the LP variable encoding to produce $O(\exp(TW))$ constraints would be a much more efficient solution than generating $O(\exp(n))$ constraints as would be done in the enumerated state case.

However, a simpler approach and often an empirically faster method in practice[11] is the technique of constraint generation [Schuurmans and Patrascu, 2001; Trick and Zin, 1997]. In

[11] This is despite the lack of similar guarantees on the maximum number of constraints generated.

this case, we perform the following solution procedure where we have specified some solution tolerance ϵ:

1. Initialize LP with $\vec{w}^i = \vec{0}$, $i = 0$, and empty constraint set.

2. For each constraint in the cost-network form of Equation 3.19 instantiated with the current solution $\vec{w}^i$, find the maximally violated constraint C (if one exists) using variable elimination.

3. If C's constraint violation is larger than ϵ, add C to LP constraint set, otherwise return $\vec{w}^i$ as solution.

4. Solve LP for new solution $\vec{w}^{i+1}$, goto step 2

Using these constraint generation techniques, one can now efficiently apply either API or ALP with linear-value approximation to factored MDPs. However in comparing API and ALP, we note that in practice, one cannot always guarantee a compact structure for the policies generated during API. In addition, API requires one optimization of an LP on each iteration until convergence or some stopping criterion is reached. In contrast, ALP does not require a representation of the policy and tends to have a lower tree-width in its constraints. ALP also solves the problem with one LP optimization. Consequently, ALP tends to be much faster than API as noted by Schuurmans and Patrascu [2001], but they also note in their experiments that API produced better policies.

Basis Function Generation

One additional difficulty with linear value function approximation is that of generating a good set of basis functions. Certainly, a set of basis functions that poorly approximate the optimal value function can have an adverse impact on decision quality. Consequently, one can take a number of approaches to generating basis functions such as finding subtasks with additive reward [Poupart *et al.*, 2002a], performing branch-and-bound search to find Bellman-error minimizing basis functions [Poupart *et al.*, 2002b], or analyzing the dual of the LP solution to heuristically generate basis function candidates [Poupart *et al.*, 2002b]. Unfortunately, at this point in time, generating a good basis function set is still more of an art than a science, and there are no currently known methods that allow one to attain *a priori* guarantees on the decision quality for a given set of basis functions.

3.4 Exploiting CSI, Additive, and Multiplicative Independence

Previously we discussed how value iteration could be defined in terms of ADDs — this was specifically exploited in the SPUDD [Hoey *et al.*, 1999] algorithm. Unfortunately, SPUDD only exploits CSI and shared substructure in value functions due to its use of ADDs. Although ADDs can exploit some structure in additive rewards as was shown in Figure 3.1(c), ADDs were not intended to directly exploit additive structure nor can they compactly represent all additive functions. What is needed is a decision diagram that can exploit CSI, additive, and perhaps other forms of structure.

To address this need, we propose an affine extension to ADDs called *affine ADDs (AADDs)* capable of compactly representing context-specific, additive, and multiplicative structure. We show that the AADD has worst-case time and space performance within a multiplicative constant of that of ADDs, but that it can be linear in the number of variables in cases where ADDs are exponential in the number of variables. We provide an empirical comparison of tabular, ADD, and AADD representations used in standard Bayes net and MDP inference algorithms and conclude that the AADD performs at least as well as the other two representations and may yield an exponential performance improvement over both the ADD and tabular representations when additive or multiplicative structure can be exploited.

3.4.1 Limitations of ADDs

As shown in Figure 3.7, ADDs often provide an efficient representation of functions with context-specific independence, such as functions whose structure is conjunctive (3.7a) or disjunctive (3.7b) in nature. Thus, as previously mentioned, ADDs can offer exponential space savings over a fully enumerated tabular representation. However, the compactness of ADDs does not extend to the case of additive or multiplicative independence, as demonstrated by the exponentially large representations when this structure is present (3.7c). Unfortunately such structure may occur in probabilistic and decision-theoretic reasoning domains, potentially leading to exponential running times and space requirements for inference on these problems.

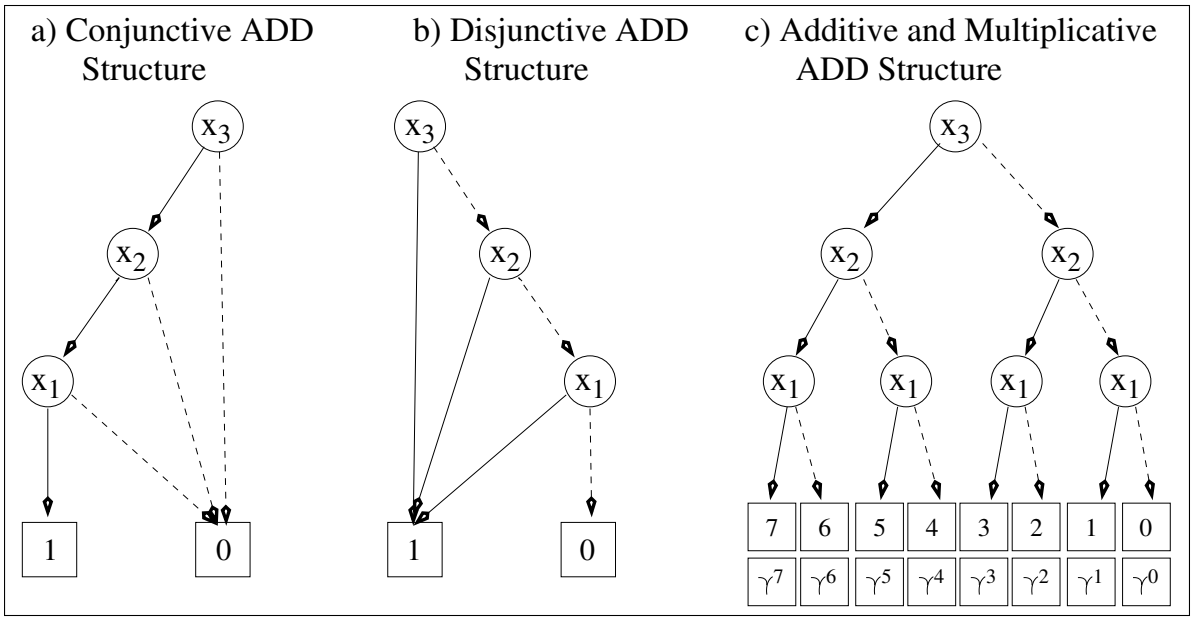


Figure 3.7: Some example ADDs showing a) conjunctive structure ($f = \mathit{if}\ (x_1 \wedge x_2 \wedge x_3)\ \mathit{then}\ 1\ \mathit{else}\ 0$, b) disjunctive structure ($f = \mathit{if}\ (x_1 \vee x_2 \vee x_3)\ \mathit{then}\ 1\ \mathit{else}\ 0$), and c) additive ($f = 4x_3 + 2x_2 + x_1$) and multiplicative ($f = \gamma^{4x_3+2x_2+x_1}$) structure (top and bottom sets of terminal values, respectively). The high (true) edge is solid, the low (false) edge is dotted.

3.4.2 Affine Algebraic Decision Diagrams (AADDs)

To address the limitations of ADDs, we introduce an affine extension to the ADD (AADD) that is capable of canonically and compactly representing context-specific, additive, and multiplicative structure in functions from $\mathbb{B}^n \rightarrow \mathbb{R}$. However, before we formally define AADDs we begin with two examples of AADDs that compactly represent additive and multiplicative structure.

Figure 3.8 shows portions of the exponentially sized ADDs from Figure 3.7c represented by AADDs of linear size. The evaluation of an AADD is essentially the same as ADDs: given a variable assignment, one traverses the AADD from the root to the leaf following branches at each node corresponding to the given variable assignment. However, one will note that the edges are labelled with two parameters $\langle c, b \rangle$ that denote an affine transform of the subnode it points to. That is, if the subnode evaluates to v, then the affine transform of that subnode evaluates to $c + b \cdot v$. This very simple modification to ADDs to specify affine transforms on edges turns out to be quite powerful in that previously exponentially-sized ADDs can be represented as linearly-sized ADDs as shown in these examples.

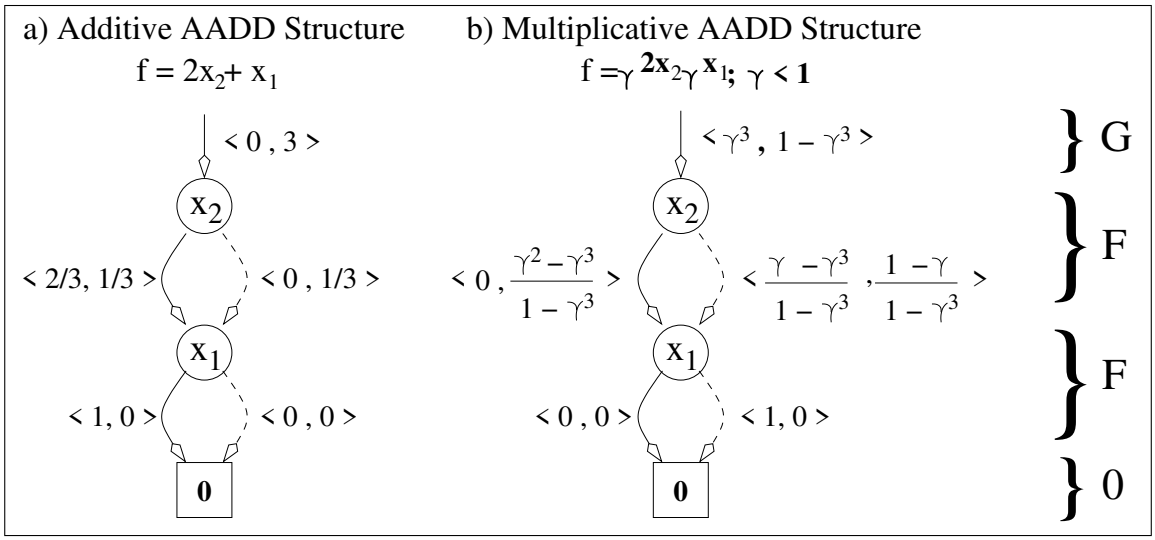


Figure 3.8: Portions of the ADDs from Figure 3.7(c) expressed as generalized AADDs. The edge weights are given as $\langle c, b \rangle$. The curly braces on the right indicate the elements of the AADD grammar that correspond to each portion of the AADD diagram.

Recalling our definitions from Section 3.1.2 for ADDs, we formally define AADDs with the following BNF grammar where F represents a *normalized AADD* that we will subsequently restrict to have maximum range $[0, 1]$ and G represents a *generalized AADD* with range $[c, c+b]$:

$$\begin{aligned} G &::= c + bF \\ F &::= 0 \mid \mathit{if}\ (F^{var})\ \mathit{then}\ c_h + b_h F_h\ \mathit{else}\ c_l + b_l F_l \end{aligned}$$

F may be the constant 0 terminal node or an internal decision node represented as $\mathit{if}\ (F^{var})\ \mathit{then}\ c_h + b_h F_h\ \ \mathit{else}\ c_l + b_l F_l$. Internal decision nodes have essentially the same semantics as they did for ADDs in the BNF grammar from Equation 3.2 except that there is an affine transform $c_h + b_h \cdot F_h$ on the high edge (evaluated when $var = true$) and an affine transform $c_l + b_l \cdot F_l$ on the low edge (evaluated when $var = false$). Here, c_h and c_l are real constants in the closed interval $[0, 1]$, b_h and b_l are real constants in the half-open interval $(0, 1]$, F^{var} is a boolean variable associated with F, and F_l and F_h are of grammar F (i.e., normalized AADDs themselves). We also impose the following constraints to enforce canonicity of the AADD representation:

1. The variable F^{var} does not appear in F_h or F_l.

2. $\min(c_h, c_l) = 0$

3. $\max(c_h + b_h, c_l + b_l) = 1$

4. If $F_h = 0$ then $b_h = 0$ and $c_h > 0$. Similarly for F_l.

5. In the grammar for G, we require that if $F = 0$ then $b = 0$, otherwise $b > 0$.

These constraints require that F is normalized to have range $[0, 1]$ (when $F \neq 0$). Since normalized AADDs in grammar F are restricted to the range $[0, 1]$, we need the top-level positive affine transform of generalized AADDs in grammar G to allow for the representation of functions with arbitrary range. One can verify that these constraints hold for the AADDs in Figure 3.8 where all variable and terminal nodes are normalized AADD nodes in the grammar for F and the affine transform for the root node of the AADD is a generalized node in the grammar for G.

Let $Val(F, \rho)$ be the value of AADD F under variable value assignment ρ. This can be defined recursively by the following equation:

$$Val(F, \rho) = \begin{cases} F = 0 : & 0 \\ F \neq 0 \wedge \rho(F^{var}) = true : & c_h + b_h \cdot Val(F_h, \rho) \\ F \neq 0 \wedge \rho(F^{var}) = false : & c_l + b_l \cdot Val(F_l, \rho) \end{cases}$$

Lemma 3.4.1. *For any normalized AADD F over a variable domain $x_1, \ldots, x_n$ and for all variable assignments ρ to variables in F's domain, we have that $Val(F, \rho)$ is in the interval $[0, 1]$, $\min_\rho Val(F, \rho) = 0$, and if $F \neq 0$ then $\max_\rho Val(F, \rho) = 1$.*

Proof. For the base case of $F = 0$, the lemma obviously holds. Now, for $F \neq 0$, we inductively assume that F_l and F_h satisfy the lemma and are in the interval $[0, 1]$. Then for F, we obtain the range $[\min(c_h + \min(F_h), c_l + \min(F_l)), \max(c_h + b_h \cdot \max(F_h), c_l + b_l \cdot \max(F_l))]$, which simplifies to $[\min(c_h, c_l), \max(c_h + b_h, c_l + b_l)]$ based on our inductive assumption. Our previous constraints (2) and (3) then imply the range of F is $[0, 1]$, which proves the inductive case. □

Recalling our previous definition of variable ordering for ADDs, we say that F satisfies a given variable ordering if $F = 0$ or F is of the form $if\ (F^{var})\ then\ c_h + b_h F_h\ else\ c_l + b_l F_l$ where F^{var} does not occur in F_h or F_l and F^{var} is the earliest variable under the given ordering occuring in F. We say that a generalized AADD of form $c + bF$ satisfies the order if F satisfies the order.

Lemma 3.4.2. *Fix a variable ordering over $x_1, \ldots, x_n$. For any non-constant function $g(x_1, \ldots, x_n)$ mapping $\mathbb{B}^n \longrightarrow \mathbb{R}$, there exists a unique generalized AADD G over variable domain*

Algorithm 4: $GetGNode(v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle) \longrightarrow \langle c_r, b_r, F_r\rangle$

input : $v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle$: Var, offset, mult, and node id for high/low branches
output : $\langle c_r, b_r, F_r\rangle$: Return values for offset, multiplier, and canonical node id
begin
// If branches redundant, return child
if *(*$c_l = c_h \wedge b_l = b_h \wedge F_l = F_h$*)* **then**
return $\langle c_l, b_l, F_l\rangle$;
// Non-redundant so compute canonical form
$r_{min} := \min(c_l, c_h)$;
$r_{max} := \max(c_l + b_l, c_h + b_h)$;
$r_{range} := r_{max} - r_{min}$;
$c_l := (c_l - r_{min})/r_{range}$;
$c_h := (c_h - r_{min})/r_{range}$;
$b_l := b_l/r_{range}$;
$b_h := b_h/r_{range}$;
// Make new node if not in cache
if *(*$\langle v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle\rangle \rightarrow id$ *is not in node cache)* **then**
$id :=$ currently unallocated id;
insert $\langle v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle\rangle \rightarrow id$ in cache;
// Return the cached, canonical node
return $\langle r_{min}, r_{range}, id\rangle$;
end

$x_1, \ldots, x_n$ *satisfying the given variable ordering such that for all* $\rho \in \mathbb{B}^n$ *we have* $g(\rho) = Val(G, \rho)$.

Proof. See Section B.1 of Appendix B.

This second lemma shows that under a given variable ordering, generalized AADDs are canonical, i.e., two identical functions will always have identical AADD representations.

3.4.3 Algorithms

We now define AADD algorithms that are analogs of those previously given for ADDs. As such, familiarity with the $GetNode$, $Reduce$, and $Apply$ algorithms from Section 3.1.2 will greatly aid in understanding the extensions to these algorithms for AADDs.

Similar to ADDs, we begin by defining a procedure for maintaining a cache of unique AADD nodes. All algorithms rely on the helper function $GetGNode$ given in Algorithm 4 that takes an unnormalized AADD node of the form $if\ (v)\ then\ c_h + b_h F_h\ else\ c_l + b_l F_l$ and returns

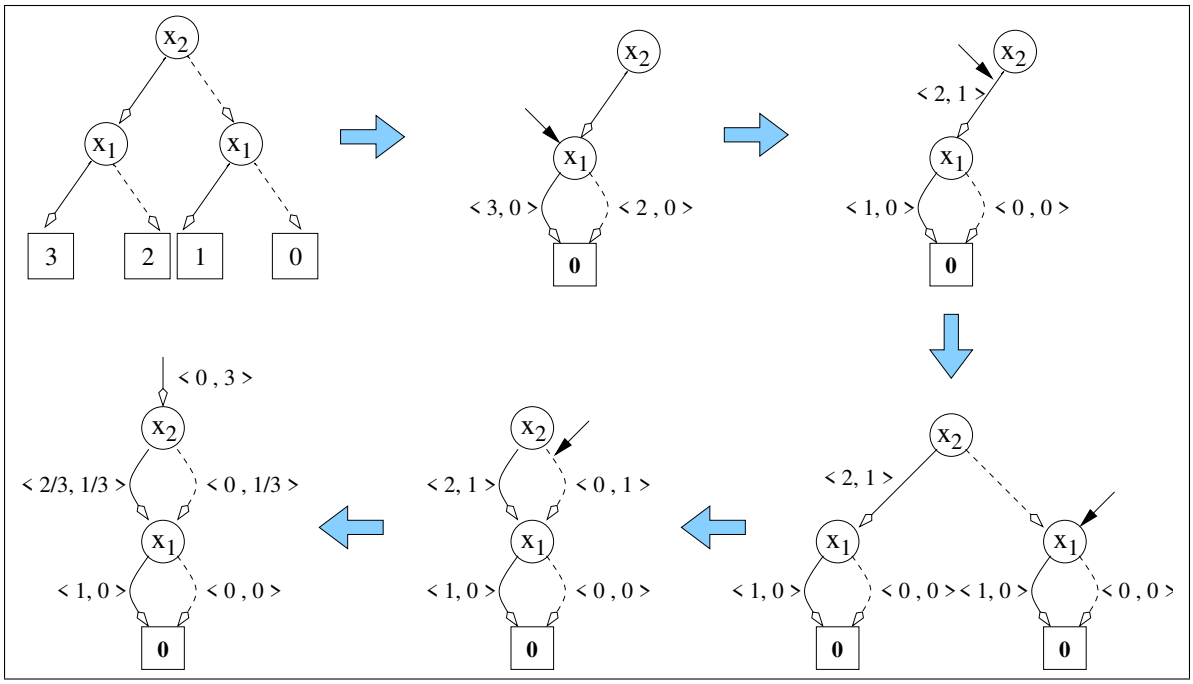


Figure 3.9: An example application of the $Reduce$ algorithm. The input is the top, leftmost diagram (all edge weights are assumed to be $\langle 0, 1 \rangle$). The solid arrow shows the node currently being evaluated by $Reduce$ while the next diagram shows the result after this evaluation; when the solid arrow is on a branch rather than a node itself, it indicates that it is completing the evaluation of that branch within the $Reduce$ call for the parent node. The bottom, leftmost diagram is the final canonical AADD representation of the input.

the unique cached, generalized[12] AADD node of the form $\langle c_r + b_r F_r \rangle$. As for $GetNode$ with ADDs, such a procedure is needed to ensure that there is a single unique node representing any given function.[13]

Then, given a potentially unnormalized representation of an entire AADD, we define an AADD generalization of the $Reduce$ algorithm that constructs a corresponding canonical generalized AADD, removing any redundant structure in the process. Next, we define an AADD generalization of the $Apply$ algorithm to specify an efficient procedure for performing binary operations on these AADDs. From these operations, we can then build the remaining operations such as unary $\min$ and $\max$ and marginalization that we will need for probabilistic inference.

At an abstract level, one can view the $GetNode$, $Reduce$, and $Apply$ algorithms for AADDs

[12]Thus the "G" in the procedure name for $GetGNode$.

[13]Throughout all of the algorithms we use the tuple representation $\langle c, b, F \rangle$, while in the text we often use the equivalent notation $\langle c + bF \rangle$ to make the node semantics more clear.

Algorithm 5: $Reduce(\langle c, b, F \rangle) \longrightarrow \langle c_r, b_r, F_r \rangle$

```
input  : ⟨c, b, F⟩ : Offset, multiplier, and node id
output : ⟨c_r, b_r, F_r⟩ : Return values for offset,
         multiplier, and node id
begin
    // Check for terminal node
    if (F = 0) then
        return ⟨c, 0, 0⟩;

    // Check reduce cache
    if ( F → ⟨c_r, b_r, F_r⟩ is not in reduce cache) then
        // Not in cache, so recurse
        ⟨c_h, b_h, F_h⟩ := Reduce(c_h, b_h, F_h);
        ⟨c_l, b_l, F_l⟩ := Reduce(c_l, b_l, F_l);

        // Retrieve canonical form
        ⟨c_r, b_r, F_r⟩ := GetGNode(F^var, ⟨c_h, b_h, F_h⟩, ⟨c_l, b_l, F_l⟩);

        // Put in cache
        insert F → ⟨c_r, b_r, F_r⟩ in reduce cache;

    // Return canonical reduced node
    return ⟨c + b · c_r, b · b_r, F_r⟩;
end
```

as essentially identical to those for ADDs except that they are extended to propagate the affine transform of the edge weights on recursion and to compute the normalization of the resulting node on return.

Reduce

The $Reduce$ algorithm given in Algorithm 5 takes an arbitrary ordered AADD, normalizes and caches the internal nodes, and returns the corresponding generalized AADD. This produces a unique representation of the AADD that removes any redundant structure in the input representation. One will note that the $Reduce$ algorithm precisely follows the constructive proof in Lemma 3.4.2. This is sufficient to prove correctness of the algorithm. An example application of the $Reduce$ algorithm is given in Figure 3.9.

One nice property of the $Reduce$ algorithm is that one does not need to prespecify the structure that the AADD should exploit. If the represented function contains context-specific, additive, or multiplicative independence, the $Reduce$ algorithm will compactly represent this structure uniquely and automatically w.r.t. the variable ordering as guaranteed by previous lemmas.

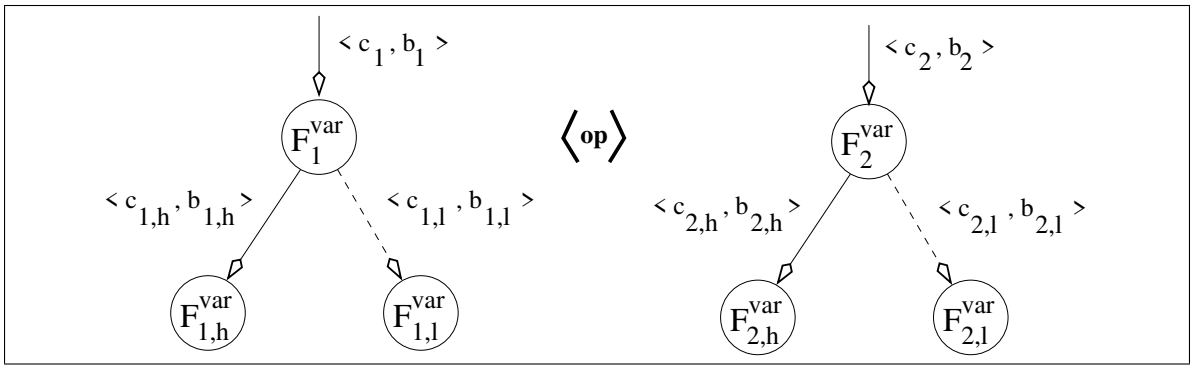


Figure 3.10: Two AADD nodes F_1 and F_2 and a binary operation op with the corresponding notation used in the presentation of the $Apply$ algorithm.

Apply

We let op denote a binary operator on AADDs with possible operations being addition, subtraction, multiplication, division, min, and max denoted respectively as $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\min(\cdot, \cdot)$, and $\max(\cdot, \cdot)$. We do not explicitly provide binary comparison functions $\geq$, $>$, $\leq$, $<$ for AADDs as we did for ADDs, but note that they could be easily defined analogously to the other binary operations, if needed.

The $Apply$ routine given in Algorithm 6 takes two generalized AADD operands and an operation as given in Figure 3.10 and produces the resulting generalized AADD. The control flow of the algorithm is straightforward: We first check whether we can compute the result immediately, otherwise we normalize the operands to a canonical form and check if we can reuse the result of a previously cached computation. If we can do neither of these, we then choose a variable to branch on and recursively call the $Apply$ routine for each instantiation of the variable. We cover these steps in-depth in the following sections.

Terminal computation The function $ComputeResult$ given in the *top half* of Table 3.2, determines if the result of a computation can be immediately computed without recursion. The first entry in this table is required for proper termination of the algorithm as it computes the result of an operation applied to two terminal 0 nodes. However, the other entries denote a number of pruning optimizations that immediately return a node without recursion. For example, given the operation $\langle 3 + 4F_1 \rangle \oplus \langle 5 + 6F_1 \rangle$, we can immediately return the result $\langle 8 + 10F_1 \rangle$ since F_1 is shared by both operands.

$ComputeResult(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle c_r, b_r, F_r\rangle$	
Operation and Conditions	**Return Value**
$\langle c_1 + b_1F_1\rangle \langle op\rangle \langle c_2 + b_2F_2\rangle;\ \ F_1 = F_2 = 0$	$\langle (c_1 \langle op\rangle c_2) + 0 \cdot 0\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle);\ \ c_1 + b_1 \leq c_2$	$\langle c_2 + b_2F_2\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle);\ \ c_2 + b_2 \leq c_1$	$\langle c_1 + b_1F_1\rangle$
$\langle c_1 + b_1F_1\rangle \oplus \langle c_2 + b_2F_2\rangle;\ \ F_1 = F_2$	$\langle (c_1 + c_2) + (b_1 + b_2)F_1\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_1\rangle);\ \ F_1 = F_2,$ $(c_1 \geq c_2 \wedge b_1 \geq b_2) \vee (c_2 \geq c_1 \wedge b_2 \geq b_1)$	$c_1 \geq c_2 \wedge b_1 \geq b_2 : \langle c_1 + b_1F_1\rangle$ $c_2 \geq c_1 \wedge b_2 \geq b_1 : \langle c_2 + b_2F_1\rangle$
Note: for all $\max$ operations above, return opposite for $\min$	
$\langle c_1 + b_1F_1\rangle \langle op\rangle \langle c_2 + b_2F_2\rangle;\ \ F_2 = 0, op \in \{\oplus, \ominus\}$	$\langle (c_1 \langle op\rangle c_2) + b_1F_1\rangle$
$\langle c_1 + b_1F_1\rangle \langle op\rangle \langle c_2 + b_2F_2\rangle;\ \ F_2 = 0, c_2 \geq 0, op \in \{\otimes, \oslash\}$	$\langle (c_1 \langle op\rangle c_2) + (b_1 \langle op\rangle c_2)F_1\rangle$
Note: above two operations can be modified to handle $F_1 = 0$ when $op \in \{\oplus, \otimes\}$	
other	$null$

$GetNormCacheKey(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle\langle c'_1, b'_1\rangle\langle c'_2, b'_2\rangle\rangle$ **and** $ModifyResult(\langle c_r, b_r, F_r\rangle) \longrightarrow \langle c'_r, b'_r, F'_r\rangle$		
Operation and Conditions	**Normalized Cache Key and Computation**	**Result Modification**
$\langle c_1 + b_1F_1\rangle \oplus \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \oplus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 + c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \ominus \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \ominus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 - c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \otimes \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \otimes \langle (c_2/b_2) + F_2\rangle$	$\langle b_1b_2c_r + b_1b_2b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \oslash \langle c_2 + b_2F_2\rangle;\ F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \oslash \langle (c_2/b_2) + F_2\rangle$	$\langle (b_1/b_2)c_r + (b_1/b_2)b_rF_r\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle);$ $F_1 \neq 0$, Note: same for $\min$	$\langle c_r + b_rF_r\rangle = \max(\langle 0 + 1F_1\rangle, \langle (c_2 - c_1)/b_1 + (b_2/b_1)F_2\rangle)$	$\langle (c_1 + b_1c_r) + b_1b_rF_r\rangle$
any $\langle op\rangle$ not matching above: $\langle c_1 + b_1F_1\rangle \langle op\rangle \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle = \langle c_1 + b_1F_1\rangle \langle op\rangle \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle$

Table 3.2: Input and output summaries of the $ComputeResult$, $GetNormCacheKey$, and $ModifyResult$ routines.

Algorithm 6: $Apply(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle c_r, b_r, F_r\rangle$

input : $\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op$: Nodes and op
output : $\langle c_r, b_r, F_r\rangle$: Generalized node to return
begin
  // Check if result can be immediately computed
  if *($ComputeResult(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \rightarrow \langle c_r, b_r, F_r\rangle$ is not null)* **then**
    return $\langle c_r, b_r, F_r\rangle$;
  // Get normalized key and check apply cache
  $\langle\langle c'_1, b'_1\rangle, \langle c'_2, b'_2\rangle\rangle :=$
    $GetNormCacheKey(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op)$;
  if *($\langle\langle c'_1, b'_1, F_1\rangle, \langle c'_2, b'_2, F_2\rangle, op\rangle \rightarrow \langle c_r, b_r, F_r\rangle$ is not in apply cache)* **then**
    // Not terminal, so recurse
    if *(F_1 is a non-terminal node)* **then**
      if *(F_2 is a non-terminal node)* **then**
        if *(F_1^{var} comes before F_2^{var})* **then**
          $var := F_1^{var}$;
        else
          $var := F_2^{var}$;
      else
        $var := F_1^{var}$;
    else
      $var := F_2^{var}$;
    // Propagate affine transform to branches
    if *(F_1 is non-terminal $\wedge$ $var = F_1^{var}$)* **then**
      $F_l^{v1} := F_{1,l}$; $F_h^{v1} := F_{1,h}$;
      $c_l^{v1} := c'_1 + b'_1 \cdot c_{1,l}$; $c_h^{v1} := c'_1 + b'_1 \cdot c_{1,h}$;
      $b_l^{v1} := b'_1 \cdot b_{1,l}$; $b_h^{v1} := b'_1 \cdot b_{1,h}$;
    else
      $F_{l/h}^{v1} := F_1$; $c_{l/h}^{v1} := c'_1$; $b_{l/h}^{v1} := b'_1$;
    if *(F_2 is non-terminal $\wedge$ $var = F_2^{var}$)* **then**
      $F_l^{v2} := F_{2,l}$; $F_h^{v2} := F_{2,h}$;
      $c_l^{v2} := c'_2 + b'_2 \cdot c_{2,l}$; $c_h^{v2} := c'_2 + b'_2 \cdot c_{2,h}$;
      $b_l^{v2} := b'_2 \cdot b_{2,l}$; $b_h^{v2} := b'_1 \cdot b_{2,h}$;
    else
      $F_{l/h}^{v2} := F_2$; $c_{l/h}^{v2} := c'_2$; $b_{l/h}^{v2} := b'_2$;
    // Recurse and get cached result
    $\langle c_l, b_l, F_l\rangle := Apply(\langle c_l^{v1}, b_l^{v1}, F_l^{v1}\rangle, \langle c_l^{v2}, b_l^{v2}, F_l^{v2}\rangle, op)$;
    $\langle c_h, b_h, F_h\rangle := Apply(\langle c_h^{v1}, b_h^{v1}, F_h^{v1}\rangle, \langle c_h^{v2}, b_h^{v2}, F_h^{v2}\rangle, op)$;
    $\langle c_r, b_r, F_r\rangle := GetGNode(var, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle)$;
    // Put result in apply cache and return
    insert $\langle c'_1, b'_1, F_1, c'_2, b'_2, F_2, op\rangle \rightarrow \langle c_r, b_r, F_r\rangle$ into apply cache;
  return $ModifyResult(\langle c_r, b_r, F_r\rangle)$;
end

Recursive computation If a call to $Apply$ is unable to immediately compute a result or reuse a previously cached computation, we must recursively compute the result. For this we have two cases (the third case where both operands are 0 terminal nodes having been taken care of in the previous section):

- **F_1 or F_2 is a 0 terminal node, or $F_1^{var} \neq F_2^{var}$:** We assume the operation is commutative and reorder the operands so that F_1 is the 0 node or the operand whose variable comes *later* in the variable ordering so that we know to branch on F_2^{var} first.[14] Then we propagate the affine transform to each of F_2's branches and compute the operation applied separately to F_1 and *each* of F_2's high and low branches. We then build an if statement conditional on F_2^{var} and normalize it to obtain the generalized AADD node $\langle c_r, b_r, F_r \rangle$ for the result:

$$\begin{aligned}
\langle c_h, b_h, F_h \rangle &= Apply(\langle c_1, b_1, F_1 \rangle, \langle c_2 + b_2 c_{2,h}, b_2 b_{2,h}, F_{2,h} \rangle, op) \\
\langle c_l, b_l, F_l \rangle &= Apply(\langle c_1, b_1, F_1 \rangle, \langle c_2 + b_2 c_{2,l}, b_2 b_{2,l}, F_{2,l} \rangle, op) \\
\langle c_r, b_r, F_r \rangle &= GetGNode(F_2^{var}, \langle c_h, b_h, F_h \rangle, \langle c_l, b_l, F_l \rangle)
\end{aligned}$$

- **F_1 and F_2 are non-terminal nodes and $F_1^{var} = F_2^{var}$:** Since the variables for each operand match, we know the result $\langle c_r, b_r, F_r \rangle$ is simply a generalized if statement branching on F_1^{var} (= F_2^{var}) with the true case being the operator applied to the high branches of F_1 and F_2 and likewise for the false case and the low branches:

$$\begin{aligned}
\langle c_h, b_h, F_h \rangle &= Apply(\langle c_1 + b_1 c_{1,h}, b_1 b_{1,h}, F_{1,h} \rangle, \\
&\qquad \langle c_2 + b_2 c_{2,h}, b_2 b_{2,h}, F_{2,h} \rangle, op) \\
\langle c_l, b_l, F_l \rangle &= Apply(\langle c_1 + b_1 c_{1,l}, b_1 b_{1,l}, F_{1,l} \rangle \\
&\qquad \langle c_2 + b_2 c_{2,l}, b_2 b_{2,l}, F_{2,l} \rangle, op) \\
\langle c_r, b_r, F_r \rangle &= GetGNode(F_1^{var}, \langle c_h, b_h, F_h \rangle, \langle c_l, b_l, F_l \rangle)
\end{aligned}$$

Canonical caching If the AADD $Apply$ algorithm were to compute and cache the results of applying an operation directly to the operands, the algorithm would provably have the same

[14] As for ADDs, we note that the first case prohibits the use of the non-commutative $\ominus$ and $\oslash$ operations. However, a simple solution would be to recursively descend on either F_1 or F_2 rather than assuming commutativity and swapping operands to ensure descent on F_2. To accommodate general non-commutative operations, we have used this alternate approach in our specification of the $Apply$ routine given in Algorithm 6.

time complexity as the ADD $Apply$ algorithm. Yet, if we were to compute $\langle 0+1F_1\rangle \oplus \langle 0+2F_2\rangle$ and cache the result $\langle c_r + b_r F_r\rangle$, we could compute $\langle 5 + 2F_1\rangle \oplus \langle 4 + 4F_2\rangle$ without recursion as follows:

$$\begin{aligned}
&\text{(a)} \qquad \langle 5 + 2F_1\rangle \oplus \langle 4 + 4F_2\rangle &&= 9 + 2 \cdot (\langle 0 + F_1\rangle \oplus \langle 0 + 2F_2\rangle) \\
&\text{(b)} &&= 9 + 2 \cdot \langle c_r + b_r F_r\rangle \\
&\text{(c)} &&= \langle (9 + 2c_r) + 2b_r F_r\rangle
\end{aligned}$$

The key observation here is that we can (a) rewrite the second operation in a normalized form where we subtract off the constants and divide by the first coefficient, (b) substitute in the result of a previously cached computation, and then (c) modify the result to reverse the previous normalization.

This suggests a canonical caching scheme that normalizes all cache entries to increase the chance of a cache hit. The actual result can then be easily computed from the cached result by reversing the normalization as demonstrated in the example. This ensures optimal reuse of the $Apply$ operations cache and can lead to an exponential reduction in running time over the non-canonical caching version.

We introduce two additional functions to perform this caching: $GetNormCacheKey$ to compute the canonical cache key, and $ModifyResult$ to reverse the normalization in order to compute the actual result. These algorithms are summarized in the *bottom half* of Table 3.2.

Other Operations

We summarize some of the remaining operations that can be performed (efficiently) on AADDs:

- **min and max computation:** The min and max of a generalized AADD node $\langle c + bF\rangle$ are respectively c and $c + b$ due to $[0, 1]$ normalization of F.

- **Restriction:** The restriction of a variable x_i in a function to either *true* or *false* (i.e. $F|_{x_i=T/F}$) can be computed similarly to ADDs by replacing all decision nodes for variable x_i with the branch corresponding to the variable restriction and propagating the affine transform to the direct subnodes. Then $Reduce$ can be applied on the resulting decision diagram to convert it to a canonical AADD.

- **Sum out/marginalization:** A variable x_i can be summed (or marginalized) out of a function F simply by computing the sum of the restricted functions (i.e. $F|_{x_i=T} \oplus F|_{x_i=F}$)

exactly as done for ADDs.

- **Negation/reciprocation:** While it may seem that negation of a generalized AADD node $\langle c + bF \rangle$ would be as simple as $\langle -c + -bF \rangle$, we note that this violates our normalization scheme which requires $b > 0$. Consequently, negation must be performed explicitly with the $Apply$ operation as $0 \ominus \langle c + bF \rangle$. Likewise, reciprocation (i.e., $\frac{1}{\langle c+bF \rangle}$) must be performed explicitly with the $Apply$ operation as $1 \oslash \langle c + bF \rangle$.

- **Variable reordering:** Rudell's [1993] ADD variable reordering algorithm previously summarized for ADDs can be applied to AADDs without loss of efficiency. The only modification needed is to recompute the normalized affine transforms for pairwise rotations of neighboring nodes involving variables x_i and x_j, but this is simply a local application of the $Reduce$ algorithm.

Cache Implementation

If one were to use a naive cache implementation that relied on exact floating-point values for hashing and equality testing, one would find that many nodes which *should* be the same under exact computation often turn out to have offsets or multipliers differing by $\pm 1e$-15; these numerical precision errors result from repeated multiplications and divisions during the $Reduce$ and $Apply$ operations. This can result in an exponential explosion of nodes if not controlled. Consequently, it is better to use a hashing scheme that considers equality within some range of numerical precision error ϵ. While it is difficult to guarantee such an exact property for an *efficient* hashing scheme, we next outline an approximate approach that we have found to work both efficiently and nearly optimally in practice.

The node cache used in $GetGNode$ and the operation result cache used in $Apply$ both use cache keys containing four floating-point values (i.e., the offsets and multipliers for two AADD nodes). If we consider this 4-tuple of floating-point values to be a point in Euclidean space, then we can measure the distance between two 4-tuples $\langle u_1, u_2, u_3, u_4 \rangle$ and $\langle v_1, v_2, v_3, v_4 \rangle$ as the $\mathcal{L}_2$ (Euclidean) distance between these points. In an approximate caching scheme that takes numerical precision error into account, we might consider two 4-tuples corresponding to hash keys to be equivalent if their $\mathcal{L}_2$ distance from each other is smaller than ϵ:

$$\sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + (u_3 - v_3)^2 + (u_4 - v_4)^2} < \epsilon \tag{3.20}$$

Ideally, when probing the cache to see if a key exists within an $\mathcal{L}_2$ distance of ϵ, we would

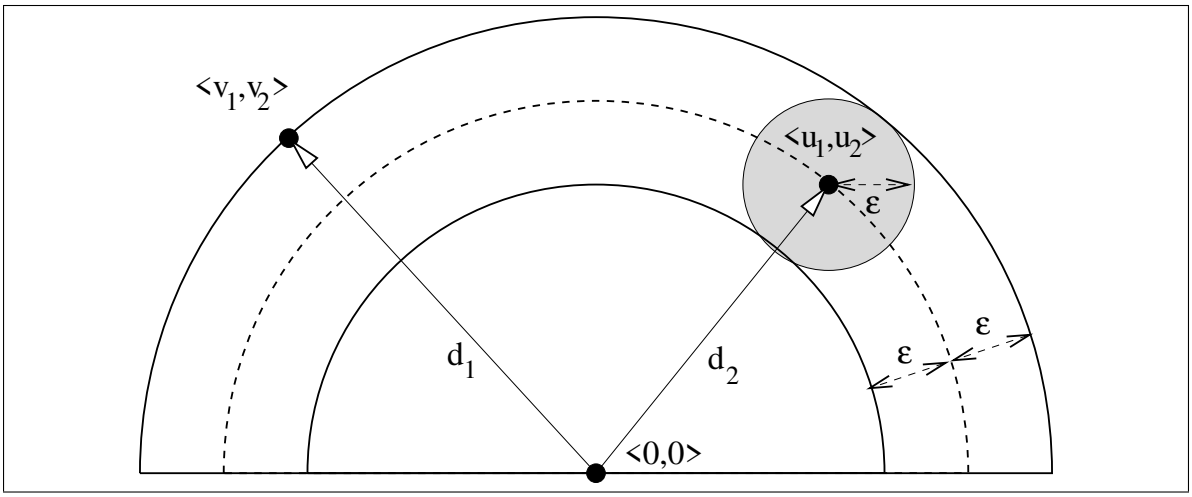


Figure 3.11: A geometric representation of the hashing scheme we use. All points within ϵ of $\langle u_1, u_2 \rangle$ (the shaded circle) lie within the ring having outer and inner radius $\sqrt{{u_1}^2 + {u_2}^2} \pm \epsilon$. Thus, a hashing scheme which hashes all points within the ring to the *same* bucket guarantees that all points within ϵ of $\langle u_1, u_2 \rangle$ also hash to the same bucket. Note that buckets are discretized according to the distance from the origin (i.e., the vantage point for comparison).

prefer to avoid a pairwise comparison of our probe key to all nodes currently in the cache. Fortunately, we can use the *vantage point* [Yianilos, 1993] method for efficiently finding nearest neighbors in a metric space. The basic idea of these methods is that we can exploit the triangle inequality to obtain the following necessary conditions implied by the previous error between two 4-tuples:

$$\sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + (u_3 - v_3)^2 + (u_4 - v_4)^2} \leq \epsilon$$
$$\implies |\sqrt{{u_1}^2 + {u_2}^2 + {u_3}^2 + {u_4}^2} - \sqrt{{v_1}^2 + {v_2}^2 + {v_3}^2 + {v_4}^2}| \leq \epsilon$$

A geometric representation providing intuitions for these necessary conditions is given for two dimensions in Figure 3.11. The benefit of these necessary conditions is that their computation only requires the relative distances of each 4-tuple to the origin (thus, we can view the origin as the vantage point for comparison). While this only gives us a necessary condition in our search for 4-tuples within some $\mathcal{L}_2$ distance of the probe, it gives us a simple test that allows us to prune out the majority of 4-tuples that we need to consider in a typical case.

Based on these necessary conditions for Equation 3.20, we can use the following approximate hashing scheme that will determine other 4-tuples in the hash table that are candidates for being within ϵ distance of a probe $\langle v_1, v_2, v_3, v_4 \rangle$: Compute the $\mathcal{L}_2$ distance d between

$\langle v_1, v_2, v_3, v_4 \rangle$ and the origin. To compute the hash key for $\langle v_1, v_2, v_3, v_4 \rangle$, extract only the bits of the floating-point representation of d representing a fractional portion greater than ϵ and use this for an integer representation of the hash key (we are effectively discretizing the distances into buckets of width ϵ). For equality testing in the hash table, test that the true $\mathcal{L}_2$ metric between a tuple $\langle u_1, u_2, u_3, u_4 \rangle$ and the probe $\langle v_1, v_2, v_3, v_4 \rangle$ is less than ϵ.

While this hashing scheme does not guarantee that all 4-tuples having ϵ distance from the origin $\langle 0, 0, 0, 0 \rangle$ hash to the same bucket (some 4-tuples within ϵ could fall over bucket boundaries), we found that with bucket width $\epsilon = 1e\text{-}9$ and numerical precision error generally less than $1e$-13, there was only a small chance of two nodes within ϵ distance hashing to different buckets. For the empirical results we describe, this hashing scheme was sufficient to prevent any uncontrollable cases of numerical precision error.

An alternate (and exact) hashing scheme would be to explicitly check the neighboring bucket for matching 4-tuples when the probe comes within ϵ of a bucket boundary.[15]

3.4.4 Theoretical Results

Here we present two fundamental results for AADDs. The first theorem bounds the worst-case space and time performance of the $Reduce$ and $Apply$ operations for AADDs in terms of the corresponding operations on ADDs:

Theorem 3.4.3. *For all functions $F_1 : \mathbb{B}^n \longrightarrow \mathbb{R}$ and $F_2 : \mathbb{B}^m \longrightarrow \mathbb{R}$ ($n \geq 0$ and $m \geq 0$), the time and space performance of $Reduce(F_1)$ and $Apply(F_1, F_2, \ op\ )$ for AADDs (operands and results represented as canonical AADDs) is within a multiplicative constant of $Reduce(F_1)$ and $Apply(F_1, F_2, \ op\ )$ for ADDs (operands and results represented as canonical ADDs) in the worst case assuming any fixed variable ordering.*

Proof. See Section B.1 of Appendix B.

While the above results bound the space and time of the AADD operations on arbitrary functions relative to the ADD operations for the same functions, it is interesting to note that the worst case space and time bounds for the $Apply$ operation given *solely* in terms of the corresponding size of the input operands is very different for ADDs vs. AADDs.

The size of the result of the ADD $Apply$ operation is known to be bounded quadratically in the size of the largest input operand. Bryant [1986] shows this simply by observing that the size of the ADD can be bounded in the number of possible distinct $Apply$ calls given two

[15] Thanks to Roni Khardon for suggesting this modification.

operands (any non-distinct calls will already have been cached), which is at most all possible pairs of nodes when taking one node each from the first and second operands. The number of these node pairs is obviously quadratic in the size of the largest input operand. Since each (recursive) $Apply$ call can contribute a maximum of one node to the ADD resulting from the $Apply$ operation, the space bound of $Apply$ follows.

On the other hand, we note that the size of the result of the AADD $Apply$ operation can only be bounded exponentially in the combined size of the operands. To understand this, note that unlike ADDs, AADDs do allow reconvergent edges when these edges are labelled with different affine transforms of the same child node. For example, this can be observed in the linearly structured AADDs of Figure 3.8. Let n be the number of nodes in a linearly structured AADD; then an $Apply$ call with one of these AADDs as operands may need to traverse all possible distinct paths from root node to terminal node, which is $\exp(n)$ due to the reconvergent structure. Following the same reasoning as for the ADD, this exponential number of $Apply$ calls can lead to a result of the $Apply$ operation that has a number of nodes exponential in the combined size of the operands (in the worst case).

Nonetheless, it is important to reiterate the result of Theorem 3.4.3 that the time and space complexity of operations on functions represented as AADDs is never more than a constant times worse than the operations applied to the same functions represented as ADDs.

The second theorem shows that in special cases, the AADD can yield an exponential-to-linear reduction in the time and space complexity over the ADD:

Theorem 3.4.4. *There exist functions F_1 and F_2 and an operator op such that the running time and space performance of $Apply(F_1, F_2, op)$ for AADDs can be linear in the number of variables when the corresponding ADD operations are exponential in the number of variables.*

Proof. See Section B.1 of Appendix B.

Empirically, we note that while the use of AADDs in place of ADDs has always led to smaller space requirements and faster operations for all of our test cases, the rather extreme best case of a reduction from exponential to linear complexity noted in Theorem 3.4.4 has rarely been observed in practice. And perhaps more disappointingly, functions that may appear to have additive and multiplicative structure that can be exploited extensively by AADDs turn out to benefit little from the use of AADDs in place of ADDs. For example, the AADD representation of the function $(\sum_{i=1}^{n} 2^i x_i)^2$ requires precisely 1/4 of the space of the ADD (for $n > 2$) even though the additive and multiplicative structure inherent in this function

ostensibly suggest that the AADD might achieve a substantially more compact representation than the ADD. Nonetheless, a 75% reduction in space obtained by using the AADD in place of the ADD for this example still justifies the use of the AADD in this case.

3.4.5 Empirical Results

First we explore the running time and space requirements of ADDs and AADDs for simple operations such as summation, multiplication, and maximization. Then we explore a number of paradigms for structured probabilistic inference and compare the performance of standard algorithms implemented using ADDs and tabular representations to those using AADDs.

Basic Operations

Figure 3.12 demonstrates the relative time and space performance of tables, ADDs, and AADDs for $\oplus$, $\otimes$, and $\max$, each for one example function. These verify the exponential to linear space and time reductions proved in Theorem 3.4.4. The functions used in these examples are simply generalizations of the additive and multplicative functions given in Figures 3.7c and 3.8 that could be represented in exponential space with ADDs and linear space with AADDs.

Bayes Nets

Since dynamic Bayes nets are used in factored MDPs, it is informative to evaluate AADDs on a variety of Bayes net structures. For Bayes nets, we simply evaluate the variable elimination algorithm [Zhang and Poole, 1996] under the greedy tree-width minimizing *min-fill* [Kjaerulff, 1990] variable ordering with the conditional probability tables (CPTs) P_j and corresponding operations replaced with those for tables, ADDs, and AADDs:

$$\sum_{x_i \notin Query} \prod_{P_1 \ldots P_j} P_1(x_1|Parents(x_1)) \cdots P_j(x_j|Parents(x_j))$$

Table 3.3 shows the total number of table entries/nodes required to represent the original network and the total running time of 100 random queries (each consisting of one query variable and one evidence variable) for a number of publicly available Bayes nets[16] and a *noisy-or* [Pearl, 1986] model $P(x_1|x_2, \ldots, x_n) = 1 - \prod_{i=2}^{n} P(x_1|x_i)$ where $P(x_1|x_i) = .1$ with $n = 15$.

[16] See the *Bayes net repository*: *http://www.cs.huji.ac.il/labs/compbio/Repository*

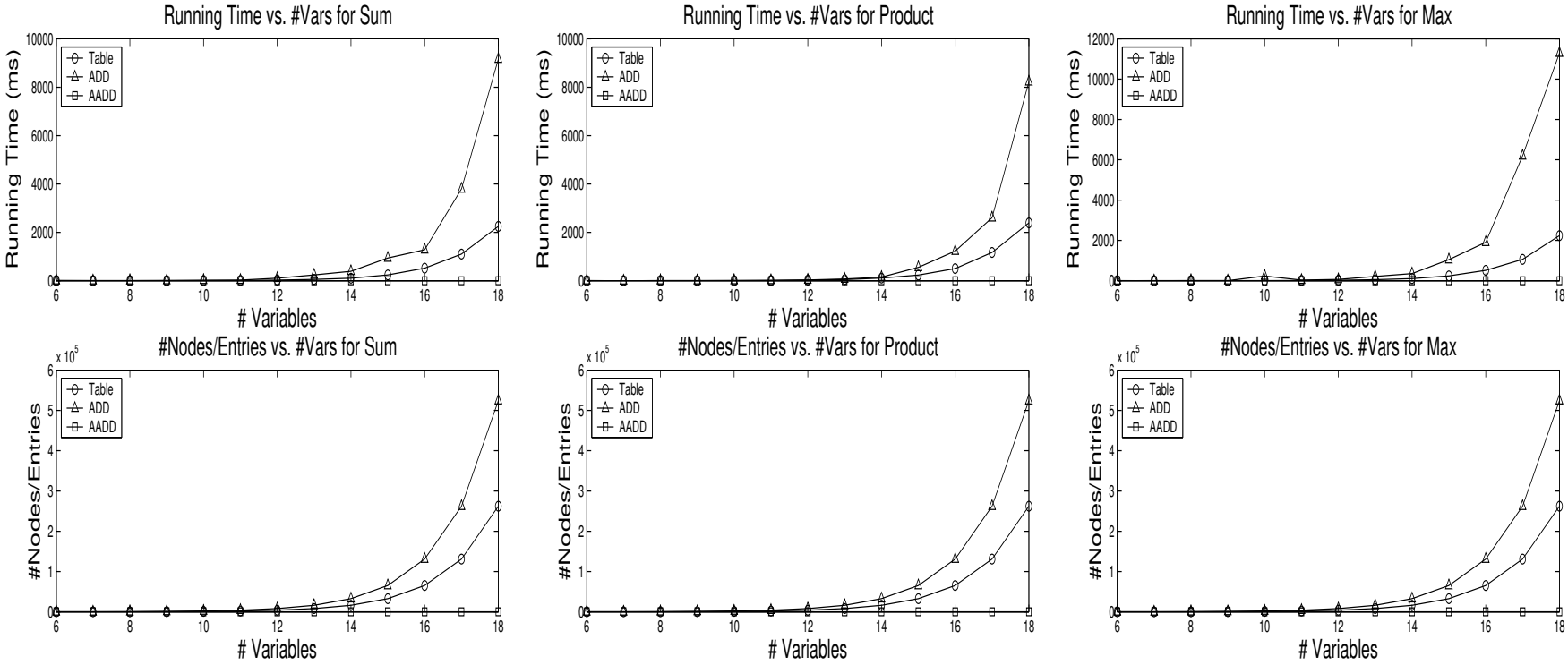


Figure 3.12: Comparison of *Apply* operation running time (top) and table entries/nodes (bottom) for tables, ADDs and AADDs. Left to Right: $(\sum_i 2^i x_i) \oplus (\sum_i 2^i x_i)$, $(\gamma^{\sum_i 2^i x_i}) \otimes (\gamma^{\sum_i 2^i x_i})$, $\max(\sum_i 2^i x_i, \sum_i 2^i x_i)$. Note the linear time/space for AADDs.

Bayes Net	**Table**		**ADD**		**AADD**	
	# Table Entries	Running Time	# ADD Nodes	Running Time	# AADD Nodes	Running Time
Alarm	1,192	2.97 s	689	2.42 s	405	1.26 s
Barley	470,294	*EML**	139,856	*EML**	60,809	207 m
Carpo	636	0.58 s	955	0.57 s	360	0.49 s
Hailfinder	9045	26.4 s	4511	9.6 s	2538	2.7 s
Insurance	2104	278 s	1596	116 s	775	37 s
Noisy-Or-15	65566	27.5 s	125356	50.2 s	1066	0.7 s

Table 3.3: Number of table entries/nodes in the original network and variable elimination running times using tabular, ADD, and AADD representations for inference in various Bayes nets. **EML* denotes that a query exceeded the 1Gb memory limit.

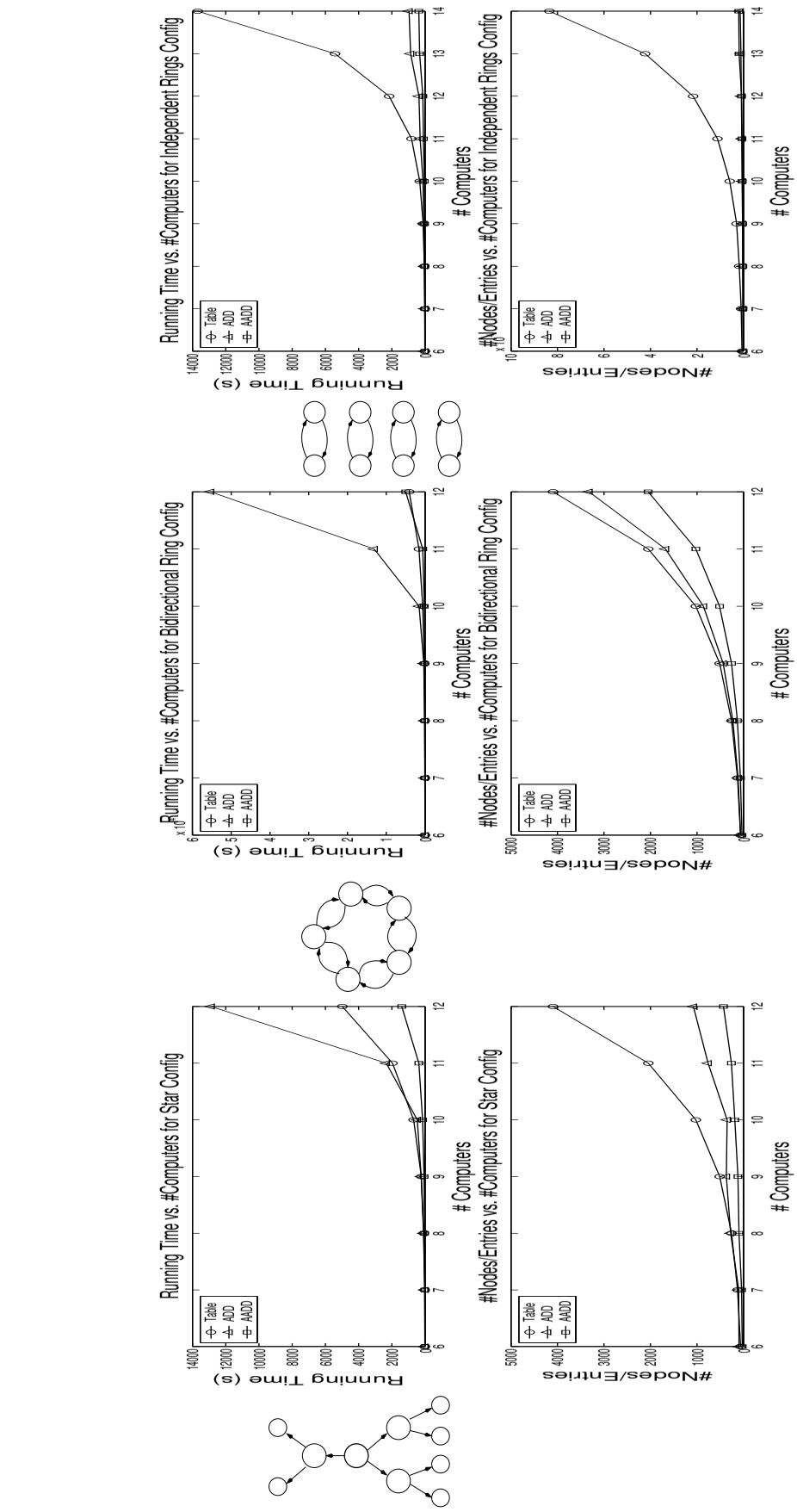


Figure 3.13: MDP value iteration running times (top) and number of entries/nodes (bottom) in the final value function using tabular, ADD, and AADD representations for various network configurations in the SYSADMIN problem.

Note that the intermediate probability tables were too large in one instance for the tables or ADDs, but not the AADDs, indicating that the AADD was able to exploit additive or multiplicative structure in these cases. Also, the AADD appears to yield an exponential to linear reduction on the *Noisy-Or-15* problem by exploiting the multiplicative structure inherent in these special CPTs. While other algorithms have been explicitly designed to exploit noisy-or network structure for efficient inference [Heckerman, 1990], the AADD automatically exploits this structure in standard variable eliminination without explicit modification.

Markov Decision Processes

For MDPs, we simply evaluate the value iteration algorithm using a tabular representation and its extension for decision diagrams as previously discussed for the SPUDD algorithm in exact structured value iteration. We apply these variants of value iteration to factored MDPs from the SYSADMIN domain introduced in Chapter 1 and formalized as a factored MDP in Section 3.1.1 of this chapter. Here we simply substitute tables, ADDs, and AADDs for the reward function, value function, and DBN transition model dynamics in the factored MDP value iteration update of Equation 3.6.

Figure 3.13 shows the relative performance of value iteration until convergence within 0.01 of the optimal value for networks in a star, bidirectional, and independent ring configuration. While the reward and transition dynamics in the SYSADMIN problem have considerable additive structure, we note that the exponential size of the AADD (as for all representations) indicates that little additive structure survives in the *exact* value function. Nonetheless, the AADD-based algorithm still manages to take considerable advantage of the additive structure during computations and thus performs comparably or exponentially better than ADDs and tables for exact value iteration.

Having provided a structured value iteration algorithm for factored MDPs, it is a natural question as to whether we can extend the approximate value iteration ideas to AADDs in the spirit of the APRICODD algorithm previously discussed. We have preliminarily explored this and found that it is much more computationally difficult to prune AADDs in a manner that does not induce unacceptably large approximations. This is because the already compact and distributed nature of the AADD means that every edge typically impacts more states than in the ADD. Nonetheless, this is an interesting area for future research.

3.4.6 Related Work

There has been much related work in the formal verification literature that has attempted to tackle additive and multiplicative structure in representation of functions from $\mathbb{B}^n \rightarrow \mathbb{B}^m$. These include *BMDs [Bryant and Chen, 1995], K*BMDs [R. Drechsler *et al.*, 1997], EVBDDs & FEVBDDs [Tafertshofer and Pedram, 1997], HDDs & *PHDDs [Chen and Bryant, 1997].[17]

However, without covering each data structure in detail, we note there are a few major differences between this related work and AADDs:

- These data structures all originated in the verification community, which means that their terminals are restricted to be vectors of boolean variables, or more generally, integers. When these diagrams can exploit both additive and multiplicative structure, normalization of nodes in these data structures requires prime factorizations of edge weights so there is no direct correspondence between this normalization and AADD normalization (obviously, the prime factorization of a value in $\mathbb{R}$ is ill-defined).

- One could attempt to perform probabilistic inference with integer terminals, thus requiring a rational or direct floating-point representation of values in $\mathbb{R}$. Unfortunately rational representations of terminals require large amounts of space to achieve comparable precision to floating-point representations. And when rational representations are restricted to the same space as floating-point representations, their computation error is much greater than that of a floating-point representation (these reasons are, in fact, the motivation behind floating-point representations). Probabilistic inference applications require manipulating very small values and small numerical approximation errors tend to multiply uncontrollably during marginalization, requiring very precise numerical representations and accurate computations. This can only be reasonably achieved with a floating-point representation.

- *PHDDs are the only decision diagrams that are intended to directly represent floating point numbers and perform standard operations on them since they were created for verification of floating-point arithmetic. However the caveat is that computation with *PHDDs is equivalent to performing all floating-point operations in software. In contrast, AADDs using direct machine floating-point representations and highly accelerated hardware implementations. So, even if *PHDDs could match AADDs in representational

[17] See [Drechsler and Sieling, 2001] for an excellent general overview of most of these decision diagrams.

efficiency (the correspondence if true, is not at all obvious and is an open question), their software-based floating-point computation would slow them down by orders of magnitude in comparison to AADDs.

3.5 Summary and Conclusions

We began this chapter by presenting a factored representation of MDPs that did not require full state enumeration. We then proceeded to describe a large body of recent work that has sought to exploit various forms of factored structure in MDP solution algorithms to likewise avoid explicit state enumeration in those solutions. This work ranges from the use of data structures like ADDs to compactly represent context specific independence in a variety of factored inference algorithms to the use of linear-value function approximation methods to exploit additive structure (and potentially CSI if using appropriate structures for the factor representation) in linear-value approximation solutions to MDPs.

Having done this, we noted that no solution could simultaneously exploit CSI and additive independence in both the reward and transition structure of factored MDPs. To remedy this, we introduced the AADD as a novel data structure that could be plugged into structured value iteration algorithms to exploit CSI, additive independence, *and* multiplicative independence in exact MDP solutions. We have proved that its worst-case time and space performance are within a multiplicative constant of that of ADDs, but can be linear in the number of variables in cases where ADDs are exponential in the number of variables. And we have provided an empirical comparison of tabular, ADD, and AADD representations used in Bayes net and MDP inference algorithms, concluding that AADDs perform at least as well as the other two representations, and can yield an exponential time and space improvement over both when additive or multiplicative structure can be exploited. However, these results are based on a very limited analysis and a more comprehensive investigation is needed in the future to determine in what situations AADDs should be used.

In practice, we note that all of these factored MDP algorithms can be quite efficient in comparison to their enumerated state versions. Nonetheless, there are a number of negative results suggesting that factored MDP representations and factored solution algorithms are not a silver bullet for the curse of dimensionality in decision-theoretic planning problems. As theoretical evidence, Littman et al. [1998] note that finding an optimal plan using tree-structured CPTs is EXP-COMPLETE, and finding an approximately optimal plan using *bounded-size*, tree-structured CPTs is PSPACE-COMPLETE. While these results do not directly apply to

other CPT structures, they are nonetheless discouraging.

However, factored MDP structure is just one type of structure that we can exploit in MDPs. The other type of structure is far more ubiquitous for most of the planning community in AI — that is relational structure. Dating back to the early days of planning when STRIPS representations were introduced [Fikes and Nilsson, 1971], planning problem specifications were inherently relational. And to this day, the predominant planning problem description languages such as ADL [Pednault, 1989], PDDL [McDermott *et al.*, 1998] and its probabilistic variant PPDDL [Younes and Littman, 2004] are still relational. Yet if we cast these problems in a factored MDP framework to solve them, we have to ground our relational representation to propositional variables. But we don't necessarily think of the problem in these ground terms. Clearly there has to be more structure that we can exploit in relational planning problems than just factored structure; we tackle this in the next chapter.

Chapter 4

First-order MDPs

In the last chapter we covered methods for compactly encoding propositionally factored representations of MDPs along with solution algorithms aimed at efficiently exploiting this structure. We originally motivated this propositionally factored structure with the observation that in many decision-theoretic planning problems, we can factor the state representation into independent variables. However, there is much more structure typical in decision-theoretic planning problems that we can exploit in the representation of MDPs and we need look no farther than the STRIPS [Fikes and Nilsson, 1971], ADL [Pednault, 1989], and PDDL family [McDermott *et al.*, 1998; Fox and Long, 2001; Younes and Littman, 2004] of planning description languages to see that all of these languages leverage relational structure for compact representations.

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 methods. This is precisely the idea behind the first-order MDP model and its symbolic dynamic programming solution [Boutilier *et al.*, 2001] that we motivate and review in the first part of this chapter.

The second half of this chapter introduces a simple procedure for generalizing the propositional ADDs and AADDs to first-order (FO) versions that we respectively denote as FOADDs and FOAADDs, or collectively as FO(A)ADDs. We show how these first-order decision diagrams can be used to exploit structure FOMDP solution algorithms in much the same manner that ADDs and AADDs could exploit structure in MDPs. We conclude with a simple set of empirical results that demonstrate that FOADDs combined with a few logical simplification rules prove sufficient to provide an automated solution to basic FOMDPs.

Also in the second half of this chapter, we introduce an additive decomposition approach for approximately solving FOMDPs with universal reward specifications. These approaches are

motivated in part by previous decomposition methods and enable the application of FOMDP solution techniques to a reward specification that otherwise renders standard solution approaches intractable.

4.1 Motivation

Before we introduce FOMDPs and their solution, we begin by introducing the basic notions 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.

4.1.1 Relational Planning Specifications

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, recalling the BOXWORLD problem from Chapter 1 and depicted graphically in Figure 1.1, we have four classes of domain objects: Box, $City$, $Truck$, and $Plane$. And for the relations that hold between them, we have $BoxIn(Box : b, City : c)$, $TruckIn(Truck : t, City : c)$, $BoxOnTruck(Box : b, Truck : t)$, $BoxOnPlane(Box : t, Plane : p)$, $PlaneIn(Plane : p, City : c)$).[1] 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. At its core, PPDDL is a probabilistic extension of a subset of PDDL conforming to the deterministic ADL planning language; ADL, in turn, introduced universal and conditional effects into the STRIPS representation. PPDDL allows for a range of goal-oriented and general reward structure in the spirit of both task and process-oriented planning discussed in the previous chapter.

To see the compactness of a relational representation, we provide a (P)PDDL[2] representa-

[1] For convenience, we restate our notational conventions from Chapter 1: throughout the thesis all predicates (including unary predicates denoting domain object classes) are capitalized and all variables and constants are lowercased. We use the notation $C : v$ to denote that variable v is restricted to domain object class C.

[2] General PPDDL specifications can be more compact for some problems than the PPDDL subset we refer to in this thesis. For example, 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 aspects as we show here [Rintanen, 2003].

- **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**:
  - $load(Box : b, Truck : t)$:
    - Aspect 1 (probability 0.9):
      - $when\ [\exists City : c.\ BoxIn(b, c) \wedge TruckIn(t, c)]\ then\ [BoxOn(b, t)]$
      - $\forall City : c.\ when\ [BoxIn(b, c) \wedge TruckIn(t, c)]\ then\ [\neg BoxIn(b, c)]\}$
  - $unload(Box : b, Truck : t)$:
    - Aspect 1 (probability 0.9):
      - $\forall City : c.\ when\ [BoxOn(b, t) \wedge TruckIn(t, c)]\ then\ [BoxIn(b, c)]$
      - $when\ [\exists City : c.\ BoxOn(b, t) \wedge TruckIn(t, c)]\ then\ [\neg BoxOn(b, t)]$
  - $drive(Truck : t, City : c)$:
    - Aspect 1 (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.

Figure 4.1: A PPDDL-style representation of a simple variant of the BOXWORLD problem. The deterministic PDDL subset would exclude the probabilistic aspects assuming that all effects occur with probability 1.0.

tion of the BOXWORLD problem in Figure 4.1 where for simplicity, we omit the $Plane$ class of objects and associated actions and relations and shorten the $BoxOnTruck(Box : b, Truck : t)$ relation to $BoxOn(Box : b, Truck : t)$.

While the meaning of the PPDDL representation in Figure 4.1 is intended to be relatively straightforward, there are a few important dimensions of the specification that should be explained. First, we assume that actions can execute in all states so we do not encode explicit preconditions.[3] When an action executes, it may have a number of different aspects encapsulating a joint set of effects, where each aspect is realized independently according to the specified probability. For example, the $unload$ action realizes its first aspect only 90% of the time it is

[3]While this assumption is not necessary, it does not have any effect on the optimal policy in a domain that already has a $noop$ action. It also simplifies (1) the amount of notation in our presentation, (2) the translation from PPDDL to the stochastic situation calculus and (3) the proofs of correctness for FOMDPs.

executed whereas the $drive$ action deterministically realizes its first aspect on each execution.

Aspects themselves consist of conjunctions of effects. Each individual effect can be universal and conditional. Universal effects allow the inclusion of universal quantifiers that permit the effect to apply to an arbitrary number of objects not explicitly named in the action's parameter list. If a universal effect applies only to the objects explicitly named in the action's parameter list, then we refer to it as a *local* universal effect. Universal effects are usually combined with conditional effects denoted by the $when$/$then$ clause pair that specify that the $then$ effects occur in the post-action state if the $when$ conditions hold for the pre-action state.[4] For example, when the $load(b, t)$ action is executed, the first aspect is realized with 90% probability. When this set is realized, then for any city c that satisfies $BoxIn(b, c) \wedge TruckIn(t, c)$ in the pre-action state, $BoxOn(b, t) \wedge \neg BoxIn(b, c)$ will be asserted in the post-action state since both aspects have equivalent $when$ conditions. When this aspect is 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 4.1 to the ground factored MDP representation in Figure 4.2 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 one 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 its 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 superlinearly in the number of domain objects that fill each relation slot (assuming the maximum arity is greater than one). To see this, let us assume for simplicity that all object class instantations have k instances. Then a single unary relation would be rep-

[4] 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.

- **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 instantion of PPDDL actions in Figure 4.1.
- **($\mathcal{R}$)eward**:
  $if\ [BoxIn(box_1, paris) \vee BoxIn(box_2, paris) \vee BoxIn(box_3, paris)]\ then\ 10\ else\ 0$

Figure 4.2: One possible ground MDP instantiation of the BOXWORLD FOMDP.

resented 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 will scale exponentially in the number of nested quantifiers, linearly in the number of relations being quantified, and exponentially 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 grounded representation would lead to rk^q ground atoms. Thus, the number of state variables and the number (if not the size) of the factors in the factored MDP

representation will scale polynomially in the domain size with an order determined by the maximum arity of relations and the maximum number of nested quantifiers in any formula — at least linearly in the best case.[5]

Nonetheless, if we have adequate space to permit the grounding of a relational MDP w.r.t. a domain instantiation to obtain a factored MDP *and* we have the time to find an (approximately) optimal solution to this factored MDP, then grounding gives us one approach to representing and solving relational MDPs for specific domain instances.

4.1.2 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 4.1 remains constant. Consequently, this invites the following question: if we can avoid a domain-instance dependent blowup in the representation of a relational MDP such as in PPDDL, can we avoid a domain-instance dependent blowup in its solution too?

Although we have yet to discuss the specifics of the first-order MDP representation, in Figure 4.3 we provide an optimal domain-instance independent value function and its corresponding policy for the relational PPDDL specification of the BOXWORLD problem in Figure 4.1 (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 a first-order logic representation.

Thus, an alternative idea to solving a FOMDP at the ground level — and an idea that is central to this chapter and the rest of thesis — is to convert the PPDDL relational specification to a lifted first-order MDP representation and solve this first-order MDP directly at the first-order level using purely symbolic methods. This approach obtains a solution that applies universally

[5]If the domain trivially has one unary relation, then the number of ground atoms would simply be the number of domain objects filling that relation.

- 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)

Figure 4.3: 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 to take is also shown for each start partition where the optimal bindings of the action variables (denoted by a *) correspond to any binding satisfying those variable names in the state formula.

to all possible domain instantiations without scaling in a manner related to any specific domain instantiation. 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 directly without resorting to grounding.

4.2 Situation Calculus Background

Before we present the first-order MDP (FOMDP) formalism, we must provide the foundations for the situation calculus that provides the logical foundation for FOMDPs. We assume a basic knowledge of sorted first-order logic and refer the reader to Reiter [2001] and Brachman and Levesque [2004] for an overview of first-order logic concepts relevant to the material presented here.

We do make two additional notes w.r.t. our presentation:

- Previously we have specified the sorts of variables explicitly, for example $\forall City : c\, \phi(c)$. This notation can be easily converted to standard unsorted first-order logic for use with many popular theorem provers, for example, $\forall City : c\, \phi(c)$ can be rewritten as $\forall City(c) \implies \phi(c)$ and likewise $\exists City : c\, \phi(c)$ can be rewritten as $\exists City(c) \wedge \phi(c)$. We omit explicit sort specifications on quantifiers when they can be inferred from context, for

example, from the sort specification of relation slots or via transitivity of equality tests.

- Throughout this background review, we note that the situation calculus is *deterministic* and thus we will be temporarily assuming a deterministic representation conforming to the PDDL subset of the PPDDL specification for BOXWORLD in Figure 4.1. As noted previously, this is equivalent to assuming that all of the action effects in Figure 4.1 occur with probability 1.0.

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 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.

4.2.1 Basic Ingredients

The situation calculus is a first-order language for axiomatizing dynamic worlds [McCarthy, 1963]. It's 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 would be represented by $load(b, t)$.

- *Situations*: A situation is a first order term denoting a specific state. The initial situation is usually denoted as s_0 and subsequent situations resulting from action executions are obtained from the $do(a, s)$ function that represents the situation resulting from the execution of action a in state 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, let us imagine that we are given an initial state s_0 such that the fluent $BoxOn(b, t, s_0)$ is false,

but the 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), $BoxOn(b, t, do(load(b, t), s_0))$ will hold true. We do not consider functional fluents in this exposition, but they could be easily added to the language without adverse computational side effects [Reiter, 2001].

4.2.2 From PDDL to a First-order Domain Theory

Now that we've defined the basics of the situation calculus, we need to describe how one may go about axiomatizing a domain theory. In order to do so, 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.[6]

- *Positive Effect Axioms*: Following is a set of the positive effect axioms stating which actions can explicitly make each fluent true:

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

- *Negative Effect Axioms*: Following is a set of the negative effect axioms stating which actions can explicitly make each fluent false:

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

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 4.1. In fact, one can verify that these effect axioms are simply syntactic rewrites of the PDDL effects where we have made the following transformations:

[6] 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.

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*.

Without covering the various proposals for solutions to the *Frame Problem*, we jump directly to Reiter's [Reiter, 1991] default solution. In this solution, one must specify all positive and negative effects for a fluent, which conveniently, we have just done in our translation from PDDL to the positive and negative effect axioms.

We use the following normal form for positive effect axioms:

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)) \tag{4.1}$$

And we use the following normal form for negative effect axioms:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)) \tag{4.2}$$

We note here that the potential difference between are previous presentation of positive and negative effect axioms and this normal form is 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

$$[(C_1 \supset F) \wedge (C_2 \supset F)] \equiv [(C_1 \vee C_2) \supset F], \tag{4.3}$$

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

Finally, we need to add in unique name axioms for actions that specify for distinct action names A and B:

$$A(\vec{x}) \neq B(\vec{y}) \tag{4.4}$$

and also that identical actions have identical arguments:

$$A(x_1, \ldots, x_k) = A(y_1, \ldots, y_k) \supset x_1 = y_1 \wedge \ldots \wedge x_k = y_k \tag{4.5}$$

From this normal form, unique names axioms, and additional explanation closure axioms that state that 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:

$$\begin{aligned} F(\vec{x}, do(a, s)) &\equiv \Phi_F(\vec{x}, a, s) \\ &\equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \end{aligned} \tag{4.6}$$

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

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

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

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

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.

4.2.3 Regression

An important tool in the development of first-order MDPs will be 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 definition of regression; specifically, 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 now perform regression on any first-order logic formula. For example, if we know the formula

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

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

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

At this point, we can use the unique names axioms for actions to simplify and we can additionally exploit rules for distributing quantifiers and renaming variables w.r.t. 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 now make the assumption that all object domains are non-empty, which is an assumption we will make throughout the thesis.[7] Thus we can obtain the following fully simplified form of the regression:

$$\begin{aligned}
&Regr(\exists b.\, BoxIn(b, paris, do(unload(b^*, t^*), s))) \\
&\qquad = [BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] \vee \exists b.\, BoxIn(b, paris, s)
\end{aligned} \tag{4.7}$$

[7] Specifically, we need a background theory for every object class $Class$ that states $\exists o.\ Class(o)$ in order to use the simplification $(\exists Class : o.\ o = o^*) \longrightarrow \top$.

And this final result is very intuitive. Effectively 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.

4.3 FOMDP Representation

A first-order MDP (FOMDP) [Boutilier *et al.*, 2001] 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 section 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 of these components are defined, we will have everything needed to generalize the dynamic programming solution from the ground case in previous chapters to the lifted case of symbolic dynamic programming for FOMDPs.

4.3.1 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:

$$(t \;=\; case[\phi_1, t_1; \cdots; \phi_n, t_n]) \;\equiv\; \left(t \;=\; \begin{array}{|c|} \hline \phi_1 : t_1 \\ \hline \vdots \quad \vdots \\ \hline \phi_n : t_n \\ \hline \end{array} \right)$$
$$\equiv \left(\bigvee_{i \le n} \{ \phi_i \wedge t = t_i \} \right) \tag{4.8}$$

Here the ϕ_i are *state formulae* where fluents in these formulae do not contain the term do[8] and the t_i are terms. 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,

[8] 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.

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 pseudo-function (not necessarily assigning a distinct value to all elements of its domain) and thus cases may be compared with inequalities and manipulated with arithmetic operations to produce other case statements (all at a meta-logical level). In this sense, we often omit the $t =$ prefix from the case syntax and manipulate $case[\cdot]$ or its tabular representation as if it were a function; we note that without the $t =$ prefix, a case statement is not a logical formula, but rather a protological statement.

We use $case_1[\cdot] = case_2[\cdot]$ to mean two things. When we say $R(s) = rCase(s)$, it is a metalogical notation for $R(s) =$ "some specific case statement" (sometimes $rCase(s)$ is used, sometimes a tabular representation, and sometimes an explicit logical formula is used). The same follows for value functions $V(s) = vCase(s)$, Q-functions $Q(s,a) = qCase(s,a)$, transition functions $P(\cdot,\cdot,s) = pCase(\cdot,\cdot,s)$, basis functions $b_i(s) = bCase_i(s)$, and policies that we will define later. At many other points in the text, we will use $case_1[\cdot] = case_2[\cdot]$ in sequences of equational rewrites. The formal definition of such equality rewrites is $t = case_1[\cdot] \equiv t = case_2[\cdot]$ (for a new variable t).

In the forthcoming text, we will define first-order decision-theoretic regression $FODTR[\cdot]$ and backup operators $B^A[\cdot]$ as algorithmic/mathematical operations on case statements that produce a new case statement. These operations can be interpreted as applying to real functions (like value functions) to produce new functions, or their logical/protological representations. We abuse notation and occasionally use both representations.

To provide an example of the first usage of equality discussed above, we represent our BOXWORLD FOMDP reward function $R(s)$ from our PPDDL representation in Figure 4.1 as the following $rCase(s)$ statement that reflects the immediate reward obtained in situation s:[9]

$$rCase(s) = \begin{array}{|l|} \hline \exists b.BoxIn(b, paris, s) \quad : 10 \\ \hline \neg\exists b.BoxIn(b, paris, s) : \quad 0 \\ \hline \end{array} \tag{4.9}$$

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. Thus, for BOXWORLD, we write $R(s) = rCase(s)$ and wherever $R(s)$ occurs, we can syntactically substitute the specific tabular case statement on the RHS of Equation 4.9 above.

[9]For simplicity of presentation, we will assume the reward is not action dependent, but such dependences can be introduced without difficulty, if needed.

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 manner. Indeed, $V^0(s) = R(s)$ in the first-order version of value iteration.

The case representation can also be used to define probabilities. We will see an instance of such a usage when we define the transition function for stochastic actions. Before we do this, however, let us first discuss the operations that can be performed on case statements.

4.3.2 Case Operations

In this section we introduce various unary, binary, n-ary operations that can applied to case statements. We begin by introducing a formal logical definition that can be used in proofs and then proceed to give a graphical example that intuitively demonstrates the case operation being applied.

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

$$case[\phi_i, t_i : i \leq n] \otimes case[\phi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i \cdot v_j : i \leq n, j \leq m] \quad (4.10)$$

$$case[\phi_i, t_i : i \leq n] \oplus case[\phi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i + v_j : i \leq n, j \leq m] \quad (4.11)$$

$$case[\phi_i, t_i : i \leq n] \ominus case[\phi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i - v_j : i \leq n, j \leq m] \quad (4.12)$$

Intuitively, to perform an operation on case statements, we simply perform the corresponding operation on the intersection 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 (as opposed to adding them) to obtain the result. Some partitions resulting from the application of the $\oplus$, $\ominus$, and $\otimes$ operators may be inconsistent; we simply discard such partitions (since they can obviously never correspond to any world state).

We use the $\bigoplus$ and $\bigotimes$ operators to respectively denote summations and products of multiple case operands.

We define a few additional operations on case statements, the first is the binary $\cup$ operation for which we provide a formal definition:

$$case[\phi_i, t_i : i \leq n] \cup case[\psi_j, v_j : j \leq m] = case[\phi_1, t_1; \cdots; \phi_n, t_n; \psi_1, v_1; \cdots; \psi_m, v_m] \tag{4.13}$$

In this operation we simply union together the partitions from each of the case statements. Following is an 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 push the existential quantifier through to each individual case partition:

$$\begin{aligned} \exists \vec{x}. \begin{array}{|c|}\hline \phi_1(\vec{x}) : t_1 \\ \hline : \quad : \ : \\ \hline \phi_n(\vec{x}) : t_n \\ \hline \end{array} &= \exists \vec{x}.\ \bigvee_{i \leq n} \{\phi_i(\vec{x}) \wedge t = t_i\} \\ &= \bigvee_{i \leq n} \{\exists \vec{x}.\ \phi_i(\vec{x}) \wedge t = t_i\} \\ &= \begin{array}{|c|}\hline \exists \vec{x}.\ \phi_1(\vec{x}) : t_1 \\ \hline : \quad : \ : \\ \hline \exists \vec{x}.\ \phi_n(\vec{x}) : t_n \\ \hline \end{array} \end{aligned} \tag{4.14}$$

The second unary operation is a unary maximization that we denote "casemax" 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:

$$\text{casemax } case[\phi_1, t_1; \cdots; \phi_n, t_n] \;=\; case[\phi_i \wedge \bigwedge_{j \le i} \neg\phi_j, v_i : i \le n] \tag{4.15}$$

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

$$\text{casemax} \begin{array}{|l|} \hline \psi_1 : t_1 \\ \hline \psi_2 : t_2 \\ \hline \vdots \;\; : \;\; \vdots \\ \hline \psi_n : t_n \\ \hline \end{array} = \begin{array}{|l|} \hline \psi_1 \qquad\qquad\qquad\qquad\qquad\qquad : t_1 \\ \hline \psi_2 \wedge \neg\psi_1 \qquad\qquad\qquad\qquad : t_2 \\ \hline \vdots \qquad\qquad\qquad\qquad\qquad\qquad\;\; : \;\; \vdots \\ \hline \psi_n \wedge \neg\psi_1 \wedge \neg\psi_2 \wedge \cdots \wedge \neg\psi_{n-1} : t_n \\ \hline \end{array}$$

One can easily verify that if assuming sorting of partitions in order of highest (top) to lowest (bottom) value then the highest value is assigned to each partition by rendering lower value partitions disjoint from their higher-value antecedents.

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 a comparison function. Consequently, we will assume for the rest of this chapter that the case values are numeric rather than symbolic in order to use the casemax operator. However, we will relax this assumption to accommodate general symbolic value representations in Chapter 6.

4.3.3 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 choice deterministic actions $n_1(\vec{x}), \cdots, n_k(\vec{x})$, we represent the distribution over $n_i(\vec{x})$ given $A(\vec{x})$ 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 this aspect, and we let the deterministic action $loadF(b,t)$ denote the failure of these effects to execute. In order to do this, we must redefine our SSAs from the previous PDDL case, where now $load(b,t)$ will be a stochastic action executed by the agent with $loadS(b,t)$ and $loadF(b,t)$ being possible deterministic outcomes of selecting this action. In fact, we will do similarly for 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:

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

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

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

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.[10]

We can now specify a distribution over Nature's choice deterministic outcome for each stochastic action where we denote a specific instance of $P(n_j(\vec{x}), A(\vec{x}), s)$ with the case state-

[10]Since we intend the failure versions of the actions to represent the "no effects" case, they obviously do not play any role in the SSAs. The frame assumption implicit in the SSAs will ensure that what was not explicitly changed will remain the same.

ment $pCase(n_j(\vec{x}), A(\vec{x}), s)$:

$$pCase(loadS(b,t), load(b,t), s) = \boxed{\top : 0.9} \tag{4.16}$$

$$pCase(loadF(b,t), load(b,t), s) = \boxed{\top : 0.1} \tag{4.17}$$

$$pCase(unloadS(b,t), unload(b,t), s) = \boxed{\top : 0.9} \tag{4.18}$$

$$pCase(unloadF(b,t), unload(b,t), s) = \boxed{\top : 0.1} \tag{4.19}$$

$$pCase(driveS(b,t), drive(b,t), s) = \boxed{\top : 1.0} \tag{4.20}$$

$$pCase(driveF(b,t), drive(b,t), s) = \boxed{\top : 0.0} \tag{4.21}$$

Since the above axiomatization does not fully illustrate the power of the FOMDP representation in that the probabilities are not state dependent, we digress for a moment 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 a distribution over Nature's choice deterministic outcome for this stochastic action:

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

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

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

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

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

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 verified easily and we verify that the first

property holds as follows:

$$pCase(\ driveS(t,c), drive(t,c), s\ ) \oplus pCase(\ driveF(t,c), drive(t,c), s\ )$$

$$= \begin{array}{|l:r|}\hline \exists b.BoxOn(b,t,s) \wedge Volatile(b) & : 0.9 \\ \hline \neg(\exists b.BoxOn(b,t,s) \wedge Volatile(b)) & : 1.0 \\ \hline \end{array} \oplus \begin{array}{|l:r|}\hline \exists b.BoxOn(b,t,s) \wedge Volatile(b) & : 0.1 \\ \hline \neg(\exists b.BoxOn(b,t,s) \wedge Volatile(b)) & : 0.0 \\ \hline \end{array}$$

$$= \boxed{\top : 1}.$$

4.4 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 of SDP leads directly to a domain-independent value iteration solution to FOMDPs.

Although we will assume a constant numerical representation of values in order to explicitly perform the casemax during SDP in this chapter (see the previous discussion), an appropriate generalization of casemax allows the definitions covered here to apply to general symbolic value representations, hence the original use of "symbolic" in the name of the SDP algorithm. We will demonstrate SDP applied to some symbolic value extensions in Chapter 6.

4.4.1 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 is a case statement representing a Q-function.

We define the *first-order decision theoretic regression (FODTR)* operator in the following

manner:[11]

$$FODTR[V(s), A(\vec{x})] = R(s) \oplus \gamma \left[\bigoplus_{j=1}^{k} \{ P(n_j(\vec{x}), A(\vec{x}), s) \otimes Regr(V(do(n_j(\vec{x}), s))) \} \right] \quad (4.22)$$

This is a meta-logical notation where $FODTR$ takes as arguments $V(s)$ representing the case statement for a value function with situation variable s and a parameterized stochastic action term $A(\vec{x})$ with free variables $\vec{x}$. Because the meta-logical notation of $V(s)$ refers to variable s in the underlying logical language, we can make a logical substitution of terms such as $do(n_j(\vec{x}), s)$ for s (standardizing apart usage of the variable s before substitution) to obtain $V(do(n_j(\vec{x}), s))$ where we apply this same substitution directly to the (proto)logical case representation of $V(s)$. The result of applying $FODTR$ is a case statement. All subsequently defined operations on case statements in this thesis will be defined analogously.

As opposed to the deterministic regression operator for first-order formulae, FODTR takes into account a decision-theoretic expectation over the regression of Nature's choice deterministic actions w.r.t. the utilities of the possible post-action formulae as specified by $V(s)$. From here out, we will denote an instance of the value function $V(s)$ by the case statement $vCase(s)$. And as defined previously, we also assume that instances of Nature's choice action probabilities $P(n_j(\vec{x}), A(\vec{x}), s)$ and reward function $R(s)$ are denoted respectively by $pCase(n_j(\vec{x}), A(\vec{x}), s)$ and $rCase(s)$.

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 4.9. Since $vCase(s)$ is logically defined, we can push the $Regr$ operator into the individual $vCase(s)$ partitions as follows:

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

Now, since the stochastic action $A(\vec{x}) = unload(b^*, t^*)$, we know that Nature's determin-

[11] If the reward were action dependent here, then we would simply replace $R(s)$ with $R(s, A(\vec{x}))$.

istic 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. 4.18 and 4.19, respectively.

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

We note that we have already computed $Regr(\exists b. BoxIn(b, paris, do(unloadS(b^*, t^*))))$ from Equation 4.7 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 negate Equation 4.7 to obtain $Regr(\neg\exists b. BoxIn(b, paris, do(unloadS(b^*, t^*))))$. It is important to note that just as $rCase(s)$ partitioned the state space, the $Regr$ operator preserves this partitioning in $Regr(rCase(\cdot))$. This result follows directly from the properties of $Regr$ and can be easily verified for the two partition case of ϕ and $\neg\phi$. We note that

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

since $unloadF(b^*, t^*)$ has no effects and this is equivalent to a $noop$ action. Making these substitutions, explicitly multiplying in the action probabilities and $\gamma = 0.9$, 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:

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

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

$$FODTR[vCase(s), unload(b^*, t^*)] = \tag{4.23}$$

$$= \begin{array}{|l|r|} \hline \exists b.\, BoxIn(b, paris, s) & : 19.0 \\ \hline \neg \text{“} \wedge [\exists c.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : \ 8.1 \\ \hline \neg \text{“} & : \quad 0 \\ \hline \end{array} \tag{4.24}$$

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 reward total.

It is important to note that the case statement resulting from FODTR contains free variables for the action parameters $\vec{x}$ in $A(\vec{x})$. In this case, the action $A(\vec{x}) = unload(b^*, t^*)$ so the parameters were b^* and t^* and we note that these do, in fact, occur in the final result.

This case statement represents the value of taking a specific 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 [12] 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:

$$Q^t(A, s) = \exists \vec{x}.\, FODTR[V^{t-1}(s), A(\vec{x})] \tag{4.25}$$

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 $A(\vec{x})$. And 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 the follow-

[12] Recall Equation 2.8 from Chapter 2 for the enumerated state version of the Q-function.

ing Q-function description for action $unload$ where we have removed vacuous quantifiers:

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

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

This gives us a very intuitive result that 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.

Technically, $qCase^1(unload, s)$ would not be an exhaustive partitioning of the state space in that the 0 value partition from Equation 4.25 is not the same one implied here from the $\neg$“ because the partition formulae above it have been quantified. However, throughout this thesis, we can exploit our assumption that all FOMDPs have a $noop$ action to assume that the minimum value for any state is 0. Thus we will always show the final 0 partition as $\neg$“ (thus forcing the partitioning to be exhaustive by filling in any state space not covered by other partitions with a 0 value) even when this does not directly follow from the logical operation being applied.

We additionally remark that $qCase^1(unload, s)$ may no longer have mutually exclusive partitions due to the existential quantification: the formulae pair $A(x)$ and $\neg A(x))$ are mutually exclusive, but the formulae pair $\exists x, y.A(x, y)$ and $\exists x, y.\neg A(x, y)$ are not mutually exclusive. The resulting ambiguity of value assignments occurring after existential quantification will be resolved by assigning every state its maximal value as discussed next.

4.4.2 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 step in the spirit of Equation 2.9 from Chapter 2, we need to know the maximum value that can be achieved by *any* action. For example, in the BOXWORLD FOMDP, our possible ac-

tion choices are $unload(b,t)$, $load(b,t)$, and $drive(t,c)$ and our Q-function computations using Equation 4.25 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 $vCase^{t-1}(s)$.

One way to obtain a case description of the value function $vCase^t(s)$ would simply be to apply 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 does, in fact, give us a description of the value function $vCase^t(s)$, the caveat is that the state spaces of each Q-function overlap with each other and thus the union of case partitions from each Q-function typically assigns multiple values to overlapping regions of state space. 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:

$$V^t(s) = \text{ casemax } \left[Q^t(A_1,s)\cup\ldots\cup Q^t(A_m,s)\right] \tag{4.26}$$

Recalling the way in which the casemax operation is computed from Equation 4.15, 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 $v_i > v_{i+1}$:

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

This approach effectively gives us a decision-list representation of our value function[13] — 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).

[13] Recall the optimal value function representation from Figure 4.3.

4.4.3 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 2.8 and 2.9 from Chapter 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 the $V^t(s)$ from $V^{t-1}(s)$ using Eqs. 4.25 and 4.26.
3. If the following is not true

$$\|V^t(s) \ominus V^{t-1}(s)\|_\infty \leq \frac{\epsilon(1-\gamma)}{2\gamma}, \tag{4.27}$$

   then go to step 2, otherwise terminate.

For example, applying first-order value iteration to the 0-stage-to-go value function (i.e., $vCase^0(s) = rCase(s)$, given previously) yields the following simplified 1- and 2-stage-to-go value functions in the BOXWORLD 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 corresponding policy) from Fig. 4.3.

Having presented the value iteration algorithm, we may now wish to prove some properties of it. 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. However, they do not provide an explicit correspondence between FOMDPs formalized with the deterministic situation calculus and MDPs as formalized in Chapter 2 with explicit stochastic actions. While the correspondence is not difficult to show, it is nonetheless useful to make this explicit. Thus, we provide a

direct correspondence between FOMDPs and MDPs in Appendix A.7.1 and provide an alternate proof of correctness of first-order value iteration based on this correspondence:

Theorem 4.4.1. *Given a correspondence between a FOMDP and a ground MDP obtained from the FOMDP for* any *domain instantiation, the value function $vCase^t(s)$ computed by first-order value iteration corresponds to the value function $V^t(s)$ computed by enumerated state value iteration.*

Proof. Follows directly from Theorem A.7.1. Refer to Appendix A for complete definitions and a proof of Theorem A.7.1.

From this theorem, we get the following corollary:

Corollary 4.4.2. *Terminating according to the criteria given in Step 3 of first-order value iteration guarantees that $vCase^t(s)$ is an ϵ-optimal value function for* any *domain instantation.*

Proof. The proof of this corollary follows directly from the correspondence given in Theorem 4.4.1. That is, we know from Puterman [1994] that this property holds for all ground MDPs, and this theorem tells us there is a direct correspondence between a FOMDP and all possible ground MDP instantiations of that FOMDP. Therefore the corollary trivially follows. □

4.4.4 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 Chapter 2 and the factored case in Chapter 3. 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.

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 $\cdot$ from which the policy is derived as follows (taking into account a few modifications to the case operators that we discuss in a moment):

$$\pi(s)[V(s)] = \text{casemax}\Big(\bigcup_{i=1\ldots m} \exists \vec{x}.\, FODTR[V(s), A_i(\vec{x})]\Big) \tag{4.28}$$

We denote a specific instance of $\pi(s)$ by a modified case statement representation $\pi Case(s)$ that we describe here. For bookkeeping purposes, 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 4.24 and that $FODTR[rCase(s), noop]$ will just be $rCase(s)$ with 10 replaced by 19, we easily obtain the following policy:[14]

$$\begin{aligned}
&\pi Case[rCase(s)] \\
&\quad = \text{casemax}(\{\exists b^*, t^*.\ FODTR[rCase(s), unload(b^*, t^*)]\} \\
&\qquad\qquad \cup \{FODTR[rCase(s), noop]\}) \\
&\quad = \begin{array}{|l|l|}
\hline
\exists b.\ BoxIn(b, paris, do(a, s)) & : 19.0 \longrightarrow noop \\
\hline
\neg \text{“} \wedge [\exists b^*, t^*, c.\ BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] & : \ \ 8.1 \longrightarrow unload(b^*, t^*) \\
\hline
\neg \text{“} & : \ \ \ 0 \longrightarrow noop \\
\hline
\end{array}
\end{aligned}$$

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

For a more interesting policy, we refer the reader back to the optimal value function and policy for BOXWORLD given in Figure 4.3.

4.5 Comments on Policy Iteration and Linear Programming

We do not provide first-order policy iteration or linear programming algorithms for FOMDPs, although it is possible to define a first-order modified policy iteration algorithm as we discuss shortly. As a non-trivial consequence of the extension of MDPs to first-order MDPs, the num-

[14] This is in fact the optimal policy for BOXWORLD using $vCase(s) = rCase(s)$, but one has to grind through all the FODTR applications for $load(b, t)$ and $drive(t, c)$ and simplifications to show this. Here, we restrict the action space to demonstrate a more obvious result.

ber of distinct values in the *exact* value function can be infinite. This is problematic because our current piecewise-constant case representation of the value function can only make a finite number of value distinctions given finite space. Since representing an infinitely sized value function would be impossible in the current case representation this precludes two computations under our current representation: (1) the exact computation of the value for a first-order policy in a FOMDP using a linear system in the spirit of Equation 2.11 from Chapter 2; and (2) the exact linear programming solution of a FOMDP in the spirit of Equation 2.24 from Chapter 2.

In the case of first-order policy iteration, it is possible to define a first-order modified policy iteration algorithm by generalizing successive approximation methods for value determination under a policy to the first-order case. This is straightforward as we need only replace the symbolic maximization over all actions in SDP with the actual policy being executed analogously to that done for successive approximation in the factored MDP case (c.f., Equation 3.6 of Chapter 3). We present a method for computing SDP under policy restrictions in the next chapter that could be used in a modified policy iteration algorithm. This would effectively be the first-order generalization of the structure policy iteration (SPI) algorithm from Chapter 3. For the restricted case of a first-order logic with only existential quantifiers, Wang *et al.* [2007] provide a modified policy iteration algorithm with special data structures to handle this restricted logic.

Generalizing the exact linear programming approach is a bit more difficult in that it requires the exact value function structure beforehand and there is no obvious workaround for this in the exact case. Since the value function can be infinite in our piecewise-linear case representation, this precludes the possibility of exact linear programming as a general solution to all FOMDPs using our current representation. On the other hand, if we have some clue as to what an approximate value function structure may look like then we can relax our requirements to an approximate solution and leverage first-order generalizations of the approximate linear programming approaches that we used in Chapters 2 and 3. We will present just such an approach in the next chapter.

4.6 Representation and Solution with First-order (A)ADDs

Just as we previously exploited CSI in the factored propositional representation of MDPs, we can easily generalize this technique to exploit CSI in the case representation of first-order MDPs. In the first-order framework, we can define methods for breaking down first-order case partition formula into their propositional components and create a *first-order ADD (FOADD)* or

first-order AADD (FOAADD) representation of the case statement. Then we can apply standard ADD or AADD operations to perform the $\otimes$, $\oplus$, and $\ominus$ case operations. We can also define extensions of the casemax, $\exists \vec{x}$, and $Regr$ operators capable of exploiting FO(A)ADD structure. After discussing each of these topics in turn, we end with a discussion of the practical use of FO(A)ADDs, a small example of a FOADD application to SDP and a discussion of related approaches.

4.6.1 Constructing FO(A)ADDs from a Case Representation

The first aspect of FO(A)ADDs concerns how to construct them automatically from a logical case representation. The key to our approach to FOADDs is that their decision tests are propositional in nature so we need some method of finding propositional structure in first-order formulae. We can do this by distributing quantifiers as deeply into case formulae as possible using the following rewrite rule templates where $\diamond$ indicates variables other than those explicitly quantified:

- $[\exists x, y.\, \phi] \longrightarrow [\exists y, x\, \phi]$
- $[\forall x, y.\, \phi] \longrightarrow [\forall y, x\, \phi]$
- $[\exists x.\, A(x, \diamond) \vee B(x, \diamond)] \longrightarrow [(\exists x.\, A(x, \diamond)) \vee (\exists x.\, B(x, \diamond))]$
- $[\forall x.\, A(x, \diamond) \wedge B(x, \diamond)] \longrightarrow [(\forall x.\, A(x, \diamond)) \wedge (\forall x.\, B(x, \diamond))]$
- $[\exists x.\, A(x, \diamond) \wedge B(y, \diamond)] \longrightarrow [(\exists x.\, A(x, \diamond)) \wedge (B(y, \diamond))]$
- $[\forall x.\, A(x, \diamond) \vee B(y, \diamond)] \longrightarrow [(\forall x.\, A(x, \diamond)) \vee (B(y, \diamond))]$

One can see an example application of these rewrite rules in Figure 4.4(a,b) where the formula

$$\exists x.[A(x) \vee \forall y.A(x) \wedge B(x) \wedge \neg A(y) \tag{4.29}$$

has been rewritten to the equivalent form

$$[\exists x.A(x)] \vee ([\exists x.A(x) \wedge B(x)] \wedge [\forall y.\neg A(y)]) \tag{4.30}$$

where quantifiers have been distributed into the nested formula structure as far as possible.

Once we have pushed quantifiers as far down as possible, we now want to extract the propositional structure of the formula by considering propositional connectives over quantified

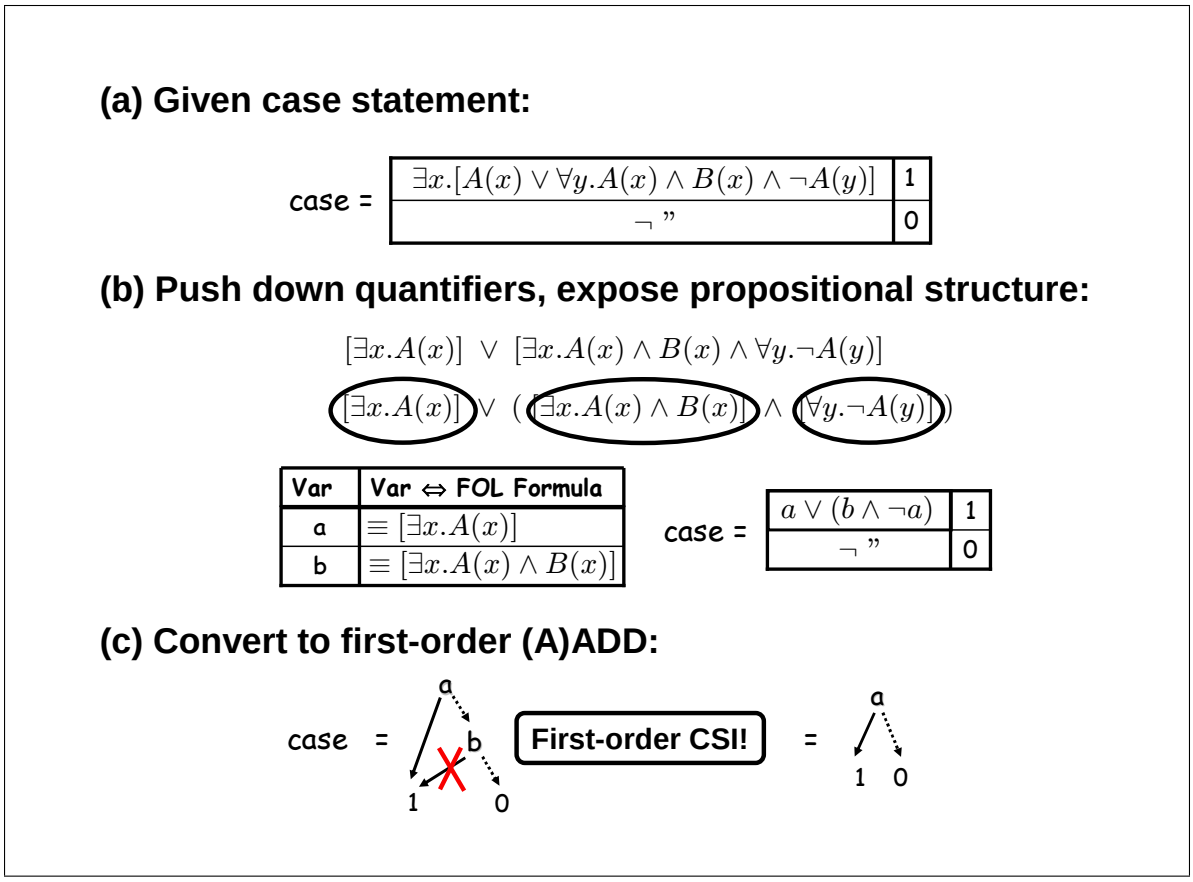


Figure 4.4: An example conversion from a case statement to a compact FOADD representation demonstrating first-order CSI.

formula as follows:

$$\boxed{\exists x.A(x)} \vee \left(\boxed{[\exists x.A(x) \wedge B(x)]} \wedge \boxed{\forall y.\neg A(y)} \right) \tag{4.31}$$

Each of these boxes represents formula that we cannot further decompose into propositional components. Consequently, we rename each of these boxes with propositions. To do this, we maintain a table of mappings from propositional variables p to first-order formulae ψ: $\{p \rightarrow \psi\}$. When we want to convert a formula ϕ to a propositional variable, we examine each formula-to-proposition mapping in our table. If $\phi \equiv \psi$, we return p as the proposition, otherwise if $\phi \equiv \neg\psi$, we return $\neg p$ as the proposition. If no proposition in the table matches then we create a new proposition q and add the mapping $q \rightarrow \phi$ to our table and return q. Having built the table shown in Figure 4.4(b), we can proceed to convert the above formula to

its propositional version:

$$a \vee (b \wedge \neg a) \tag{4.32}$$

At this point, we can build an ADD or AADD from a case statement whose formulae are purely propositional. What makes this (A)ADD first-order is the additional proposition to first-order formula mapping that gives each proposition a first-order definition. While the traditional (A)ADD can exploit CSI, we note there is now an additional form of CSI that we can exploit in FO(A)ADDs — *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 the FO(A)ADD representation of this case formula we can force the decision node for b in the context of a. This is shown in Figure 4.4(c).

We note that there are a range of options for detecting first-order CSI ranging over the following:

1. Do not perform any first-order CSI detection at all.

2. Maintain information about all pairwise implications in the propositional mapping table and detect just this pairwise first-order CSI during the application of FO(A)ADD operations.

3. 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 FO(A)ADD.

Obviously option (1) requires no additional computation at the expense of FO(A)ADDs with potentially dead paths whereas option (3) requires substantial computation in return for full simplication of the decision diagram. We note that if variable reordering is permitted in the FO(A)ADD then one must resort to option (1) since the prunings may not be sound for other reorderings. On the other hand, when we know that variables will not be reordered in the FO(A)ADD, we find option (2) to be a reasonable tradeoff between computation and simplification.

It is straightforward to extend the ADD and AADD algorithms to do consistency checking in the presence of parent decisions when performing the standard (A)ADD $Apply$ and $Reduce$ operations. In doing this, it is important to note that although there may be multiple paths to reach a node in the FO(A)ADD, each distinct path to a node is mutually exclusive and thus other parents need not be considered when pruning along that path. It is simply crucial that pruning along one path does not accidentally prune along another path, but this can be avoided

in the following manner: a parent node should check its child node for implied branches and if one is found, the parent node's pointer should be changed to refer to the child's implied branch. In this way, the child itself is not changed and any other links pointing into the child are not erroneously modified.

We note that replacing case statements with FO(A)ADDs 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 $Regr$ 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. Consequently, our approach to representing FO(A)ADDs is well-suited to FOMDPs as we will soon demonstrate with a small example.

4.6.2 Operations on FO(A)ADDs

Recalling the case of propositionally factored MDPs from Chapter 3, once we had represented factors as (A)ADDs, we could directly apply the standard binary operations of addition, multiplication, and subtraction directly to these data structures. The same basic idea holds for FO(A)ADDs. Once we convert the case representation to these data structures, we can apply the $\otimes$, $\oplus$, and $\ominus$ case operations directly to FO(A)ADDs by making using of the analagous (A)ADD operations. Note that we do not need to consult the proposition to first-order formula mapping table to compute these operations; in general, we only need to do this in three cases:

1. We consult this table when constructing a FO(A)ADD.

2. We consult this table when converting a FO(A)ADD back to a case representation or evaluating a ground state.

3. If we are exploiting first-order CSI, then we may consult this table during the (A)ADD $Reduce$ and $Apply$ procedures.

SDP algorithms for FOMDPs also require special unary operations such as $Regr$, casemax, and $\exists \vec{x}$ that we define now. First, we discuss each FO(A)ADD unary operator at a *general level* as their use in practice has certain limitations that we discuss momentarily:

- $Regr$: We can apply the $Regr$ operator directly to each decision node in a FO(A)ADD. This property is obvious since an (A)ADD can be represented as a propositional expression over propositional variables (with additional affine transforms in the AADD case). Exploiting the previously defined properties of the $Regr$ operator, we can simply push the $Regr$ into these individual propositional nodes. To regress a propositional formulae p at a node, we simply apply $Regr$ directly to the first-order formula ϕ represented by the node (looking up $p \longrightarrow \phi$ in our variable mapping table), and replace it with a new variable q where $q \longrightarrow Regr(\phi)$. We do not attempt to further decompose q at this point. We note that this new formula has free variables (but there is nothing prohibiting this in FO(A)ADDs). However, the one problem that can occur is that the replacement of decision node variables with other variables may introduce conflicts with the global FO(A)ADD ordering. In this case, decision nodes can be internally rotated to correct the (A)ADD variable ordering (see [Rudell, 1993] for ADDs and the *Other Operations* discussion in Section 3.4.2 of Chapter 3 for AADDs).

- $\exists \vec{x}$: The $\exists \vec{x}$ unary case operation can also be applied directly to FO(A)ADDs. In this case we assume that the variables being quantified are present in at least one decision node of the FO(A)ADD (otherwise the quantifiers are vacuous and can be removed). Since an ADD decision node $if\ (a)\ then\ \phi_h\ else\ \phi_l$[15] can be written as the logical expression $(a \wedge \phi_h) \vee (\neg a \wedge \phi_l)$[16], we note that when the decision test a does not contain the free variables being quantified then we can perform the following rewrite:

$$\begin{aligned} &\exists \vec{x}.\, [(a \wedge \phi_h) \vee (\neg a \wedge \phi_l)] \\ &\quad \equiv [(a \wedge \exists \vec{x}.\, \phi_h) \vee (\neg a \wedge \exists \vec{x}.\, \phi_l)] \end{aligned}$$

  This allows us to recursively push the existential quantifiers down through each FO(A)ADD decision node until we encounter a decision node that *does* contain the variables being quantified. At this point, we cannot push the quantifiers down any further. Consequently, the key to an efficient $\exists x.$ operation is to (1) rotate all of the decision nodes with free variables to the bottom of the FO(A)ADD, (2) push the quantifiers down to that level, (3) explicitly convert the FO(A)ADD structure from that level down to a logical or arithmetic representation of the formula, (4) perform the existential quantification, (5) rebuild

[15] Recall our (A)ADD notation from Chapter 3 where we use h to denote the high/true branch of a decision node, and l to denote the low/false branch of a decision node.

[16] A similar property holds for AADDs except that it is an arithmetic expression.

a FO(A)ADD from that logical representation and insert it in place of the old quantified structure, and (6) rotate nodes to maintain variable ordering. We make two important notes: (a) because nodes are being reordered in this approach, we cannot perform pruning, and (b) since nodes may overlap due to existential quantification we perform a casemax (discussed next) simultaneously with this operation to render overlapping nodes disjoint. Overall, this is a non-trivial algorithm to implement and we discuss it further in a moment. An example of this along with the casemax operator discussed next is provided in Figure 4.5.

- casemax: Because this can be an expensive operation, it is important to apply casemax opportunistically directly where it is needed. That is, the only operation that can transform an exhaustive and disjoint case partitioning into one that does not satisfy these properties is the $\exists \vec{x}$ operator. As a consequence, our approach is to apply the casemax jointly with the $\exists \vec{x}$ operator, when needed. Since the $\exists \vec{x}$ requires breaking the quantified part of a FO(A)ADD down into its case representation, we can easily apply both casemax and $\exists \vec{x}$ to this representation. A concrete example which illustrates this is given in Figure 4.5 and an explanation follows.

Here we provide a step-by-step explanation of the joint application of the casemax and $\exists \vec{x}$ operators to a specific FOADD instance in Figure 4.5:

(a) First, we show the case statement represented as a FOADD before the casemax and $\exists \vec{x}$ operators are applied. We have already rotated nodes with free variables to the leaves where we assume the sub-diagrams on the high branches of a and b do not contain the free variables $\vec{x}$.

(b) Assuming that $case(x)$ is a disjoint partitioning of state space, we can push the casemax and $\exists \vec{x}$ operators down until a node is reached where a decision variable referencing a quantified variable is encountered.

(c) We convert the sub-diagram below the casemax$\exists x$ to a $case$ statement and explicitly perform the casemax$\exists x$ operations (not shown). We convert this result back to a FOADD and insert the FOADD in place of the original sub-diagram while inserting any new variables into our propositional mapping table.

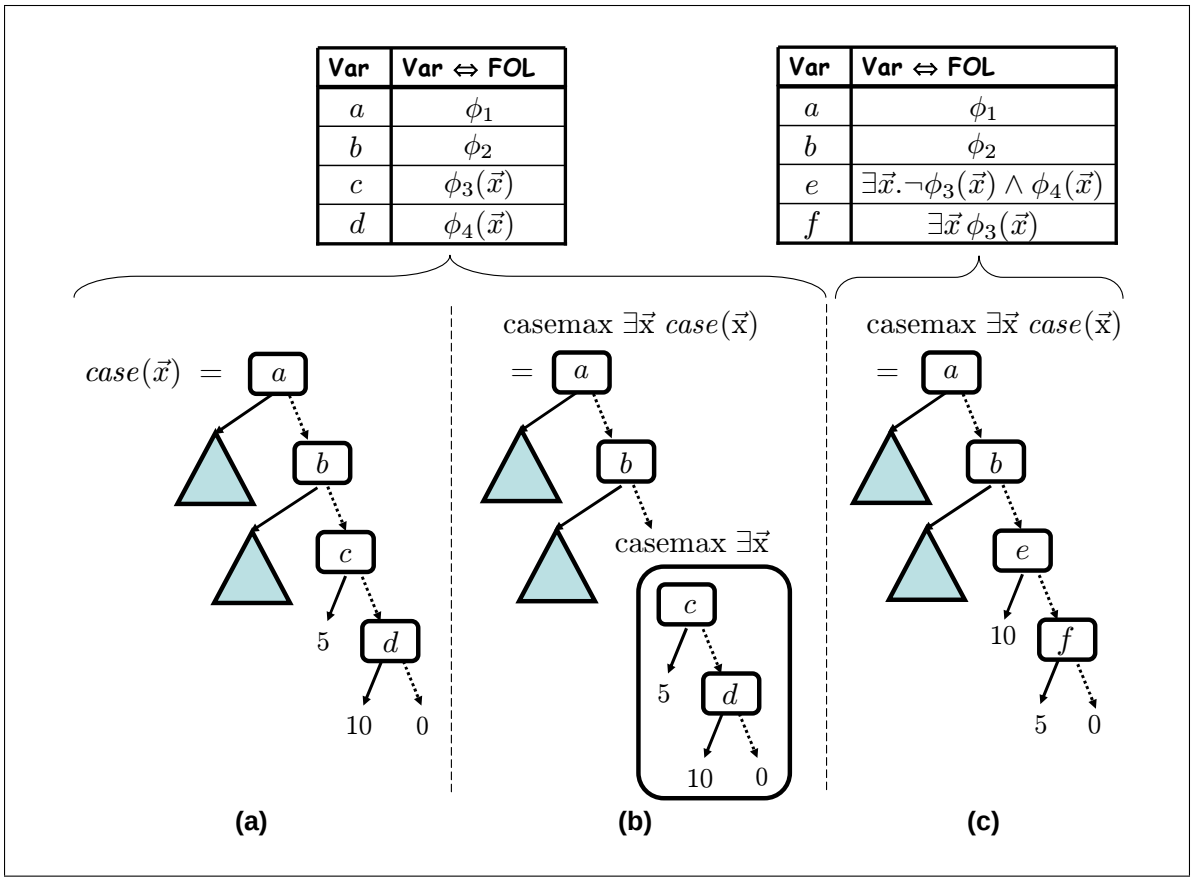


Figure 4.5: Here we demonstrate the joint application of the casemax and $\exists \vec{x}$ operators to an example *case* statement represented as a FOADD. See the text for details.

4.6.3 Practical Considerations

We did not give explicit algorithms for the FO(A)ADD operations and the reasoning for this is that they are not only quite complex to implement in practice, but their computational overhead does not give them significant advantages over the case representation. The reason for this is that the internal node rotations required to maintain canonicity of the FO(A)ADDs are quite expensive.

However, this is not to say that FO(A)ADDs have not been useful in the application of SDP, we simply need to modify the way in which they are used. In SDP, the FO(A)ADDs are best for performing efficient binary operations and formula simplification through the breakdown of propositional structure and the elimination of redundancy that occurs during their construction. In doing these simplifications, the FO(A)ADDs 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 FO(A)ADDs for the purposes that they are useful

and efficient for — binary operations and logical simplification — and we convert back to the case representation to perform most of the unary operations that can be expensive due to the need for internal node rotations. As we will see, this approach has led to a successful SDP algorithm that we discuss next.

4.6.4 Symbolic Dynamic Programming with FO(A)ADDs

The use of FO(A)ADDs in the somewhat hybrid manner discussed previously has led to an automated SDP algorithm. In this approach, we use FO(A)ADDs to perform the binary operations of FODTR and convert to the case representation to perform regression, existential quantification and symbolic maximization. As mentioned previously, the primary benefit of using FO(A)ADDs in this manner is in their ability to compactly represent and simplify the first-order case representation while permitting the efficient computation of binary operations during FODTR. In addition to FO(A)ADDs, we do perform some additional simplification of equality, relying on the non-empty assumption for object domains, the quantifier rewrite rules described previously, and the following two additional rewrite rules:

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

The first rule is fairly straightforward while the second rule follows simply from the negation of the first rule with renaming. We provide the following example application of these previously described rewrite and simplification rules to demonstrate their power in simplifying formulae with equality:

$$\begin{aligned}
&\exists x, z.\, [x = y \wedge A(x, \diamond) \wedge B(y, z)] \\
&\equiv\ \exists x.\, [x = y \wedge A(x, \diamond) \wedge (\exists z.\, B(y, z))] \\
&\equiv\ (\exists x.\, x = y \wedge A(x, \diamond)) \wedge (\exists z.\, B(y, z)) \\
&\equiv\ A(y, \diamond) \wedge (\exists z.\, B(y, z))
\end{aligned}$$

Together using FOADDs and equality simplification, we have managed to provide an automated first-order value iteration solution to our running BOXWORLD FOMDP example. The FOADDs for the reward, optimal value function and policy are given in Figure 4.6. For the variable ordering, we simply maintained the order of formulae as they were added to the variable mapping table in the FOADD during the SDP algorithm. We used the Vampire theorem

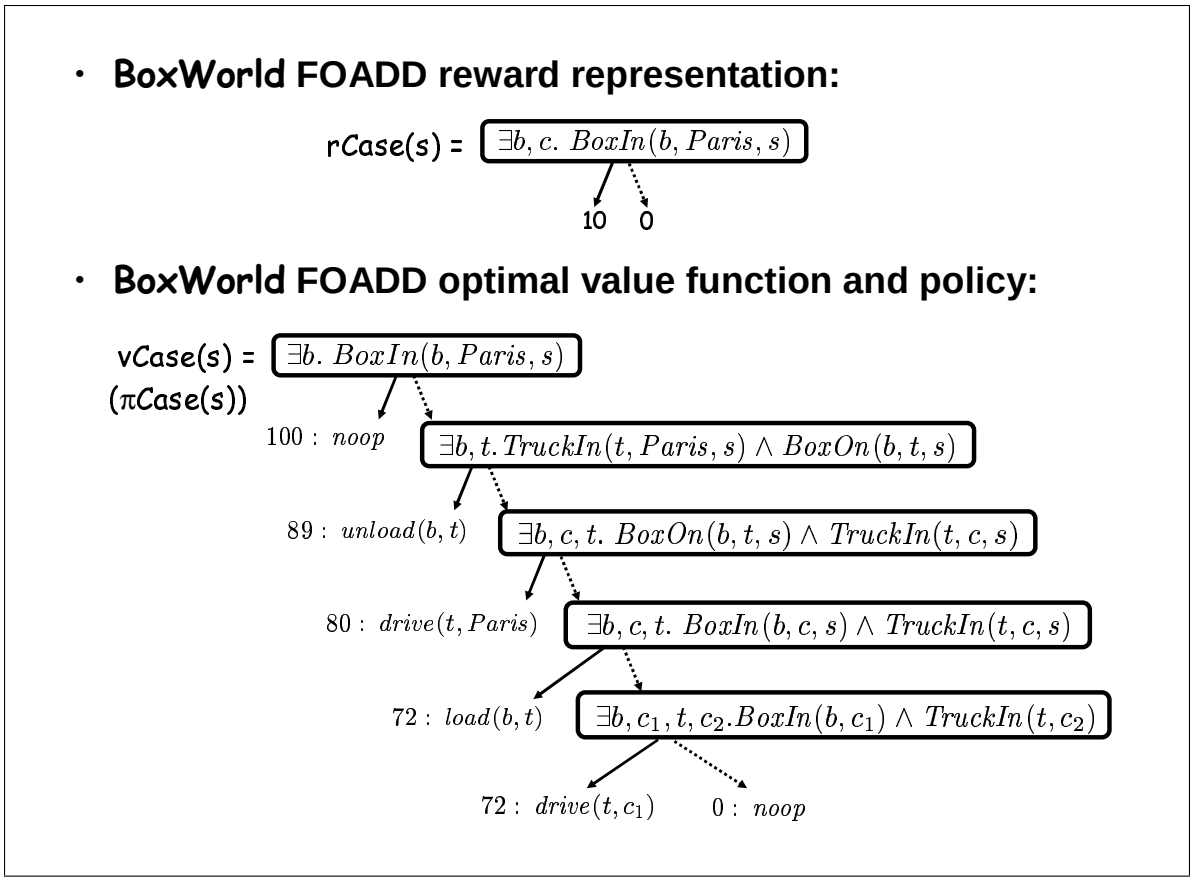


Figure 4.6: An example FOADD representation of the reward in BOXWORLD and the FOADD representation of the optimal value function and policy for this domain.

prover [Riazanov and Voronkov, 2002] as the theorem proving component for detecting equivalence and inconsistency. The total running time for this solution 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.

We have also used our FOADD approach to solve other variants 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 of the following structure:

$$R(s) = \begin{array}{|l|} \hline \exists b. BoxIn(b, paris, s) \wedge TypeA(b) \quad : 10 \\ \hline \neg \text{“} \wedge \exists b. BoxIn(b, paris, s) \wedge \neg TypeA(b) : \ 5 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \tag{4.33}$$

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 finite 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 maximum solution times for these more complex problems was 489s on a 2Ghz Pentium with 2Gb of RAM. The use of FOAADDs led to slightly slower runtimes due to the fact that these test problems did not have any additive or multiplicative structure to exploit.

Our experience indicates that there seem to be two general criteria for problem domains to demonstrate a finitely sized optimal value function: (1) the only non-zero rewards must be existentially quantified and (2) the FOMDP dynamics must not introduce transitive structure that cannot be finitely bounded by domain axioms. This last requirement is somewhat vague, so let us provide an example. In the BOXWORLD problem covered in this chapter, 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 an infinite value function. Another case of an infinite value function comes when (1) is violated as we discuss in the next section which concerns rewards with universal quantifiers. While both of these problems elucidate limitations of the current SDP algorithm, it is possible that with modifications to the SDP algorithm and the case (or FOADD) representation, these difficulties could be overcome.

Unfortunately, the FOADD solution approach with the current SDP algorithm has failed to scale to more complex problems used in the planning community (particularly problems from the ICAPS 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 we are in need of additional solution techniques to handle this problem as we discuss in the next section.

4.7 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 $Dst(b, c)$:

$$R(s) = \begin{array}{|l|} \hline \forall b, c.\ Dst(b, c) \rightarrow BoxIn(b, c, s) : 1 \\ \hline \neg \text{“} \qquad\qquad\qquad\qquad : 0 \\ \hline \end{array} \tag{4.34}$$

One difficulty with such rewards is that our case statements provide a piecewise-constant representation of the value function. However, the value function for problems with universal rewards typically depends (often in a linear or exponential way) on the *number* of domain objects of interest. For instance, 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 for example, 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):

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

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.*, 1998b; Poupart *et al.*, 2002a]. We divide our solution into off-line and on-line compo-

nents where the on-line component requires a finite domain assumption in order to execute the policy.

4.7.1 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. So in the offline solution of the MDP we assume a 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 example, the conditions in the antecedent indicate the goal objects of interest and the head of the implication indicates the specific goal $G(\vec{y}, s)$. In our running BOXWORLD example the conditions $Dst(b, c)$ indicate that we will have goals for all pairs $\langle b, c \rangle$ where $Dst(b, c)$ holds and the goal that must be achieved for these object pairs is $BoxIn(b, c, s)$.

It is easy to construct a generic instance of a reward function $rCase_{G(\vec{y}^*)}(s)$ given a single goal. In our BOXWORLD example we would introduce the distinguished constants b^* and c^* to denote our goal objects of interest $G(b^*, c^*)$:

$$rCase_{G(b^*,c^*)}(s) = \boxed{\begin{array}{l} BoxIn(b^*, c^*, s) \quad : 1 \\ \hline \neg BoxIn(b^*, c^*, s) : 0 \end{array}} \tag{4.35}$$

Given this simple reward, it is then easy to solve the resulting FOMDP using first-order value iteration or the approximate FOMDP solution algorithms that we will introduce in the next chapters. This produces a value function $vCase_{G(\vec{y}^*)}(s)$ and policy that assumes that $\vec{y}^*$ is the only object vector of interest satisfying relevant sort 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 4.3 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-function instances for each action $A_i(\vec{x})$ from the value function $vCase_{G(\vec{y}^*)}(s)$ for the simplified "generic" domain:

$$qCase_{G(\vec{y}^*)}(A_i, s) = \exists \vec{x}.\ FODTR[vCase_{G(\vec{y}^*)}(s), A_i(\vec{x})] \tag{4.36}$$

Given a ground state s, the optimal action for this generic goal can be determined by finding the ground action instantiation $A_i(\vec{c})$ for this s with maximal Q-value.

4.7.2 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 would obtain 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:

$$\{qCase_{G(b_1,paris)}(A_i, s),\ qCase_{G(b_2,berlin)}(A_i, s),\ qCase_{G(b_3,rome)}(A_i, s)\} \tag{4.37}$$

Action selection requires finding an action that maximizes value w.r.t. the original universal reward. Following [Boutilier *et al.*, 1997; Meuleau *et al.*, 1998b], we do this by treating the *sum of the Q-values* of any action in the subgoal MDPs as 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 7.

While this additive and uniform decomposition may not be appropriate for all domains with goal-oriented universal rewards, we have found it to be highly effective for domains such as BOXWORLD as we demonstrate empirically in the next chapter. And while this approach can only currently handle rewards with universal quantifiers, this reflects the form of many planning problems. Nonetheless, there are potential extensions of this technique for more

Algorithm 7: $EvalPolicy(\{qCase_{G(\vec{y}^*)}(A_i, s)\}, \{G(\vec{y_1}), \ldots, G(\vec{y_n})\}, s) \longrightarrow A_i(\vec{c})$

input : (1) For each action template $A_1(\vec{x}), \ldots, A_m(\vec{x})$, a set of 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 w.r.t. 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(x) → v.
Initialize empty hash table h;
// Now, compute additive values for all matching ground actions
foreach (action A_i) do
    foreach (goal G(y_j)) do
        Replace all occurrences of y* in qCase_{G(y*)}(A_i, s) with y_j;
        foreach (case partition ⟨∃x φ(x), t⟩ ∈ qCase_{G(y_j)}(A_i, s)) do
            foreach (ground binding x = c satisfying ∃x φ(x)) do
                if (A_i(c) → v is already in h for some v) then
                    Update h to contain A_i(c) → (v + t/n);
                else
                    Update h to contain A_i(c) → t/n;
// Assume h tracks its maximal entry: A_i(c) → v.
Return the maximal A_i(c) from h;
```

end

complex universal rewards, the general open question being how to assign credit among the constituents of such a reward.

4.8 Related Work

A variety of exact algorithms have been introduced to solve MDPs with relational (RMDP) and first-order (FOMDP) structure.[17] *Symbolic dynamic programming (SDP)* [Boutilier *et al.*, 2001] is the original first-order value iteration algorithm for solving FOMDPs introduced here. *First-order value iteration (FOVIA)* [Hölldobler and Skvortsova, 2004; Karabaev and Skvortsova, 2005] and the *relational Bellman algorithm (ReBel)* [Kersting *et al.*, 2004] are value iteration algorithms for solving RMDPs. *First-order decision diagrams (FODDs)* have been introduced to compactly represent case statements and to permit efficient application of

[17] We use the term *relational MDP* to refer to models that allow implicit existential quantification, and *first-order MDP* for those with explicit existential and universal quantification.

symbolic dynamic programming operations to solve RMDPs via value iteration [Wang *et al.*, 2007] and policy iteration [Wang and Khardon, 2007]; we elaborate on the differences between FOADDs and FODDs in a moment. All of these algorithms have some form of guarantee on convergence to the (ϵ-)optimal value function or policy. However, aside from the SDP algorithm discussed at length in this chapter, all of these other methods are restricted to RMDPs and thus do not permit the explicit specification of universally quantified formulae in their representation. As for approximate and heuristic FOMDP solution algorithms, we discuss these approaches at the end of the next chapter in the context of our own approximate FOMDP solution algorithms.

Since FODDs are very similar in spirit to the FO(A)DDs we defined in this chapter, we enumerate some of the major differences between these two formalisms:

1. FODDs disallow explicit universal quantification in rewards and to some extent in their SSA representation when variables in both the precondition and the post-action fluent do not occur as action parameters. This prohibits ADL extensions of STRIPS planning such as non-local universal effects.

2. FODDs rely on a range of simplification rules to maintain compact representations. However, rather than having a well-defined simplification algorithm, simplification in FODDs is somewhat open-ended and heuristic.

3. Rather than perform explicit $\exists x.$ and casemax operations, FODDs assume an implicit semantics where the maximal value is assumed for all instantiations of the free variables. This can lead to very compact representations during value iteration, but this semantics requires more complex computation during policy evaluation and 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 w.r.t. the work we present in this thesis is that the expressiveness of FODD-based FOMDPs is limited to probabilistic extensions of STRIPS and minor variants. Ideally the best approach would be to combine the advantages of FO(A)ADDs with those of FODDs. This is a non-trivial problem, however, and is not addressed in the current research literature.

In concluding the discussion of related work, we summarize by noting that the SDP algorithms covered in this chapter are the *only* methods capable of exactly solving FOMDPs with

both explicit existential and universal quantifiers in their specification. Thus, SDP and the extensions that we will define in future chapters are the only FOMDP solution algorithms that can generally handle the important planning construct of non-local universal effects from PPDDL (c.f., the PPDDL/ADL discussion at the beginning of this chapter).

4.9 Summary

In concluding this section on FOMDPs, we note that this framework offers many attractive properties from an MDP perspective. First, it allows one to draw on relational probabilistic planning problem specifications like PPDDL to specify FOMDP dynamics directly. Furthermore, FOMDP solution algorithms such as first-order value iteration are completely domain-independent and do not require explicit state and action enumeration. Therefore these techniques can solve for very concise representations of optimal value functions and policies when they exist, even when the underlying domain may be infinitely sized as in BOXWORLD.

On the other hand, the expressivity of FOMDPs comes with a few drawbacks. First, theorem proving and and a range of first-order logic simplification methods are required to maintain compact case representations. While techniques such as FOADDs and FOAADDs substantially reduce the simplification and theorem proving burden by exploiting propositional structure common to many FOMDPs, these approaches merely delay the inevitable fact that current simplification and theorem proving technologies can only scale so far. Second, although our case, FOADD, and FOAADD representations are attempts at compactly representing structure common to many FOMDPs, even these structures are inadequate for problems with difficult reward structure such as universal rewards. In this case we had to suffice with an approximate decomposition-based solution technique, albeit an ad-hoc one with no general performance guarantees. But the need for approximation in order to obtain tractable solutions is a general lesson that we should take to heart. Ideally though, we would also desire to have some form of error bounds on approximate solutions, something possible with APRICODD-style extensions [St-Aubin *et al.*, 2000] of SDP that have not been explored to date.

In general, even though we can now exploit structure that was not possible with ground MDPs or factored MDPs, the fact that we are now domain-independently representing and solving FOMDPs adds a new dimension of complexity that is not easily overcome in exact solution approaches. Given that approximate solution approaches such as linear-value approximation [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001; de Farias and Van Roy, 2003] have allowed MDP solution algorithms to scale far beyond the limits of exact algorithms while

offering reasonable loss-bounds on performance, this suggests that we might be able to achieve similar results by generalizing linear-value solution techniques to FOMDPs, which we do next.

Chapter 5

Linear-value Approximation for FOMDPs

Perhaps the greatest difficulty with the previously described exact and approximate value iteration solutions for FOMDPs is that the size of the value function case representation can grow according to a high-order polynominal on each iteration[1] and thus exponentially in terms of the number of iterations. Similar growth properties can occur for the first-order formulae representing the state partitions of the value function. Once these formulae become too large to 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. Unfortunately, current research has not identified an alternate representation nor a set of logical simplification rules that can maintain relatively compact case statements across a variety of planning problems.

Thus, faced with the difficulty of exact and approximate value iteration-based MDP solution methods, we seek alternate approaches based on linear-value approximation. In this paradigm, we reduce the task of solving a FOMDP to that of obtaining good weights for a set of basis functions that approximates the optimal value function. We have already defined such techniques for ground (factored) MDPs and in this chapter, our goal is to generalize these frameworks to the first-order case. This is a non-trivial task as it requires the generalization of linear programs to the case with first-order constraints and efficient extensions of solution methods such as constraint generation and variable elimination in cost networks to exploit the first-order structure of these constraints.

In the process of developing a completely automated linear-value approximation solution approach to FOMDPs and in an effort to answer the question "where do basis functions come

[1]Note that in the worst case, just 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.

from?", we adapt techniques proposed by Gretton and Thiebaux [2004] for the automatic generation of basis functions. With appropropriate domain axioms defining legal states, these techniques give us a practical automated approach to decision-theoretic planning in PPDDL-derived representations; we demonstrate the efficacy of these techniques on probabilistic planning problems from the ICAPS 2004 and ICAPS 2006 International Probabilistic Planning Competitions [Littman and Younes, 2004b; Gerevini *et al.*, 2006].

Parts of the work described in this chapter appeared in [Sanner and Boutilier, 2005; Sanner and Boutilier, 2006].

5.1 Linear-value Approximation with Basis Functions

Linear-value approximation solutions to FOMDPs are attractive for a number of reasons:

- Given that much of the computation in linear-value approximation solutions reduces to linear program optimization, this reduces the algorithm design space to the setup and solution of linear programs.
- Since the size of linear-value approximations is fixed, the size of the linear-value approximation 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.
- For algorithms like approximate policy iteration, we can obtain error bounds on the resulting value approximation if the algorithm converges, thus providing us with a domain-independent bound on approximate solution quality.
- Linear-value approximation solutions do not require logical simplification, just weight projections that make use of a theorem prover. This is a huge advantage over exact techniques that require simplification in order to maintain a compact representation.
- Linear-value approximation solutions have yielded reasonable empirical performance for ground and factored MDPs, which is an encouraging indication that these results may extend to FOMDPs.

Motivated by the potential advantages of linear-value approximation, we now proceed to generalize our representation from the propositional case to the first-order case.

5.1.1 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:

$$V(s) = \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \tag{5.1}$$

Throughout this chapter, we will 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.

Using this linear-value function representation, we can often achieve a reasonable approximation of the exact value function by exploiting the additive structure inherent in many real-world problems. For example, as argued in previous chapters, many planning problems have additive reward functions or multiple goals to be achieved, 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, we may wish to approximate the value function for our BOXWORLD FOMDP from the last chapter as follows where we refer to specific instances of $b_i(s)$ as $bCase_i(s)$:

$$bCase_1(s) = \begin{cases} \exists b.\ BoxIn(b, paris, s) : 1 \\ \neg\text{“} \quad : 0 \end{cases} \tag{5.2}$$

$$bCase_2(s) = \begin{cases} \exists b, t.\ BoxOn(b, t, s) : 1 \\ \neg\text{“} \quad : 0 \end{cases} \tag{5.3}$$

$$bCase_3(s) = \begin{cases} \exists b, t.\ TruckIn(t, paris, s) \wedge BoxOn(b, t, s) : 1 \\ \neg\text{“} \quad : 0 \end{cases} \tag{5.4}$$

and a specific instance of $V(s)$ as $vCase(s)$:

$$vCase(s) = w_1 \cdot bCase_1(s) \oplus w_2 \cdot bCase_2(s) \oplus w_3 \cdot bCase_3(s) \tag{5.5}$$

Here we note that 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 BOX-WORLD value function.

5.1.2 Backup Operators

Just as we defined an action backup operator for MDP value functions as a useful notation in Chapter 3, we can do the same for FOMDP case representations. Suppose we are given a value function in the form $V(s)$. 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 4.22 in Chapter 4, except that we do not explicitly add in the reward. Slightly modifying our definitions from Section 4.3.3, we let $n_1(\vec{x}), \ldots, n_q(\vec{x})$ be the set of Nature's deterministic action outcomes for stochastic action $A(\vec{x})$.[2] Then we define $B^{A(\vec{x})}[\cdot]$ as follows:

$$\begin{aligned} &B^{A(\vec{x})}[V(s)] \\ &\qquad = \gamma \left[\bigoplus_{j=1}^{q} \{P(n_j(\vec{x}), A(\vec{x}), s) \otimes Regr(V(do(n_j(\vec{x}), s)))\}\right] \end{aligned} \tag{5.6}$$

Defining $B^{A(\vec{x})}[\cdot]$ in this way makes it a linear operator with properties similar to the linear operators we defined for factored MDPs in Chapter 3, Equation 3.10. Thus, if we apply this

[2] In general, the set of Nature's choice actions $n_1(\vec{x}), \ldots, n_q(\vec{x})$ are associated with a stochastic action $A(\vec{x})$ and $A(\vec{x})$ will always be clear from context.

operator to our linear-value function representation, we see that it distributes to each first-order basis function:

$$\begin{aligned} B^{A(\vec{x})}[V(s)] &= B^{A(\vec{x})}\left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s)\right] \\ &= \bigoplus_{i=1}^{k} w_i \cdot B^{A(\vec{x})}\left[b_i(s)\right] \end{aligned} \tag{5.7}$$

Having defined $B^{A(\vec{x})}[\cdot]$, we now use it to define $B^{A}[\cdot]$:[3]

$$B^{A}[V(s)] = \exists \vec{x}.\ \left\{B^{A(\vec{x})}[V(s)]\right\} \tag{5.8}$$

Unfortunately, if we apply $B^{A}[\cdot]$ to our linear-value function representation in the following manner

$$\begin{aligned} B^{A}[V(s)] &= B^{A}\left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s)\right] \\ &= \exists \vec{x}.\left\{\bigoplus_{i=1}^{k} w_i \cdot B^{A(\vec{x})}\left[b_i(s)\right]\right\} \end{aligned} \tag{5.9}$$

we see that $B^{A}[\cdot]$ is not necessarily a linear operator. The difficulty in this case 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.

However, all is not lost. To show how these problems can be mitigated, we begin with a few definitions.

Definition 5.1.1. *We say that a deterministic Nature's choice 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 4.2.2). We say that a stochastic action $A(\vec{x})$ affects a fluent F if at least one of Nature's choice deterministic outcomes $n_j(\vec{x})$ of $A(\vec{x})$ affects F. And we say that a formula ϕ is affected by a stochastic $A(\vec{x})$ action 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.*

Next, we note the following property:

[3]For simplicity, we assume that the reward is independent of the arguments $\vec{x}$ for any action $A(\vec{x})$ and thus omit such reward dependencies here. However, if this was not the case, we could easily insert it in this equation and make appropriate adjustments to our later equations.

Property 5.1.2. 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 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 $i \in I_A^-$, $B^{A(\vec{x})}[b_i(s)]$ does not contain the free variables $\vec{x}$). Then, we 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 result for B^A applied to a linear-value function representation:

$$B^A\left[\bigoplus_i w_i b_i(s)\right] \tag{5.10}$$
$$= \left(\bigoplus_{i \in I_A^-} w_i B^{A(\vec{x})}\left[b_i(s)\right]\right) \oplus \exists \vec{x}.\ \left(\bigoplus_{i \in I_A^+} w_i B^{A(\vec{x})}\left[b_i(s)\right]\right)$$

Consequently, if no fluent occurs in more than a few basis functions and few fluents are affected by an action then we can reasonably expect the result of applying B^A to retain some additive structure.

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 5.5:

$$B^{drive}[vCase(s)] = \exists t^*, c^*\ B^{drive(t^*,c^*)}[vCase(s)] \tag{5.11}$$
$$= \exists t^*, c^*\ B^{drive(t^*,c^*)}[w_1 \cdot bCase_1(s) \oplus w_2 \cdot bCase_2(s) \oplus w_3 \cdot bCase_3(s)]$$
$$= \exists t^*, c^*\ \left\{w_1 \cdot B^{drive(t^*,c^*)}[bCase_1(s)] \oplus w_2 \cdot B^{drive(t^*,c^*)}[bCase_2(s)]\right.$$
$$\left. \oplus w_3 \cdot B^{drive(t^*,c^*)}[bCase_3(s)]\right\}$$
$$= \exists t^*, c^* \left\{ w_1 \cdot \begin{array}{|l|r|} \hline \exists b.\ BoxIn(b, paris, s) & : 0.9 \\ \hline \neg \text{“} & : \ 0 \\ \hline \end{array} \oplus w_2 \cdot \begin{array}{|l|r|} \hline \exists b, t.\ BoxOn(b, t, s) & : 0.9 \\ \hline \neg \text{“} & : \ 0 \\ \hline \end{array} \right.$$
$$\left. \oplus w_3 \cdot \begin{array}{|l|r|} \hline \exists b, t.\ [t = t^* \wedge c^* = paris \wedge \exists c_1\, TruckIn(t, c_1, s)] & \\ \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) & : 0.9 \\ \hline \neg \text{“} & : \ 0 \\ \hline \end{array} \right\}$$

Here, we note that the first and second basis functions were 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:

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

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

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

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 as described in Section 4.2.3 of Chapter 4. This allows us to remove the $\exists t^*, c^*$ by rewriting equalities and exploiting the non-empty domain assumption:

$$B^{drive}[vCase(s)] = \exists t^*, c^*.\ B^{drive(t^*,c^*)}[vCase(s)] \tag{5.12}$$

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

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

This example demonstrates the best case 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.

Unfortunately 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 worst case effectively cancels 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 aspect during basis function generation.

5.2 Approximate Solution Methods

In this section, we describe two useful linear-value approximation methods: first-order approximate linear programming (FOALP), and first-order approximate policy iteration (FOAPI). Both of these methods are generalized from the propositional MDP case outlined in Chapter 3, Section 3.3.2 to the first-order case. We do not cover first-order approximate value iteration due to its issues with divergence in the ground case [Tsitsiklis and Van Roy, 1996; Guestrin *et al.*, 2001] that trivially extend to the first-order case. Indeed, our own experimentation with first-order approximate value iteration approaches has proved fruitless due to divergence issues.

5.2.1 First-order Approximate Linear Programming

We now generalize the approximate linear programming (ALP) approach for propositional factored MDPs from Equation 3.15 in Chapter 3 to first-order MDPs. If we simply substitute appropriate notation, we arrive at the following formulation of the first-order ALP (FOALP) approach:

$$\begin{aligned}
&\text{Variables: } w_i \ ; \ \ \forall i \leq k \\
&\text{Minimize: } \sum_s \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \\
&\text{Subject to: } 0 \geq R(s) \oplus B^A \left[\bigoplus_{i=1}^{k} w_i \cdot b_i(s) \right] \\
&\qquad\qquad \ominus \bigoplus_{i=1}^{k} w_i \cdot b_i(s) \ ; \ \forall \ A, s
\end{aligned} \tag{5.13}$$

As for 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. Unfortunately, while the objective and constraints in ALP for a factored MDP range over a finite number of states s, this

direct generalization to the FOALP approach for FOMDPs requires dealing with infinitely (or indefinitely) many situations s.

Since we are summing over infinitely many situations 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 the ALP approach of Equation 3.15, 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 (recall the case statement representation from Equation 4.8), we note that 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 would 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 above FOALP objective as the following:

$$\begin{aligned} \sum_s \bigoplus_{i=1}^{k} w_i \cdot b_i(s) &= \bigoplus_{i=1}^{k} w_i \sum_s b_i(s) \\ &\sim \bigoplus_{i=1}^{k} w_i \sum_{\langle \phi_j, t_j \rangle \in b_i} \frac{t_j}{|b_i|} \end{aligned} \tag{5.14}$$

Here we use $|b_i|$ to indicate the number of partitions in the ith basis function. Thus, we see that this approach can be seen as aggregating states within a basis function partition into one abstract state and then weighting this abstract state uniformly in importance w.r.t. the other abstract states. When the b_i are simply indicator functions for some conditions as we will often assume in this chapter, we note the objective further simplifies to $\sum_i w_i$ — every basis function and its associated weight is equally important. 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. We discuss the strengths of weaknesses of such an approach in our concluding remarks for this chapter.

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 situation 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 (for which we provide an upcoming example in Equation 5.17) hold iff the original infinite set of constraints in Equation 5.13 hold.

To understand this, let us provide an example of the constraints for the $drive$ action for FOALP substituting our previously defined basis functions $bCase_i(s)$ from Equation 5.4 for $b_i(s)$, the results of the B^{drive} operator for these basis functions from Equation 5.12, and the reward definition for BOXWORLD given by $rCase(s)$ in Equation 4.9 for $R(s)$. We substitute all of these directly into the constraint form of Equation 5.13 above:

$$
\begin{aligned}
0 \geq & \begin{array}{|l|}\hline \exists b. BoxIn(b, paris, s) : 10 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \oplus w_1 \cdot \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \\
& \oplus w_2 \cdot \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \\
& \oplus w_3 \cdot \begin{array}{|l|}\hline \exists b, t.\ [(\exists c_1.\ TruckIn(t, c_1, s)) \vee TruckIn(t, paris, s)] \wedge BoxOn(b, t, s) : 0.9 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \\
& \ominus w_1 \cdot \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 1 \\ \hline \neg \text{“} \quad : 0 \\ \hline \end{array} \ominus w_2 \cdot \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : 1 \\ \hline \neg \text{“} \quad : 0 \\ \hline \end{array} \\
& \ominus w_3 \cdot \begin{array}{|l|}\hline \exists b, t.\ TruckIn(t, paris, s) \wedge BoxOn(b, t, s) : 1 \\ \hline \neg \text{“} \quad : 0 \\ \hline \end{array} ; \ \forall\, s
\end{aligned} \tag{5.15}
$$

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:

$$
\begin{aligned}
0 \geq & \begin{array}{|l|}\hline \exists b.\ BoxIn(b, paris, s) : 10 - 0.1 \cdot w_1 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \oplus \begin{array}{|l|}\hline \exists b, t.\ BoxOn(b, t, s) : -0.1 \cdot w_2 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} \\
& \oplus \begin{array}{|l|}\hline \exists b, t.\ TruckIn(t, paris, s) \wedge BoxOn(b, t, s) \quad : -0.1 \cdot w_3 \\ \hline \neg \text{“} \wedge \exists b, t, c_1.\ TruckIn(t, c_1, s) \wedge BoxOn(b, t, s) : \ 0.9 \cdot w_3 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array} ; \ \forall\, s
\end{aligned} \tag{5.16}
$$

To maintain our representation in a compact and perspicuous form, we define the following

propositional renamings for the first-order formulae in these case statements:[4]

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

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

$$\begin{array}{|l|l|r|}
\hline
\phi_1(s) \wedge \phi_2(s) \wedge \phi_3(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 + -0.1 \cdot w_3 \\
\hline
\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 + 0.9 \cdot w_3 \\
\hline
\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 \\
\hline
\phi_1(s) \wedge \neg\phi_2(s) \wedge \phi_3(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_3 \\
\hline
\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + 0.9 \cdot w_3 \\
\hline
\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s) & : & 0 \geq 10 - 0.1 \cdot w_1 + -0.1 \cdot w_2 \\
\hline
\neg\phi_1(s) \wedge \phi_2(s) \wedge \phi_3(s) & : & 0 \geq -0.1 \cdot w_2 + -0.1 \cdot w_3 \\
\hline
\neg\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s) & : & 0 \geq -0.1 \cdot w_2 + 0.9 \cdot w_3 \\
\hline
\neg\phi_1(s) \wedge \phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s) & : & 0 \geq -0.1 \cdot w_2 \\
\hline
\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \phi_3(s) & : & 0 \geq -0.1 \cdot w_3 \\
\hline
\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \phi_4(s) & : & 0 \geq 0.9 \cdot w_3 \\
\hline
\neg\phi_1(s) \wedge \neg\phi_2(s) \wedge \neg\phi_3(s) \wedge \neg\phi_4(s) & : & 0 \geq 0 \\
\hline
\end{array} \; ; \; \forall\, s \tag{5.17}$$

Here, if any case 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 situation 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, one approach to the FOALP solution would enumerate all consistent constraints for every action and then directly solve the resulting LP. In addition to the above constraints for the $drive$

[4]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 and FOAPI.

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.

However, as the number of basis functions increases in size, the number of constraints can clearly grow exponentially in the number of case statements in the constraint, just as in the propositional version where the number of constraints was exponential in the number of state variables. We reviewed various techniques in Section 3.3.2 for working around this problem in the propositional case, including the constraint generation techniques of Schuurmans and Patrascu [2001] used to efficiently generate a subset of the constraints required to solve the LP. This suggests we might benefit by generalizing constraint generation to solve first-order LPs. But before we attack this problem specifically for FOALP, we introduce first-order approximate policy iteration that happens to define a first-order LP similar to FOALP.

5.2.2 First-order Approximate Policy Iteration

We now generalize approximate policy iteration from the propositional case to the case of first-order approximate policy iteration (FOAPI).

To start off, FOAPI requires that we derive a suitable first-order policy representation from a value function $V(s)$. For this, we can use the policy representation that we introduced in Equation 4.28 of Chapter 4 updated to use the $B^A[\cdot]$ operator with our implicit linear-value function representation of $V(s)$ from Equation 5.1 where we assume m stochastic actions of the form $A_i(\vec{x})$:

$$\pi(s)[V(s)] = \text{casemax}\,[R(s) \oplus \bigcup_{i=1\ldots m} B^{A_i}[V(s)] \tag{5.18}$$

At this point, we know that the the result of $B^{A_i}[\cdot]$ may retain linear structure, however we have not given a definition of the $\cup$ operator that can exploit additive structure. We return to this issue in a moment, but for now assume that the explicit "cross-sum" $\oplus$ is applied to (1) any additive structure remaining in the result of the $B^{A_i}[\cdot]$ operator and (2) the sum of $R(s)$ and the result of the $\cup$ operator.

We will assume that this policy derivation method makes the additional policy annotations that were made for Equation 4.28. To provide an example, we recall $vCase(s)$ from Equation 5.5 and assume that we have the weight assignment $\{w_1 = 10, w_2 = 0, w_3 = 0\}$ for this linear-value function representation. This weight assignment essentially reduces $vCase(s)$ to

a representation of $rCase(s)$, so we let $V(s) = rCase(s)$ and recall the result of our previous policy instance $\pi Case(s)[rCase(s)]$ derived in this instance:

$$\pi Case(s)[rCase(s)] \tag{5.19}$$

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

We note that the weights we chose for $vCase(s)$ made the derivation of the policy quite trivial (by zeroing out the second and third basis functions), so let us briefly digress to consider a more complex case and its implications. If $\{w_1 = 1, w_2 = 1, w_3 = 1\}$, the first issue one might notice when attempting to derive a greedy policy is that the $\cup$ operator forces us to perform an explicit "cross-sum" $\oplus$ over its operands (assuming they retained some additive structure after $B^A[\cdot]$ was applied). However, performing an explicit sum of the operands forces policy derivation to require time and space that is exponential in the number of basis functions, effectively cancelling out the representational benefits of a linear-value function representation. However, there are a number of ways to avoid this exponential blowup:

1. We could attempt to exploit additivity in the policy representation with additional representational machinery.

2. We could use the method of comparing each action Q-function to a $noop$ policy in the spirit of Guestrin *et al.* [2001; 2002], in an attempt to extract a compact policy representation.

3. We could exploit assumptions in the specification of the basis functions that allow us to achieve a compact policy representation.

While all of these options are viable, the simplest and most straightforward method for our purposes comes from the third choice. However, since we have not yet covered basis function generation, we postpone this discussion until later and assume for now that $\pi Case(s)$ is compact and can be derived efficiently (i.e., without considering policy representations that are exponential in the number of basis functions).

For FOAPI approach, we will need to define a set of case statements for each action A_i that is satisfied only in the world states where A_i should be applied according to $\pi(s)$. Conse-

quently, we define an action restricted policy $\pi_{A_i}(s)$ as follows:

$$\pi_{A_i}(s) = \{\langle \phi, t \rangle | \langle \phi, t \rangle \in \pi(s) \text{ and } \langle \phi, t \rangle \rightarrow A_i\}$$

Our previous policy example from Equation 5.19 allows us to derive two example instantiations $\pi\, Case_{unload}(s)$ and $\pi\, Case_{noop}(s)$ of action restricted policies:

$$\pi\, Case_{unload}(s) = \boxed{\begin{array}{l} \neg[\exists b.\, BoxIn(b, paris, do(a, s))] \\ \wedge[\exists b^*, t^*, c.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] : 8.1 \longrightarrow unload(b^*, t^*) \end{array}}$$

$$\pi\, Case_{noop}(s) = \boxed{\begin{array}{l|r} \exists b.\, BoxIn(b, paris, do(a, s)) & : 19.0 \longrightarrow noop \\ \hline \neg \text{“} \wedge \neg[\exists b^*, t^*, c.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] : & 0 \longrightarrow noop \end{array}}$$

Now, we make two additional modifications to π_{A_i}. Given that we know that every partition of π_{A_i} contains one non-negated formula of the form $\exists \vec{x}^*.\phi(\vec{x}^*)$ (meaning if this formula can be satisfied, execute $A_i(\vec{x}^*)$), we briefly "unquantify" the formula and re-express it in terms of free variables — essentially, we remove the $\exists \vec{x}^*$. We refer to the result as $\pi_{A_i(\vec{x}^*)}$ since the action variables are again free. Also, following the example of our policy indicator functions from Equation 3.13 in Chapter 3, we convert the value of the case statement to be 0.[5] To make this concrete, let us modify $\pi\, Case_{unload}(s)$ as described to yield $\pi\, Case_{unload(b^*,t^*)}(s)$:

$$\pi\, Case_{unload(b^*,t^*)}(s) = \boxed{\begin{array}{l} \neg[\exists b.\, BoxIn(b, paris, do(a, s))] \\ \wedge[\exists c.\, BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] : 0 \longrightarrow unload(b^*, t^*) \end{array}} \tag{5.20}$$

Of course the only differences between $\pi\, Case_{unload}(s)$ and $\pi\, Case_{unload(b^*,t^*)}(s)$ are that the action variables b^* and t^* are now free variables and the value of 8.1 has been converted to 0. The reason for doing this will become apparent now that we can define the first-order LP for

[5] In Equation 3.13 we set all policy "partitions" where the corresponding action should be applied to 0 and all other partitions to $-\infty$ so that the constraint would be trivially satisfied. In the first-order case, the case representation affords us the ability to simply not represent any partitions where the policy would not be applied as this will prevent any constraint from being applied for these situations.

FOAPI.

We can generalize approximate policy iteration to the first-order case by calculating successive iterations of weights $w_j^{(i)}$ that represent the best approximation of the fixed-point value function for policy $\pi^{(i)}(s)$ at iteration i. We do this by performing the following two steps at every iteration i after initializing $\vec{w}^{(0)} = \vec{0}$ and $i = 1$:

1. Obtain the policy $\pi^{(i)}(s)$ from the current value weights $\vec{w}^{(i-1)}$ using Equation 5.18.

2. Solve the following first-order LP that determines the weights $\vec{w}^{(i)}$ for the L_∞ minimizing projection of the approximate value function for policy $\pi^{(i)}(s)$:

$$\begin{aligned} &\text{Variables: } w_1^{(i)}, \ldots, w_k^{(i)} \\ &\text{Minimize: } \beta^{(i)} \\ &\text{Subject to: } \beta^{(i)} \geq \left| R(s) \oplus \exists \vec{x}^* \left(\pi^{(i)}_{A(\vec{x}^*)}(s) \oplus B^{A(\vec{x}^*)} \left[\bigoplus_{j=1}^{k} w_j^{(i)} \cdot b_j(s) \right] \right) \right. \\ &\qquad \left. \ominus \bigoplus_{j=1}^{k} w_j^{(i)} \cdot b_j(s) \right| ; \ \forall A, s \end{aligned} \tag{5.21}$$

3. If $\pi^{(i)}(s) = \pi^{(i-1)}(s)$ (equivalently $\vec{w}^{(i)} = \vec{w}^{(i-1)}$) or $\beta^{(i)}$ is less than a prespecified tolerance then exit, else increment i and goto step (1).

Here we note that this first-order LP is an exact analogue of Equation 3.12 from Chapter 3. We use $\pi^{(i)}_{A(\vec{x}^*)}(s)$ for two purposes: (1) as an indicator function to restrict the validity of the constraint for action A only to those regions of state space where A should apply according to the policy, and (2) as a constraint to enforce action selection in the free $B^{A(\vec{x}^*)}$ variable backup operator to be consistent with one of the policy partitions.[6]

We've reached convergence *if* $\pi^{(i)}(s) = \pi^{(i-1)}(s)$ (or equivalently $\vec{w}^{(i)} = \vec{w}^{(i-1)}$). And if convergence is reached, the following theorem holds given the projection error $\beta^{(i)}$ obtained from the final LP solution of Equation 5.21:

Theorem 5.2.1. *Let $V(s)$ be the approximated value function obtained by the weights $\vec{w}^{(i)}$ of the final LP solution of Equation 5.21 for FOAPI applied to a given FOMDP where FOAPI has converged. Let $\beta^{(i)}$ be the objective value of this final LP solution. Then the error bounds on $V_{\tilde{\pi}}(s)$ (the value function obtained by acting according to the greedy policy $\tilde{\pi}$ w.r.t. $V(s)$)*

[6] As an aside, we also remark that the constraints represent a symbolic dynamic programming step under policy restrictions, thus also enabling a form of successive approximation in first-order MDPs.

derived from plugging $\beta^{(i)}$ in for β in Equation 2.19 hold for all possible finite ground domain instantiations of this FOMDP.

Proof. See Section B.2 of Appendix B.

Thus, FOAPI provides us with an approximate solution method that provides error bounds that hold for *all* possible domain instantiations when convergence is obtained. But what do we do if FOAPI does not converge? Although we don't provide a formal proof, loss bounds should still be possible to guarantee for a policy derived from FOAPI if the algorithm did not converge. In the case of a (ground) MDP, such a loss bound can be computed from the sum of discounted projection errors resulting from each iteration of approximate policy iteration. For an explanation and derivation of this result, we refer the reader to Section 3.2.4 of *Guestrin et al.* [2002].

The final question remaining for FOAPI is how to solve the first-order LP given in Equation 5.21. First, we note that it is simply a mechanical process to write out the exact form of the constraints of Equation 5.21 on every iteration. To achieve a useful constraint form, it is important to explicitly compute the $\exists \vec{x}^*$ operator in each constraint in a similar manner to that done for the $B^A[\cdot]$ operator. Once this is done, we will have an additive form of the constraints similar to those observed for FOALP in Equation 5.15. The question again is how to solve this first-order LP without enumerating all of the constraints as shown previously for FOALP. We tackle this problem next.

5.3 First-order Linear Programs

All linear-value approximation methods require the specification and solution of a *first-order linear program (FOLP)*. 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. However, efficiently solving FOLPs poses a number of difficulties and we work through efficient solutions to these difficulties in this section.

5.3.1 General Formulation

A first-order linear program is specified as follows:

$$\begin{aligned} &\text{Variables: } w_1, \ldots, w_k \ ; \\ &\text{Minimize: } f(w_1, \ldots, w_k) \\ &\text{Subject to: } 0 \geq case_{1,1}(\vec{w}, s) \oplus \ldots \oplus case_{1,n}(\vec{w}, s) \ ; \ \forall \ s \\ &\qquad\qquad \vdots \\ &\qquad\qquad 0 \geq case_{m,1}(\vec{w}, s) \oplus \ldots \oplus case_{m,n}(\vec{w}, s) \ ; \ \forall \ s \end{aligned} \tag{5.22}$$

The variables and objective are as defined in a typical LP, the main difference being the form of the constraints. 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 LPs for FOALP and FOAPI 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 all sums into a single case statement that enumerated all constraints as in Equation 5.17. 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.

5.3.2 First-order Cost Network Maximization

Recalling our discussion of constraint generation for the ground case from Chapter 3, there were two important components that enabled an efficient solution. First, we must be able to rewrite the first-order LP constraints in the following format where the RHS of the constraint assumes the form of a cost network:[7]

$$0 \geq \max_s [case_1(\vec{w}, s) \oplus \ldots \oplus case_n(\vec{w}, s)] \tag{5.23}$$

Second, we need to show that we can efficiently generate the maximizing value within this first-order cost network without enumerating all combinations of assignments to each case statement (which grows exponentially in n).

[7] While we have previously used casemax for maximization over cases, we note that here we are just interested in the maximum value that is possible via a consistent joint selection of case partitions from each case statement in the constraint.

Algorithm 8: $FOMax(C, \langle R_1 \dots R_n \rangle) \longrightarrow \langle S, v \rangle$

input : (1) A set $C = \{case_1, \dots, case_n\}$.
(2) An ordering $\langle R_1 \dots R_n \rangle$ of *all* relations in C.
output : (1) The maximum value v achievable.
(2) A set S of partitions $\{\langle \phi_i, t_i \rangle \in case_i\}$ for $i = 1 \dots n$ s.t. $v = t_1 + \dots + t_n$.
begin
  // Convert C into CNF
  for *($i = 1 \dots n$)* **do**
    foreach $(\langle \phi_i, t_i \rangle \in case_i(s))$ **do**
      Convert ϕ_i to a set of CNF formulae.
  foreach *(relation $R \in \langle R_1 \dots R_n \rangle$ (in order))* **do**
    // Divide C into two sets of cases based on whether they contain R
    $C_R^- := \{case_i | \forall j.(\langle \phi_j, t_j \rangle \in case_i) \wedge \phi_j$ does not contain relation $R\}$
    $C_R^+ := \{case_i | \exists j.(\langle \phi_j, t_j \rangle \in case_i) \wedge \phi_j$ contains relation $R\}$
    Remove all case statements in C_R^+ with their explicit "cross-sum" $\oplus$.

    // C_R^+ is now a single case statement and ϕ_j is a set of CNF formulae for all $\langle \phi_j, t_j \rangle \in C_R^+$
    foreach *($\langle \phi_j, t_j \rangle \in C_R^+$ in order from highest to lowest value)* **do**
      Resolve all clauses in ϕ_j (including new resolvents) on relation R.
      Remove all clauses in ϕ_j containing R.
      if *($\emptyset \in \phi_j$)* **then**
        Discard $\langle \phi_j, t_j \rangle$ and continue with next $\langle \phi_j, t_j \rangle$.
      foreach *($\langle \phi_i, t_i \rangle \in C_R^+$ where $t_i > t_j$)* **do**
        if *($\phi_j \supseteq \phi_i$ (modulo variable renaming))* **then**
          Discard $\langle \phi_j, t_j \rangle$ and continue with next $\langle \phi_j, t_j \rangle$.
    $C := \{C_R^+\} \cup C_R^-$
  $v := 0$; $S := \emptyset$
  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 v, S.
end

To determine the $\max_s$ in this form of the constraints, we provide the $FOMax$ algorithm in Algorithm 8 that efficiently carries out this computation. It is similar to variable elimination [Zhang and Poole, 1994], except that we use first-order ordered resolution in place of propositional ordered resolution. Thus, we term this generalized variable elimination technique used by $FOMax$ to be *relation elimination*. We provide a concrete example of the application of $FOMax$ in Figure 5.1 in the context of the first-order constraint generation algorithm introduced in the next section.

We note that the ordered resolution strategy we are using here is not refutation-complete in that 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 semidecidable. Nonetheless, we note that when the resolution procedure does finitely terminate, then the conjunction of case partition formulae returned by $FOMax$ is guaranteed to be satisfiable as a consequence of the completeness of refutation resolution.

From a practical implementation standpoint, it is necessary to bound the number of resolutions performed at each relation elimination step to prevent non-termination of $FOMax$ due to an infinite number of resolutions. This renders the algorithm refutation-incomplete and thus 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.

5.3.3 First-Order Constraint Generation

We can use the $FOMax$ algorithm to efficiently find the maximal constraint violation when we have constraints of the form in Equation 5.23. This allows us to define the following first-order constraint generation algorithm where we have specified some solution tolerance ϵ:

1. Initialize LP with $\vec{w}^i = \vec{0}$, $i = 0$, and empty constraint set.
2. For each constraint in the cost-network form of Equation 5.23, 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. 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 set, which would be exponential in the number of case statements in the constraint. To demonstrate this, we provide a simple example of finding the most violated constraint in Figure 5.1.

Using first-order constraint generation, we can now efficiently solve the first-order LP from Equation 5.22, which forms a subroutine of both the FOALP and FOAPI algorithms.

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

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

Suppose our last LP solution yielded weights $w_1 = 2$ and $w_2 = 1$. We can efficiently compute the most violated constraint (if one exists) by evaluating the weights in the constraint and applying the FOMax algorithm. We begin by converting all first-order formulae to CNF where $c_1, \ldots, c_6$ are Skolemized constants:

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

Assume the relation elimination order is $BoxIn$, Dst, $TruckIn$. We enter the main loop of FOMax and begin by eliminating the $BoxIn$ relation: we take the cross-sum $\oplus$ of case statements containing $BoxIn$, resolve all clauses in each partition (including all new resolvents), and remove all clauses containing $BoxIn$ (indicated by struck-out text):

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

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 clauses of the value 2 partition are a superset of the clauses of the value 8 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.

Figure 5.1: An example use of FOMAX to find the maximally violated constraint during first-order constraint generation.

Algorithm 9: $BasisGen(\text{FOMDP}, \text{FOALP/FOAPI}, \tau, \text{n}) \longrightarrow \text{B}$

input : (1) A FOMDP specification.
(2) A solution method (FOALP or FOAPI)
(3) a value threshold τ
(4) 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 $A_i(\vec{x})$)* **do**

$B := B \cup$	$\neg\phi_i \wedge \exists \vec{x}\ Regr(\phi_i(do(A_i(\vec{x}), s))) : 1$
$\neg$“	$: 0$

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

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

5.4 Automatic Generation of Basis Functions

The use of linear approximations requires a good set of basis functions that span a space that includes a good approximation to the value function. While some work has addressed the issue of basis function generation [Poupart *et al.*, 2002a; Mahadevan, 2005], no methods have been specifically targeted to generate basis functions exploiting first-order structure for FOMDPs. We consider a basis function generation method that draws on the work of Gretton and Thiebaux [2004], who use inductive logic programming (ILP) techniques to construct a value function from sampled experience. Specifically, they use regressions of the reward as candidate building blocks for ILP-based construction of the value function. This technique has allowed them to generate fully or t-stage-to-go optimal policies for a range of Blocks World problems.

We leverage a similar approach for generating candidate basis functions for use in the FOALP and FOAPI solution techniques. Algorithm 9 provides an overview of our basis func-

tion 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 so we negate it and conjoin it with $Regr(\phi)$.

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

$$bCase_1(s) = w_1 \cdot \begin{array}{|l|} \hline \exists b.\ BoxIn(b, paris, s) : 10 \\ \hline \neg \text{“} \quad : \ 0 \\ \hline \end{array}, \tag{5.24}$$

we would derive the following weighted basis function from $bCase_1(s)$ when considering deterministic action $A_i = unloadS(b^*, t^*)$ during basis function generation:

$$bCase_2(s) = \tag{5.25}$$
$$w_2 \cdot \begin{array}{|l|} \hline \neg[\exists b.\ BoxIn(b, paris, s)] \wedge [\exists c.\ BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] : 1 \\ \hline \neg \text{“} \quad : 0 \\ \hline \end{array}$$

If one examines the form of these two basis functions, the “orthogonality” inherent between the new basis functions and the ones from which they were derived allows for significant computational optimizations. For example, we note that 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 basis functions:

$$bCase_{1,2}(s) = bCase_1(s) \oplus bCase_2(s) \tag{5.26}$$
$$= \begin{array}{|l|} \hline \exists b.\ BoxIn(b, paris, s) \quad : w_1 \cdot 10 \\ \hline \neg[\exists b.\ BoxIn(b, paris, s)] \wedge [\exists c.\ BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, paris, s)] : \quad w_2 \\ \hline \neg \text{“} \quad : \quad 0 \\ \hline \end{array}$$

This style of basis function generation also has many computational advantages for FOALP and FOAPI. To see this, we return to our original discussion concerning the fact that the $B^A[\cdot]$ operator as defined in Equation 5.10 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. Recalling Property 5.1.2, if all basis functions are affected by A, then the backup $B^{A(\vec{x})}[\cdot]$ of each basis function will contain free variables $\vec{x}$ requiring their explicit "cross-sum" to be computed when the $\exists \vec{x}$ operator of $B^A[\cdot]$ is applied. However, in the best case, if the explicit "cross-sum" was already pre-computed for n orthogonal basis functions by merging them into a single case statement of $n+1$ partitions then there will be no explicit "cross-sum" to perform during $B^{A(\vec{x})}[\cdot]$ and the $\exists \vec{x}$ operator can be directly applied without a representational blowup.

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 5.26 that provides an example sum of two orthogonal basis functions. In general, any mutually orthogonal subset of basis functions can be merged in this way.

The policy derivation of FOAPI can be similarly efficient since it relies on the application of the $B^A[\cdot]$ operator. And the exploitation of orthogonal basis functions in the partial computation of the "cross-sum" in the linear-value representation also facilitates the $FOMax$ algorithm since it lowers the worst-case complexity of $FOMax$ where all case statements must be explicitly summed. Thus, for both FOALP and FOAPI, we note that we can exploit orthogonal basis function generation to mitigate exponential space and time scaling in the number of basis functions, where worst-case exponential scaling arises at various points in both solution algorithms due to the need to explicitly compute the "cross-sum" of the linear-value representation.

On a final note, while we do not claim that this method of basis function generation will be appropriate for all domains, we will next demonstrate that it works reasonably well for many stochastic planning problems and that it is relatively efficient in this case.

5.5 Empirical Results

We discuss a number of empirical results on PPDDL planning problems from the ICAPS 2004 [Littman and Younes, 2004b] 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.

5.5.1 ICAPS 2004 Probabilistic Planning Problems

We applied FOALP and FOAPI to the BOXWORLD logistics and BLOCKSWORLD probabilistic planning problems from the ICAPS 2004 IPPC [Littman and Younes, 2004a]. 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 while picking it up or placing it on a stack according to some probability.

We used the Vampire [Riazanov and Voronkov, 2002] theorem prover and the CPLEX 9.0 LP solver in our FOALP and FOAPI implementations and applied the basis function generation algorithm given in Algorithm 9 to a FOMDP version of these PPDDL domains. It is important to note that we generate our solution to the BOXWORLD and BLOCKSWORLD domains offline. Since each of these domains has a universally quanitified reward, our offline solution is for a generic instantiation of this reward following Section 4.7 of Chapter 4. Then at evaluation time when we are given an actual problem instance (i.e., a set of domain objects and initial state configuration), we decompose the value function for each ground instantiation of the reward and execute a policy using the additive decomposition approach also outlined in Section 4.7 of Chapter 4. We do use additional axioms to restrict certain fluent slots to have functional characteristics, e.g. in BOXWORLD, we restrict trucks to only be in one city. Unfortunately, this information restricting legal states is not encoded in PPDDL, but required for our solution (otherwise additional constraints for illegal states are generated and these adversely influence the basis function weights). We do not enhance or otherwise modify our offline solution once given actual domain information although this would be an avenue for future research.

We set an iteration limit of 7 in our offline basis function generation algorithm and recorded the running times per iteration of FOALP and FOAPI; these are shown in Figure 5.2. There appears to be exponential growth in the running time as the number of basis functions increases.

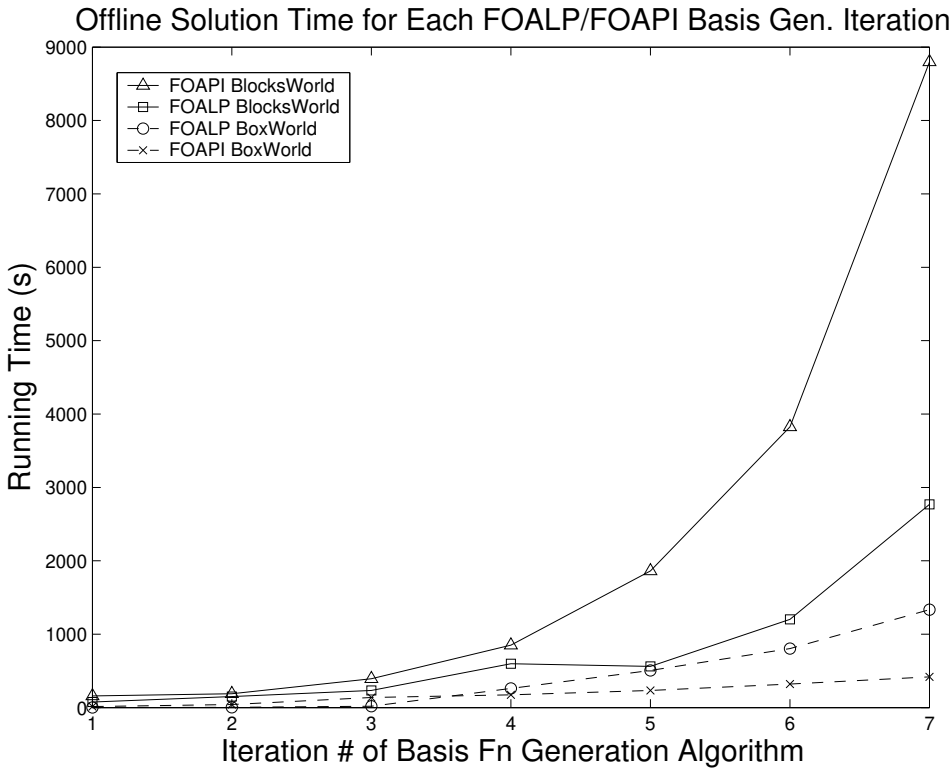


Figure 5.2: FOAPI and FOALP solution times for the BOXWORLD and BLOCKSWORLD problems vs. the iteration of basis function generation.

However, we note that if we were not using the "orthogonal" basis function generation, we would not get past iteration 2 of basis function generation due to the prohibitive amount of time required by FOALP and FOAPI in this case (> 10 hours). Consequently, we can conclude that our basis function generation algorithm and optimizations have substantially increased the number of basis functions for which FOALP and FOAPI remain viable solution options. In terms of a comparison of the running times of FOALP and FOAPI, it is apparent that each performs better in different settings. In BOXWORLD, FOAPI takes fewer iterations of constraint generation than FOALP and thus is slightly faster. In BLOCKSWORLD, the policies tend to grow more quickly in size because the Vampire theorem prover has difficulty refuting inconsistent partitions on account of the heavy use of equality in this FOMDP domain. This impacts not only the solution time of FOAPI, but also its performance as we will see next.

We applied the policies generated by the FOALP and FOAPI versions of our basis function function generation algorithm to three BOXWORLD and five BLOCKSWORLD problem instances from the ICAPS 2004 IPC. We compared our planning system to the three other top-

Problem	Competing Probabilistic Planners						
	NMRDPP	mGPT	Humans	Classy	FF-Replan	FOALP	FOAPI
bx c10 b5	438	184	419	376	425	433	433
bx c10 b10	376	0	317	0	346	366	366
bx c10 b15	0	–	129	0	279	0	0
bw b5	495	494	494	495	494	494	490
bw b11	479	466	480	480	481	480	0
bw b15	468	397	469	468	0	470	0
bw b18	352	–	462	0	0	464	0
bw b21	286	–	456	455	459	456	0

Table 5.1: Cumulative reward of 5 planning systems and the FOALP and FOAPI (100 run avg.) on the BOXWORLD and *Blocks World* 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.

performing planners on these domains:[8] *NMRDPP* is a temporal logic planner with human-coded control knowledge [Gretton *et al.*, 2004]; *mGPT* is an RTDP-based planner [Bonet and Geffner, 2004]; *(Purdue-)Humans* is a human-coded planner, *Classy* is an inductive 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 5.1.

We make four overall observations w.r.t. these results:

1. FOALP and FOAPI produce the same basis function weights and therefore the same policies for the BOXWORLD domain.

2. We only used 7 iterations of basis function generation and 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 both FOALP and FOAPI failed on this instance.[9]

3. Due to aforementioned problems with the inability of FOAPI to detect inconsistency of policy partitions in the BLOCKSWORLD domain, its performance is severely degraded on these problem instances in comparison to FOALP. FOALP does not use a policy representation and thus does not encounter these same problems.

[8]The names we use for the planners are intended to intuitively denote their approach are not necessarily their given names. See the associated references for details.

[9]We could not increase the number of iterations to 8 due to memory constraints.

4. It is important to note that in comparing FOALP and FOAPI 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 fully autonomous stochastic planners were mGPT and Classy, and FOALP performs comparably to both of these planners and outperforms them by a considerable margin on a number of problem instances.

5.5.2 ICAPS 2006 Probabilistic Planning Problems

We now present results for FOALP on the ICAPS 2006 probabilistic planning competition. For these problems, we do not present results for FOAPI. As with the ICAPS 2004 competition problems, the difficulty with FOAPI is that its policy representations tend to grow combinatorically as more basis functions are added during basis function generation until the policy size outstrips the ability of the theorem prover to identify inconsistent policy partitions. This severely degrades performance and prevents scaling to a number of basis functions required to obtain reasonable performance on these planning problems.

Consequently, we present results for FOALP applied to three domains from the competition: BLOCKSWORLD, TIREWORLD, and ELEVATORS. We've already covered the basic description of BLOCKSWORLD, one of the main differences in this competition being that the table was not considered to be a block and thus there were additional actions for picking up and putting blocks down on 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 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 exists at the given 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

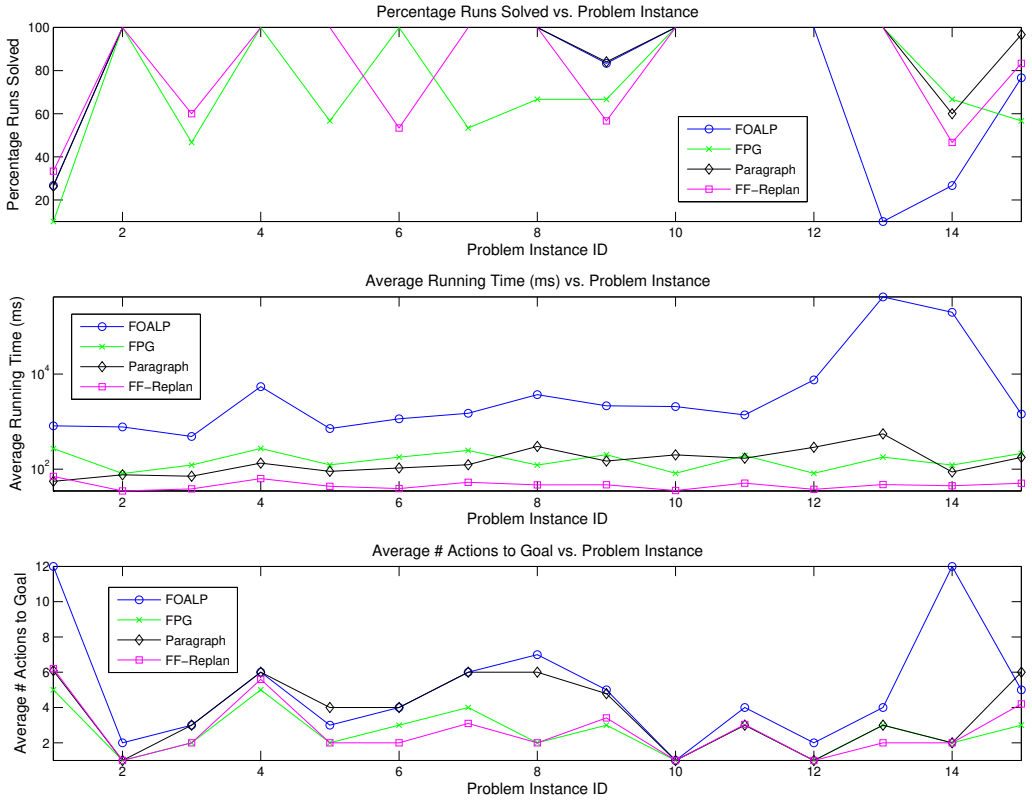


Figure 5.3: The performance of five planners on the ICAPS 2006 TIREWORLD planning competition problem. The domain instantiations become larger as the problem instance ID increases.

known positions and the goal is to retrieve them all.

Our solution approach is identical to that used for the ICAPS 2004 problems. To recap, we used the Vampire theorem prover and the CPLEX 9.0 LP solver in our FOALP implementation and applied the basis function generation algorithm given in Figure 9 to a FOMDP version of these PPDDL domains. For the BOXWORLD and ELEVATORS problems that had universal rewards, we additively decompose their solution according to Section 4.7 of Chapter 4. As done before, we specified additional background theory axioms describing constraints on functional slots of fluents that were required to obtain reasonable solutions (e.g., only one block could be directly stacked on another block in BLOCKSWORLD). Without these domain axioms, FOALP erroneously generates constraints for illegal states and these extra constraints adversely affect the solutions obtained.

In Figures 5.3, 5.4, and 5.5, we provide data for FOALP and competing planners that spec-

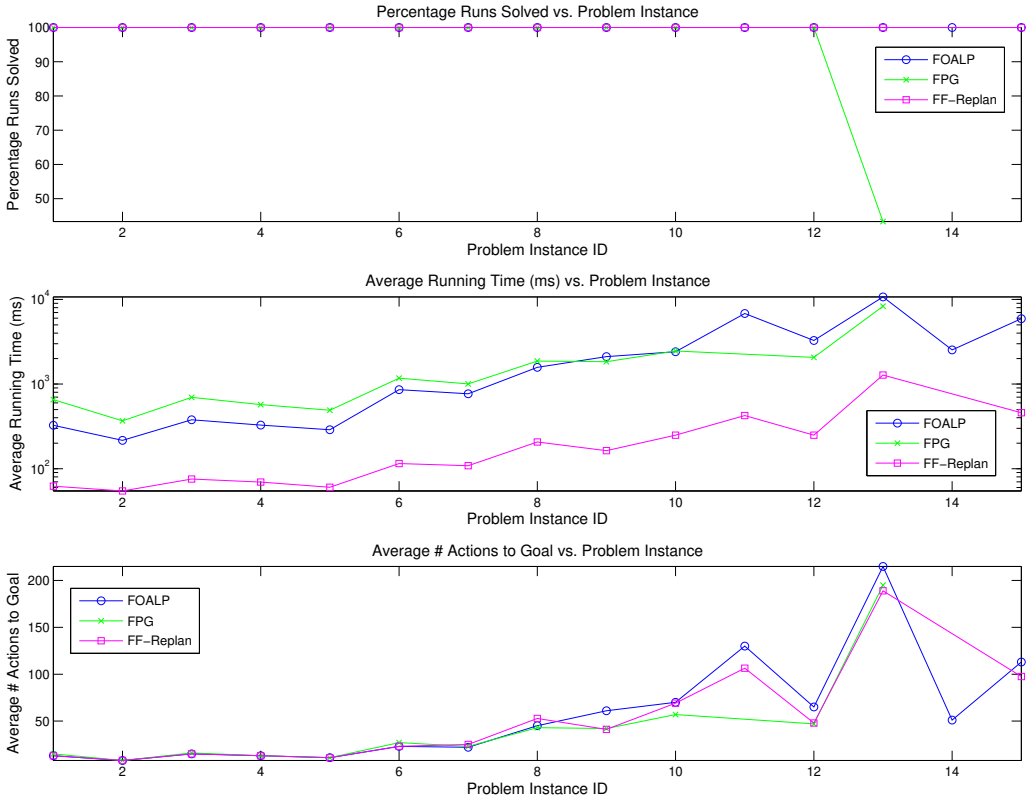


Figure 5.4: The performance of five planners on the ICAPS 2006 ELEVATORS planning competition problem. The domain instantiations become larger as the problem instance ID increases.

ifies the number of problem instances solved, the online solution generation time[10], 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: (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 SPUDD-style ADD-based dynamic programming [Hoey *et al.*, 1999] with reachability constraints based on initial state knowledge; (3) *Paragraph* [Little, 2006], which uses an 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 of the 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.

[10]Offline solutions were permitted 4 hours per problem to remain within the overall competition time limit of 24 hours. However, since the offline solution time can be amortized over an indefinite number of domain instances, we do not report this in the online time.

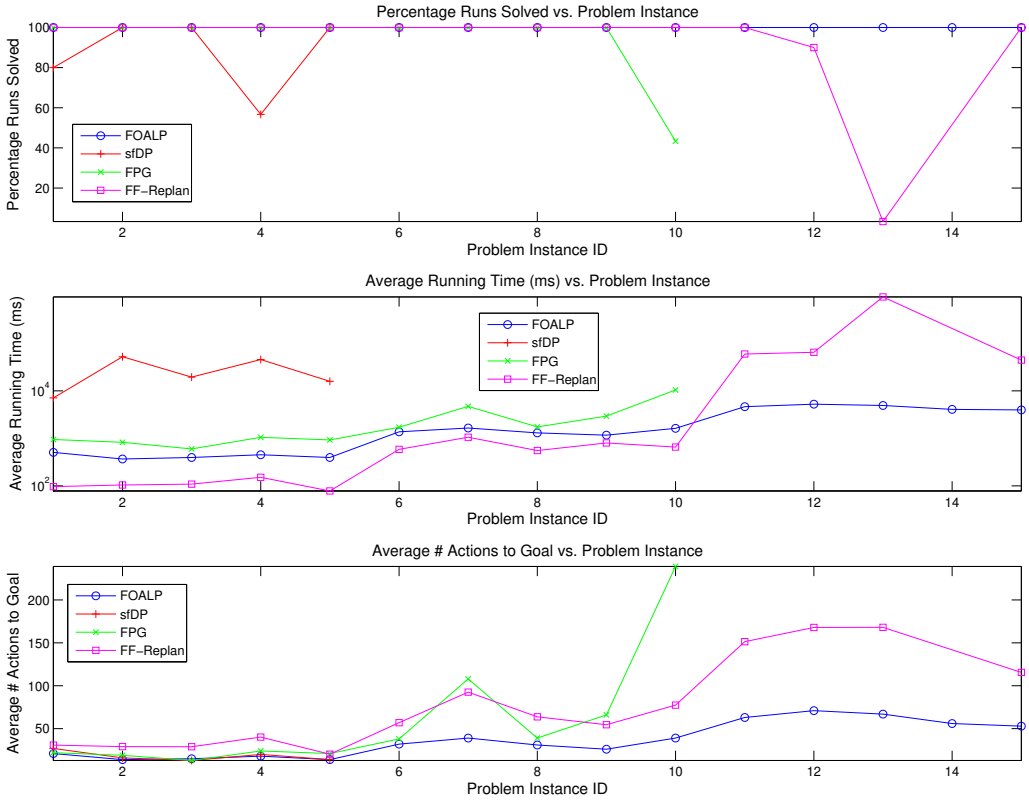


Figure 5.5: The performance of five planners on the ICAPS 2006 BLOCKSWORLD planning competition problem. The domain instantiations become larger as the problem instance ID increases.

The results vary by problem, so we explain each in turn. In TIREWORLD, FOALP's policy allowed it to solve most problems although we note that according to the average number of actions to the goal, its policy was suboptimal in comparison to other planners. In this case, 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. We note that 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 slightly 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, and reaching the goal with the

fewest actions. 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 Summary

In this chapter, we introduced a first-order generalization of linear-value approximation using a linear combination of first-order basis functions. Then we introduced generalizations of the backup operators used to exploit linearity in the dynamic programming backup of this linear-value representation. We next introduced the linear-value approximation techniques of FOALP and FOAPI for FOMDPs and cast them as the setup and solution of specialized first-order linear programs. We introduced the general first-order linear program (FOLP) formulation that captures both the FOALP and FOAPI solutions along with the solution to a FOLP via constraint generation methods. Constraint generation methods required the efficient solution of a first-order cost network and we introduced the $FOMax$ algorithm for this purpose. $FOMax$ essentially performs a lifted version of variable elimination that we term *relation elimination*. Finally, we introduced a technique for basis function generation that can be exploited efficiently in the linear-value function representation, the backup operators, the $FOMax$ algorithm, and the policy representation used by FOAPI.

Having solved all of these problems, we applied our techniques to problems from the ICAPS 2004 and ICAPS 2006 International Probabilistic Planning Competitions, comparing to a number of other planners in the process. While FOAPI tended to perform poorly due to a blowup in its policy representation and inability to scale to a reasonable number of basis functions on most problems, FOALP fared comparatively well when using the additive goal decomposition technique for universal rewards from Chapter 4, Section 4.7. In some cases, FOALP was able to exploit first-order structure in the domain and in terms of success rate and the number of actions to the goal, it could often perform on par or better than hand-coded policies, learning-based approaches, constraint-based approaches, and deterministic replanning approaches. The key to FOALP's success in these cases is that it was the only first-order planner and thus could exploit relational structure that the other ground planners could not. And perhaps the key to extending its success is to combine it with traditional online techniques used by the other planners in order to enhance its performance.

In addition, it would also be useful to revisit the computation of the objective in the first-

order LP for FOALP since it was an approximated version of the uniform state-relevance weighting assumption of the ground ALP objective. There are two reasons to revisit this computation: (1) as suggested by de Farias and Van Roy [2003], there may be better alternatives to uniform state relevance weighting, and (2) there may be better FOALP approximations of the objective than the approach we have taken here. While the current objective allows for efficient computation and appears to yield reasonable performance in practice, even better results may be attainable if we *adapt* the importance of each weight in the FOALP objective. For example, we might attempt to give more importance to weights representing more frequently visited case partitions of basis functions by computing the expected occupany probability of each basis function partition [de Farias and Van Roy, 2003].

Notwithstanding additional enhancements, our present linear-value function approximation results are very encouraging in general for the lifted first-order approach to planning that we motivated in Chapter 4. However, in addition to further practical research to enhance the results of FOALP and related methods, there are still entire classes of problems that our first-order approach presented so far can neither represent domain-independently, nor solve efficiently. For example, when we introduced factored MDPs in Chapter 4, we touted their ability to represent factored action spaces and additive rewards. Unfortunately, in the case of FOMDPs, when these factored actions and additive rewards scale with the domain size, we do not have the representational machinery to specify such a problem without resorting to domain-specific representations. The SYSADMIN problem is an ideal example of this — every computer could independently fail or reboot on each time step and the reward scaled additively with the total number of computers that were running. FOMDPs do not currently have the ability to represent this indefinite factored structure and thus in the next chapter, we turn to representing these problems in a domain-independent manner and extending the solution techniques from this chapter to efficiently solve them.

Chapter 6

Factored First-order MDPs

So far we have defined the factored MDP model in Chapter 3 and the FOMDP model in Chapter 4. However, it is interesting to note that many factored MDPs cannot be compactly specified in the FOMDP formalism previously defined where we use the term *compact* throughout this chapter to mean "the size of the representation is independent of the size of any particular domain instantiation". For example, we formalized SYSADMIN as a factored MDP in Chapter 3, but we have not yet defined it as a FOMDP. There are two difficulties in doing this: First, SYSADMIN has an additive reward that scales with the number of computers, yet the FOMDP formalism does not have the means for compactly specifying a sum that scales with the domain size. Second, SYSADMIN has a factored action space that does not make a strong frame assumption — *every* computer can independently reboot or crash as a result of the action executed. To model this compactly, we need to factor action effects into a number of independent aspects[1] that scales with the domain size and specify a factored joint probability distribution over aspects that exploits the (probabilistic) independence inherent in their definition. In the current FOMDP formalism, the representation of this joint distribution cannot be specified compactly.

In general, SYSADMIN is just a motivating example for a much larger class of relationally structured MDPs that we refer to as *factored FOMDPs*. As another example of a factored FOMDP, we can easily imagine a simple factored variant of BOXWORLD that we call F-BOXWORLD, where a box loaded on a truck may independently "drop" off the truck at each time step with some small probability if it is not otherwise explicitly loaded or unloaded, and where the "drop" of each box is probabilistically independent. While we will still assume the

[1] Roughly defined, an aspect is an independent outcome of an action as used previously in Chapter 4.

same existentially quantified reward as in the original BOXWORLD (c.f., Equation 4.9), the need to model the additional "drop" aspects for each box when a stochastic action is executed forces us to consider a transition distribution whose outcome/event space scales exponentially with the domain size, hence whose representation using deterministic Nature's choice actions scales exponentially with domain size. In general, modeling exogenous events in FOMDPs using a number of action aspects that scales linearly in the domain size will have a Nature's choice deterministic action representation that scales exponentially in the number of aspects and thus exponentially in the domain size.

The factored FOMDP formalism that we will introduce allows us to compactly represent some FOMDPs such as SYSADMIN and F-BOXWORLD that cannot be compactly represented as FOMDPs. To define the class of FOMDPs that may be compactly represented as factored FOMDPs, we say that a FOMDP representation *scales with domain size* if the space of its representation as a FOMDP from Chapter 4 is proportional to the size of a domain instantiation. Then in general, a FOMDP is a candidate for compact representation as a factored FOMDP if it has at least one of the following two properties:

(a) An indefinite additive reward whose number of additive components scales with the domain size; OR

(b) An indefinite number of independent action aspects that scales with the domain size.

In order to generalize the FOMDP representation of Chapter 4 to factored FOMDPs, we introduce the following two representational extensions to the FOMDP:

1. We introduce sum and product aggregators to specify additive rewards and joint transition distributions over aspects of stochastic actions that scale with domain size. The sum aggregator can often compactly represent property (a) while the product aggregator partially addresses property (b).

2. We define a factored model of action effects that allows a stochastic action executed by the agent to decompose into independent aspects. We then specify a joint distribution over deterministic sub-actions for each of these aspects and we introduce modifications to the situation calculus to efficiently handle (decision-theoretic) regression with these sub-actions. Together with the product aggregator, this is often sufficient to compactly represent property (b).

We will discuss the impact of both of these representational enhancements on various solution methods at length in this chapter. In general, we *could* always obtain a universal solution algorithm for factored FOMDPs by compiling them into a FOMDP[2] representation for a specific domain size, thus allowing the direct application of the FOMDP solution methods of Chapters 4 and 5. However, converting from the compact representation of a factored FOMDP satisfying properties (a) or (b) to a FOMDP representation for a specific domain size would incur a representational blowup at least linear in the domain size if only (a) held (due to the linear number of additive reward components) and at least exponential in the domain size if property (b) held (due to the exponential number of Nature's choice deterministic actions as previously discussed). But this blowup just refers to the size of the FOMDP representation and we have not even begun to consider the computational impact of the representation size on the time complexity of its solution. At the very least we can compute a simple lower bound on the time and space complexity of solution methods that convert a factored FOMDP to a FOMDP for a specific domain size since the FOMDP representation must be provided as input to these algorithms; thus, this lower bound is at least linear in the domain size if only property (a) holds and exponential in the domain size if property (b) holds.

Given these unencouraging lower bounds on the time and space complexity of solving a factored FOMDP by converting it to a FOMDP and using the previously defined solution methods of Chapters 4 and 5, one objective in the specification of factored FOMDP algorithms would be to demonstrate potential cases where the time and space complexity of a factored FOMDP solution is sub-linear in the domain size when only property (a) holds, sub-exponential in the domain size when property (b) holds, or potentially independent of domain size in special cases of (a) or (b) — results that are all impossible to obtain for methods that solve a factored FOMDP by converting it to a FOMDP. In general, when we say that a factored FOMDP solution algorithm is *efficient* for a given problem, we imply that it meets one of these three criteria. With this goal of efficiency in mind, we define (approximate) solutions for factored FOMDPs that generalize the symbolic dynamic programming and linear-value approximation techniques described in Chapters 4 and 5. To demonstrate the solution techniques that will be needed to efficiently handle specific types of problem structure, we will work through various steps of our solution algorithms for the SYSADMIN and F-BOXWORLD problems; the example representation and factored FOALP solution of SYSADMIN appeared previously in Sanner and Boutilier [2007].

[2]Throughout this chapter, we will use the unqualified use of the term "FOMDP" to mean the FOMDP representation of Chapter 4 and the term "factored FOMDP" to refer to the formalism provided in this chapter.

However, in generalizing the symbolic dynamic programming and linear-value approximation solutions from FOMDPs to factored FOMDPS, we remark that our investigation into these solution methods is only in its initial stages and our results w.r.t. SYSADMIN and F-BOXWORLD are anecdotal at best in reference to a general solution for factored FOMDPs. However, it is important to note that an efficient, exact solution algorithm for all factored FOMDPs is impossible. A negative result of Jaeger [2000] implies that there is no algorithm that can perform exact inference in the DBN representation underlying the factored FOMDP that can avoid exponential time complexity in the domain size in the worst case. We discuss this result in our concluding remarks along with general suggestions for how we might cope with this worst case in practice.

6.1 Factored FOMDP Representation

In order to specify indefinitely scaling additive rewards and transition distributions, we begin by introducing the sum and product aggregators. Then we proceed to formalize stochastic actions with multiple independent aspects and deterministic sub-action outcomes as well as a factored transition distribution based on this action model. Following this, we introduce the situation calculus machinery and assumptions required to efficiently handle sub-actions in the situation calculus. Throughout this discussion, we motivate each construct using the SYSADMIN problem. Finally, we conclude by applying the same modeling formalism to specify a factored FOMDP model for F-BOXWORLD.

6.1.1 Sum and Product Aggregators

We introduce sum aggregators that are similar in purpose and motivated by the count aggregators of Guestrin *et al.* [2003] that permit the specification of indefinite-length sums over all instantiations of a case statement for a given object class. Likewise, we introduce the novel product aggregator that performs a similar role to the sum aggregator, except for products rather than sums. We can easily motivate the sum aggregator by attempting to represent the reward in the SYSADMIN problem. Given a domain object class $Comp$ of n computers $\{c_1, \ldots, c_n\}$, we know that the SYSADMIN reward scales with the number of computers that are up and running:

$$rCase(s) = \begin{array}{|l|}\hline Up(c_1, s) \;:\; 1 \\ \hline \neg Up(c_1, s) : \; 0 \\ \hline \end{array} \oplus \begin{array}{|l|}\hline Up(c_2, s) \;:\; 1 \\ \hline \neg Up(c_2, s) : \; 0 \\ \hline \end{array} \oplus \cdots \oplus \begin{array}{|l|}\hline Up(c_n, s) \;:\; 1 \\ \hline \neg Up(c_n, s) : \; 0 \\ \hline \end{array} \tag{6.1}$$

However, we note two problems with representing the SYSADMIN reward in this manner. First, the number of case statements in the reward scales with the domain size and thus will not be an efficient representation for large domains, not to mention the impact of this representation on the time and space complexity of the solution algorithms. Second, this reward specification is domain-specific in that it refers to exactly n computers; ideally, we would prefer to have both a compact and domain-independent specification of reward for SYSADMIN. This would then directly facilitate the solution of factored FOMDPs for indefinitely large domains, something that is otherwise impossible with a domain-specific representation.

To permit the specification of indefinite length sums and products, we introduce sum and product aggregators that are defined in terms of the $\oplus$ and $\otimes$ operators where we use a meta-logical notation that can only be expanded given a concrete domain instantiation:[3]

$$\sum_{c \in C} case(c, s) = case(c_1, s) \oplus \cdots \oplus case(c_n, s) \tag{6.2}$$

$$\prod_{c \in C} case(c, s) = case(c_1, s) \otimes \cdots \otimes case(c_n, s) \tag{6.3}$$

While the sum and product aggregator can easily be expanded for small domain instantiations, there is generally no finite representation for indefinitely large n due to the piecewise constant nature of the case representation. That is, even if the $\oplus$/$\otimes$ is explicitly computed, there may be an indefinite number of distinct constant values to represent in the resulting case.

Using the sum aggregator, we can now easily define the SYSADMIN reward in a domain-independent manner:

$$rCase(s) = \sum_{c \in Comp} \left(\begin{array}{|l|} \hline Up(c, s) \ : \ 1 \\ \hline \neg Up(c, s) : \ 0 \\ \hline \end{array} \right) \tag{6.4}$$

Later we will see how we can define the transition probability in SYSADMIN using the product aggregator.

[3]Here we assume a generic object class $C = \{c_1, \ldots, c_n\}$, not necessarily computers.

6.1.2 Operations with Sum and Product Aggregators

Following are various properties of the $\sum_c$ and $\prod_c$ aggregators that we can use during the solution of our FOMDP. Since we know that

$$Regr(case_1(do(a,s)) \oplus case_2(do(a,s))) \quad = \quad Regr(case_1(do(a,s))) \oplus Regr(case_2(do(a,s)))$$

and likewise for $\otimes$ owing to the logical definition of the case statement and the properties of $Regr$, we can easily infer the following:

$$Regr\Big(\sum_c case(c, do(a,s))\Big) \quad = \quad \sum_c Regr\Big(case(c, do(a,s))\Big) \tag{6.5}$$

$$Regr\Big(\prod_c case(c, do(a,s))\Big) \quad = \quad \prod_c Regr\Big(case(c, do(a,s))\Big) \tag{6.6}$$

Since the sum aggregator is defined in terms of $\oplus$, the standard properties of commutativity, associativity, and distributivity of $\otimes$ over $\oplus$ hold. Likewise, since the product aggregator is defined in terms of $\otimes$, the standard properties of commutativity and associativity hold. In this chapter, we will make use of the following equivalences that can be easily derived as corollaries of these properties:

$$\sum_{c\in C} case_1(c,s) \oplus \sum_{d\in C} case_2(d,s) \quad = \quad \sum_{c\in C}[case_1(c,s) \oplus case_2(c,s)] \tag{6.7}$$

$$case_1(s) \otimes \sum_{c\in C} case_2(c,s) \quad = \quad \sum_{c\in C}[case_1(s) \otimes case_2(c,s)] \tag{6.8}$$

$$[case_1(s) \oplus case_2(s)] \otimes \prod_{c\in C} case_3(c,s) \quad = \quad case_1(s) \otimes \prod_{c\in C} case_3(c,s) \oplus case_2(s) \otimes \prod_{d\in C} case_3(d,s) \tag{6.9}$$

6.1.3 Factored Stochastic Actions

Recalling our original FOMDP stochastic action representation from Chapter 4, Section 4.3.3, we decomposed stochastic "agent" actions $A(\vec{x})$ into a *collection* of deterministic Nature's choice actions $n_1(\vec{x}), \cdots, n_k(\vec{x})$, each corresponding to a possible outcome of the stochastic action. We then used a case statement to specify a distribution $P(n_j(\vec{x}), A(\vec{x}))$ according to which Nature may choose a deterministic action from this set whenever the stochastic action is executed.

This approach assumes that once Nature chooses a deterministic action, it must commit to all effects incurred by that deterministic action. Let us consider modeling SYSADMIN in this FOMDP framework assuming some fixed number of computers n. When the agent reboots a computer in SYSADMIN, the status of each computer may evolve independently of the other computers. Thus, given n computers, each with 2 possible status configurations (up or not), this leads to 2^n deterministic actions since *each* distinct joint outcome must be specified with a separate deterministic action. While we cannot avoid the fact that SYSADMIN has 2^n deterministic actions, we can potentially avoid explicitly enumerating all of these joint deterministic actions by exploiting the fact that the stochastic reboot action decomposes into a number of independent aspects corresponding to direct and exogenous effects that act individually and independently (in a probabilistic sense) on each computer.

In this section, we introduce a factored approach to representing stochastic actions and their associated transition distributions as well as the necessary situation calculus machinery to integrate them into our FOMDP framework.

Aspects and Deterministic Sub-actions

When an action has multiple independent effects, a more compact representation than a direct joint enumeration of these effects would be to specify independent *aspects* of a stochastic action in a generalization of the factored PSTRIPS operators of Dearden and Boutilier [1997].[4] In this framework, we can independently specify local probability distributions over each aspect and combine them into a joint factored distribution using a DBN-like representation — albeit a DBN over an indefinite number of aspects that generally scale as a function of the domain size.

To justify the slightly cumbersome notation that we introduce in this section, we first begin with a motivating example. In SYSADMIN, if there are n computers then the $reboot(Comp : x)$ action that reboots computer x will have n aspects that we denote generically as $reboot_1(Comp : x, Comp : y)$ where y refers to the other computers that could be affected as a result of this action. Specifically, the aspects $reboot_1(x, c_1), \ldots, reboot_1(x, c_n)$ each represent the local effects of stochastic action $reboot(Comp : x)$ on each individual computer c_i. We could imagine a second class of aspects $reboot_2(Comp : x, Comp : z)$ specifying whether a computer z spontaneously catches fire when computer x is rebooted (as may be appropriate in a realistic model of Dell laptops).

[4] Motivated by this earlier work and in anticipation of factored FOMDPs, we intentionally used the term aspect to identify probabilistically independent sets of action effects in the PPDDL representation of Chapter 4.

In a general framework, we consider that each stochastic action $A(\vec{x})$ may have independent aspects that we denote by $A_1(\vec{x}, \vec{y}), \ldots, A_p(\vec{x}, \vec{y})$ where $\vec{x}$ represents the stochastic action parameters, $\vec{y}$ is optional and represents additional domain objects that are individually affected by each aspect, and the action subscripts $1 \ldots p$ denote that we have p different classes of aspects. Altogether, if $\vec{y}$ is non-empty, then we will have a total of $p \cdot |\vec{y}|$ aspects for stochastic action $A(\vec{x})$ where $|\vec{y}|$ indicates the total number of distinct assignments to the object domains of variables in $\vec{y}$.

We refer to Nature's deterministic choices for aspects to be *sub-actions*. In a general framework, for each aspect $A_i(\vec{x}, \vec{y})$ we can specify a set $N_i(\vec{x}, \vec{y})$ of deterministic Nature's choice sub-actions as $N_i(\vec{x}, \vec{y}) = \{n_{i,1}(\vec{x}, \vec{y}), \ldots, n_{i,q}(\vec{x}, \vec{y})\}$ where $q \geq 2$. Often, we use a random variable notation for Nature's choice where $n_i(\vec{x}, \vec{y}) \in N_i(\vec{x}, \vec{y})$. Then we can represent a distribution over Nature's choice deterministic sub-actions for each aspect as a case statement $P(n_i(\vec{x}, \vec{y}) | A_i(\vec{x}, \vec{y}), s)$ expression in a manner similar to Section 4.3.3.

For example, in SYSADMIN, we can specify two possible deterministic Nature's choice sub-actions for each aspect $reboot_1(x, c_i)$: $n_1(x, c_i) \in= \{rebootS(x, c_i), rebootF(x, c_i)\}$ where $rebootS(x, c_i)$ causes c_i to be running and $rebootF(x, c_i)$ causes c_i to be crashed. Following our notation from Section 4.3.3, we can specify an instance of $P(n_i(\vec{x}, \vec{y}) | A_i(\vec{x}, \vec{y}), s)$ for aspect $reboot_1(x, c_i)$ and its deterministic sub-action outcomes as $pCase(n_1(x, c_i) | reboot_1(x, c_i), s)$:[5]

$$pCase(rebootS(c_i) | reboot_1(x, c_i) \wedge x = c_i, s) = \boxed{\top \ : \ 1} \tag{6.10}$$

$$pCase(rebootS(c_i) | reboot_1(x, c_i) \wedge x \neq c_i, s) = \tag{6.11}$$

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

Here we see that the probability that a computer will be running if it was explicitly rebooted is 1. Otherwise, the probability that a computer is running depends on its previous status and the proportion of computers with incoming connections that are running. The probability of the

[5]We use the predicate $Conn(c_j, c_i)$ to indicate that there is a directed connection from computer c_j to c_i.

failure outcome $rebootF(c_i)$ is just the complement of the success case:

$$pCase(rebootF(c_i)|reboot_1(x, c_i), s) = \boxed{\top : 1} \ominus pCase(rebootS(c_i)|U(c_i)) \tag{6.12}$$

At this point, the reader familiar with the factored PSTRIPS operators of Dearden and Boutilier [1997] may note the absence of discriminants in the framework presented here. In fact, we do have discriminants, but for consistency with the FOMDP framework we model them with the case representation of transition probabilities. In the factored PSTRIPS framework, probabilities were restricted to be constants and thus discriminants were needed to distinguish between the different effect probabilities that would arise for different states and actions. However, in our specification above, these state- and action-dependent probabilities are specified directly in the representation of $P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$. Thus, for a correct probability specification, we will require that the same properties that held for the discriminants in the factored PSTRIPS discriminant representation also hold for our general $P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$ representation, namely that the probability distribution over all q sub-action outcomes of an aspect sum to 1:

$$\forall i, \vec{y} \left[\bigoplus_{j=1}^{q} P(n_{i,j}(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s) = \boxed{\top : 1} \right] \tag{6.13}$$

In addition, each $P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$ should be a disjoint partitioning of state space such that no two case partitions ambiguously assign multiple probabilities to the same state.

Joint Actions and Transition Probabilities

Now that we have specified the distribution over deterministic sub-action outcomes for individual aspects, we need to specify how these sub-actions come together to specify a joint deterministic action. This is actually quite simple. Since we will define the effects of a joint deterministic action to be the *union* of effects for each of its sub-actions in the next section, we can specify a deterministic joint action $a \in N_A(\vec{x})$ as the *associative-commutative* composition $\circ$ of Nature's choice deterministic sub-action outcomes for *all* aspects $A_i(\vec{x})$ of a stochastic action $A(\vec{x})$. Following previous notation, we can write $N_A(\vec{x})$ as the following where $\dot{\vec{y}}$ stands for a specific constant substitution for $\vec{y}$ and $\dot{\vec{y}}_j$ stands specifically for the jth possible variable substitution:

$$N_A(\vec{x}) = \{N_1(\vec{x}, \dot{\vec{y}}_1) \times \cdots \times N_1(\vec{x}, \dot{\vec{y}}_{|\vec{y}|}) \times \cdots \times N_p(\vec{x}, \dot{\vec{y}}_1) \times \cdots \times N_p(\vec{x}, \dot{\vec{y}}_{|\vec{y}|})\} \tag{6.14}$$

Now, assuming a notational equivalence between a set of terms and their associative-commutative composition using $\circ$, then for all $a \in N_A(\vec{x})$, a is a composition of one sub-action from each ground aspect of stochastic action $A(\vec{x})$.

For example, in the SYSADMIN domain, if $n = 4$ and we have 4 aspects for $reboot(x)$, namely $reboot_1(x, c_1), \ldots, reboot_1(x, c_4)$, then one deterministic joint action a could be the following:

$$a = rebootS(x, c_1) \circ rebootF(x, c_2) \circ rebootF(x, c_3) \circ rebootS(x, c_4) \tag{6.15}$$

At this point one might ask whether different sub-actions within a joint deterministic action could interfere with each other. Once we have defined the effects for sub-actions in the next section, this will be equivalent to the question of whether sub-actions can have inconsistent effects. This is an important issue and one that we will address once we formalize the semantics of deterministic joint actions within the situation calculus framework. However, we must first specify the probability distribution over joint actions as this will be an important component in our guarantee of consistency.

Our example for a in Equation 6.15 was just one of many possible joint deterministic actions for one domain instantiation of SYSADMIN, and in a general (indefinitely) factored FOMDP, we will need to specify a joint distribution over all possible joint deterministic outcomes of a stochastic action. Fortunately, we can exploit knowledge of the independence of aspects to define the probability of any joint deterministic action a in terms of its constituent sub-actions $n_i(\vec{x}, \dot{\vec{y}})$, where we assume for simplicity that each aspect depends on the same $\vec{y}$:

$$P(a|A(\vec{x}), s) = \prod_{\vec{y} \in \{\dot{\vec{y}}_1, \ldots, \dot{\vec{y}}_{|\vec{y}|}\}} \prod_{i=1}^{p} P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s) \tag{6.16}$$

As long as the properties specified in Equation 6.13 hold for each $P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$, then we can obtain the following proposition:

Proposition 6.1.1. *$P(a|A(\vec{x}), s)$ defines a proper probability distribution over a, i.e.,*

$$\sum_{a \in N_A(\vec{x})} P(a|A(\vec{x}), s) = \boxed{\top : 1}. \tag{6.17}$$

Proof. See Section B.3 of Appendix B.

Thus, for SYSADMIN, we can now easily specify a compact joint distribution over all

possible deterministic action outcomes of $reboot(x)$. For a concrete example of the probability of the joint deterministic action a specified in Equation 6.15, we would obtain the following probability specification where the actual numerical probabilities follow from the evaluation of Equations 6.10, 6.11, and 6.12 for the particular state properties of s:

$$\begin{aligned} &pCase(a|reboot(x), s) = \\ &\quad pCase(rebootS(c_1)|reboot_1(c_1), s) \otimes pCase(rebootF(c_2)|reboot_1(c_2), s) \\ &\otimes pCase(rebootF(c_3)|reboot_1(c_3), s) \otimes pCase(rebootS(c_4)|reboot_1(c_4), s) \end{aligned}$$

Joint Actions and the Situation Calculus

Up to this point we have ignored the actual semantics of Nature's choice deterministic sub-actions and joint actions, but we now address these issues. To begin, we can directly define the semantics of deterministic joint actions with effect axioms as we did in Section 4.2.2, except that we must take care to define effect axioms for *all* possible joint actions and effects. For example, in SYSADMIN with $n = 2$ computers, we would obtain the following complete set of 4 effect axioms (which can be easily converted to the normalized effect axiom form of Equations 4.1 and 4.2):

$$\begin{aligned} a &= rebootS(c_1) \circ rebootS(c_2) \supset Up(c_1, do(a, s)) \wedge Up(c_2, do(a, s)) \\ a &= rebootS(c_1) \circ rebootF(c_2) \supset Up(c_1, do(a, s)) \wedge \neg Up(c_2, do(a, s)) \\ a &= rebootF(c_1) \circ rebootS(c_2) \supset \neg Up(c_1, do(a, s)) \wedge Up(c_2, do(a, s)) \\ a &= rebootF(c_1) \circ rebootF(c_2) \supset \neg Up(c_1, do(a, s)) \wedge \neg Up(c_2, do(a, s)) \end{aligned} \tag{6.18}$$

While this representation of the effect axioms correctly defines the action semantics for SYSADMIN, it is not compact since the number of distinct joint actions scales exponentially with the domain size. In general, for a SYSADMIN domain with n computers, we would have 2^n joint actions, not to mention the conjunction of n effects we would have to specify for *each* joint action. Clearly, a solution approach based on the explicit enumeration of all effect axioms for all joint actions in a factored FOMDP such as SYSADMIN would lead to an algorithm whose complexity scales exponentially in the domain size.

To avoid explicit enumeration of the effects for all joint actions, we exploit the factored structure inherent in the joint action representation and assume that effects for joint actions can be specified as a union of effects of their sub-actions. To see that such an assumption can be

reasonable, we note that for the effect axioms given for SYSADMIN in Equation 6.18, each of the effect conjuncts in the consequence of the axiom arises due to exactly one of the sub-actions in the antecedent. We now proceed to formally define compact representation of effect axioms in this manner where *we omit a discussion of preconditions for now*, but later show how we can reintroduce them.

In this section, we represent the effect of an action to be a consistent (potentially empty) conjunction of positive and negative fluents in a post-action state (i.e., elements of this conjunction may be of the form $F(\vec{x}, do(a, s))$ or $\neg F(\vec{x}, do(a, s))$). We use the notation $E_{n_i(\vec{x},\vec{y})}$ to specify the effects of Nature's choice sub-action $n_i(\vec{x}, \vec{y})$. For example, in SYSADMIN $E_{rebootS(x,c_i)} \equiv Up(c_i, do(a, s))$ and $E_{rebootF(x,c_i)} \equiv \neg Up(c_i, do(a, s))$. We assume that an *effect set* E_a for joint actions a can be specified as a union of effects for each Nature's choice sub-action $n_i(\vec{x}, \vec{y})$ from which a is composed:

$$E_a = \bigcup_{\dot{\vec{y}},\, i=1\ldots p} E_{n_i(\vec{x},\dot{\vec{y}})} \tag{6.19}$$

Based on this relationship, we can easily write out the unnormalized effect axioms for joint actions in the following manner where we use $\bigcirc$ to specify a composition $\circ$ over multiple terms:

$$\left[a = \bigcirc_{\dot{\vec{y}},\, i=1\ldots p} n_i(\vec{x}, \dot{\vec{y}})\right] \supset \bigwedge_{\dot{\vec{y}},\, i=1\ldots p} E_{n_i(\vec{x},\dot{\vec{y}})} \tag{6.20}$$

From this definition and the previous specification of effect sets $E_{rebootS(x,c_i)}$ and $E_{rebootF(x,c_i)}$ for SYSADMIN, we can easily derive all of the effect axioms for all joint actions in SYSADMIN including the example for $n = 2$ computers in Equation 6.18.

Now, rather than explicitly construct effect axioms for each joint action a, we can exploit the fact that the effects for a joint action are just the union of effects for each of its sub-actions. Technically, this should allow us to specify effect axioms directly in terms of sub-actions if we could simply test whether a joint action contains a sub-action. But this is easy, we simply define the $\sqsupseteq$ predicate that tests whether the joint-action on the LHS is composed from the RHS term. For example, given a as specified in Equation 6.15, we know that $a \sqsupseteq rebootS(c_1)$ is true, but $a \sqsupseteq rebootF(c_4)$ is false. And we can now express unnormalized *factored effect axioms* directly in terms of sub-actions in the following general manner:

$$\forall i, \dot{\vec{y}}.\ a \sqsupseteq n_i(\vec{x}, \dot{\vec{y}}) \supset E_{n_i(\vec{x},\dot{\vec{y}})} \tag{6.21}$$

For example, in SYSADMIN, we can now specify the following positive and negative effect axioms for the joint action a directly in terms of sub-actions:

$$\forall c_i.\ a \sqsupseteq rebootS(c_i) \supset Up(c_i, do(a, s)) \tag{6.22}$$

$$\forall c_i.\ a \sqsupseteq rebootF(c_i) \supset \neg Up(c_i, do(a, s)) \tag{6.23}$$

And it is easy to see that this compact (in fact, constant-sized) specification of effect axioms suffices to specify all of the effects for any arbitrary SYSADMIN domain size, including the example for $n = 2$ computers in Equation 6.18.

Now we tackle preconditions for effects. As done for effects, we assume that preconditions are associated with sub-actions. We could easily integrate preconditions into our factored effect axioms; however, we will instead find it advantageous to embed the preconditions directly into the probability distribution governing Nature's choice sub-action outcomes since this lays bare important probabilistic structure that we can exploit in our solution methods.We formalize this in the following assumption:

Assumption 6.1.2. *If $\varphi(\vec{x}, \vec{y}, s)$ is a precondition for the successful execution of sub-action $n_i(\vec{x}, \vec{y})$ of an aspect $A_i(\vec{x}, \vec{y})$ in situation s, then we assume that the case statement representing $P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$ contains the case partition $\langle \neg\varphi(\vec{x}, \vec{y}, s), 0\rangle$ (c.f. Equation 4.8). Furthermore, letting the $noop_i(\vec{x}, \vec{y})$ sub-action represent an outcome of $A_i(\vec{x}, \vec{y})$ with no effects, we assume $P(noop_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s)$ contains the case partition $\langle \neg\varphi(\vec{x}, \vec{y}, s), 1\rangle$.*

As a consequence, if two sub-actions had mutually exclusive preconditions then this assumption would ensure that any joint action composed from them would have probability 0. Our SYSADMIN examples does not require preconditions on sub-actions, however, in the FBOXWORLD problem that we formally define in Section 6.1.4 and present in Table 6.1, we do make use of preconditions defined in this way.

Our factored representation of effect axioms now permits us to convert them to the normal form representation of Equations 4.1 and 4.2 and convert them to SSAs as done in Section 4.3.3. This in turn directly facilitates the efficient computation of the regression operator. As an example, since the two axioms above are the only effect axioms for SYSADMIN and are already in normalized form, we can use these to directly compile the SSA for SYSADMIN's only fluent $Up(c, s)$:

$$Up(c_i, do(a, s)) \equiv a \sqsupseteq rebootS(x, c_i) \vee Up(c_i, s) \wedge \neg a \sqsupseteq rebootF(x, c_i)$$

And regression follows directly from the SSAs and the definitions given in Chapter 4, Section 4.2.3. In SYSADMIN we obtain the following example of regression using SSAs compiled from our factored effect axioms:

$$\forall a.\ a \sqsupseteq rebootS(c_i) \ \supset \ Regr[Up(c_i, do(a, s))] \equiv \top \tag{6.24}$$

$$\forall a.\ a \sqsupseteq rebootF(c_i) \ \supset \ Regr[Up(c_i, do(a, s))] \equiv \bot \tag{6.25}$$

Joint Actions and Consistency

Having defined the probabilistic and logical machinery for joint actions, the question arises as to what happens if a joint action a is composed of inconsistent effects, for example, $a \sqsupseteq rebootS(x, c_i)$ and $a \sqsupseteq rebootF(x, c_i)$. Clearly for SYSADMIN, this could never occur because only one deterministic action, $rebootS(x, c_i)$ or $rebootF(x, c_i)$, is chosen per c_i (we prove this formally below). But for the sake of argument, let us assume that a conflict is possible, as it will be for more general factored FOMDPs like the F-BOXWORLD problem that we formalize next. If we evaluated $Regr[\phi(do(a, s))]$ where a is a joint deterministic action with inconsistent effects, we would obtain $\bot$ by definition. This is fine, but then we simply need to ensure that if a joint deterministic outcome a of a stochastic action $A(\vec{x})$ is inconsistent then $P(a|A(\vec{x}), s) = 0$, otherwise we could assign a non-zero probability to being in an inconsistent state.

To satisfy this condition, we make the following formal assumption regarding the consistency of action effects in a factored FOMDP definition where in this context, we assume a case statement $case(s)$ satisfies $case(s) > 0$ if one its case partitions $\langle \phi_i(s), t_i \rangle$ has $t_i > 0$:

Assumption 6.1.3. *For all joint deterministic outcomes* $a = \bigcirc_{\dot{\vec{y}}, i=1\ldots p} n_i(\vec{x}, \dot{\vec{y}})$ *of all stochastic actions* $A(\vec{x})$*, if* $P(a|A(\vec{x}), s) > 0$ *then we assume*

$$\left(\bigwedge_{\dot{\vec{y}}, i=1\ldots p} E_{n_i(\vec{x}, \dot{\vec{y}})} \right) \ \textit{is consistent.}$$

Effectively, we see here that the onus is on the specifier of the factored FOMDP to correctly formalize the problem in order to ensure that all non-zero probability joint actions are consistent. Fortunately, this is often easy to verify without explicitly enumerating all possible joint actions as we did above for SYSADMIN. The following sufficient conditions show that Assumption 6.1.3 can be verified by a pairwise analysis of aspects of all stochastic actions in the following manner:

Proposition 6.1.4. *Let $A_i(\vec{x}, \dot{\vec{y}}_j)$ and $A_h(\vec{x}, \dot{\vec{y}}_k)$ be two distinct aspects of stochastic action $A(\vec{x})$ (i.e., either $i \neq h$ or $\dot{\vec{y}}_j \neq \dot{\vec{y}}_k$) and recall that $N_i(\vec{x}, \dot{\vec{y}}_j)$ and $N_h(\vec{x}, \dot{\vec{y}}_k)$ are the respective sets of Nature's deterministic sub-action outcomes for each of these aspects. Then for all $A(\vec{x})$ and $i \neq h$ and $\dot{\vec{y}}_j \neq \dot{\vec{y}}_k$ where $n_i(\vec{x}, \dot{\vec{y}}_j) \in N_i(\vec{x}, \dot{\vec{y}}_j)$ and $n_h(\vec{x}, \dot{\vec{y}}_k) \in N_h(\vec{x}, \dot{\vec{y}}_k)$, if the following condition holds:*

$$\forall \vec{x}. \, [\, P(n_i(\vec{x}, \dot{\vec{y}}_j) | A(\vec{x}), s) > 0 \;\; \wedge \;\; P(n_h(\vec{x}, \dot{\vec{y}}_k) | A(\vec{x}), s) > 0 \\ \supset \; (E_{n_i(\vec{x}, \dot{\vec{y}}_j)} \wedge E_{n_h(\vec{x}, \dot{\vec{y}}_k)}) \text{ is consistent }] \tag{6.26}$$

then Assumption 6.1.3 must hold for the given factored FOMDP.

Proof. See Section B.3 of Appendix B.

Now we demonstrate how this proposition can be used to verify the correctness of SYSADMIN. First we note that, as the SYSADMIN factored FOMDP has been defined, there is only one stochastic action $reboot(x)$ that we need to analyze. For $reboot(x)$, we must look at all aspects $reboot_1(x, c_j)$ and $reboot_1(x, c_k)$ where $c_j \neq c_k$. We note that $n_1(x, c_j) \in \{rebootS(x, c_j), rebootF(x, c_j)\}$ and likewise $n_1(x, c_k) \in \{rebootS(x, c_k), rebootF(x, c_k)\}$. For all assignments of $n_1(x, c_j)$ and $n_1(x, c_k)$, we note

$$\forall x. \, [\, pCase(n_1(x, c_j) | reboot(x), s) > 0 \;\; \wedge \;\; pCase(n_1(x, c_k) | reboot(x), s) > 0 \,]$$

according to Equations 6.10, 6.11, and 6.12 so we must likewise show that for all x and effects $E_{n_1(x,c_j)}$ and $E_{n_1(x,c_k)}$ that their conjunction is consistent. Consequently, we enumerate all possible conjunctions of these effects as defined previously and check consistency (which holds trivially by inspection since $c_j \neq c_k$ and thus each ground fluent can take on any truth assignment while maintaining consistency):

$$\forall x. \, [\, E_{rebootS(x,c_j)} \wedge E_{rebootS(x,c_k)} \longrightarrow (Up(x, c_j) \wedge Up(x, c_k)) \text{ is consistent }]$$
$$\forall x. \, [\, E_{rebootS(x,c_j)} \wedge E_{rebootF(x,c_k)} \longrightarrow (Up(x, c_j) \wedge \neg Up(x, c_k)) \text{ is consistent }]$$
$$\forall x. \, [\, E_{rebootF(x,c_j)} \wedge E_{rebootS(x,c_k)} \longrightarrow (\neg Up(x, c_j) \wedge Up(x, c_k)) \text{ is consistent }]$$
$$\forall x. \, [\, E_{rebootF(x,c_j)} \wedge E_{rebootF(x,c_k)} \longrightarrow (\neg Up(x, c_j) \wedge \neg Up(x, c_k)) \text{ is consistent }]$$

This formally proves the consistency of the SYSADMIN factored FOMDP.

6.1.4 Formalizing Another Factored FOMDP

Having introduced the factored FOMDP formalization as motivated by SYSADMIN, we now demonstrate the flexibility of this formalism for modeling the F-BOXWORLD problem. We begin with the specification of BOXWORLD as made in Figure 4.1 of Chapter 4, except that we now modify two parts of the representation: (1) we reparameterize actions and transition probabilities for $unload, load, drive$ to satisfy the representation of action preconditions made in the factored FOMDP representation; (2) on each time step, any box that is on a truck can independently fall off the truck if it was not explicitly loaded or unloaded, which is modeled with additional exogenous aspects that indefinitely scale with domain size.

Our first task is to specify all stochastic actions, their aspects, sub-action outcomes for these aspects and their respective probabilities. These are given in Table 6.1. For the first aspects — $unload_1(b, t, c)$, $load_1(b, t, c)$, and $drive_1(t, c)$ — the deterministic action outcomes, probabilities, and corresponding effect axioms are semantically identical to thr original BOXWORLD although some preconditions in the effect axioms are now represented in the transition probabilities, thus requiring the additional shared variables in the action parameterization. We note that the second aspect of each stochastic action ranges over all boxes, trucks, and cities and permits each box to be dropped off a truck according to some independent fixed probability distribution; it is important to note that these second aspect probabilities for $dropS(\cdots, b', t', c')$ ensure that this sub-action outcome can only occur with non-zero probability when (a) it does not interfere with the first aspect and (b) when box b' is actually loaded on a truck t' in city c'.

Thus, we see that F-BOXWORLD is exactly the variant as previously described — the action dynamics are exactly that of BOXWORLD with the additional aspects that each box may independently fall off a truck with 0.1 probability when they are not being explicitly loaded or unloaded. Furthermore, this specification guarantees that all pairs of non-zero probability sub-actions that can co-occur in a joint action have consistent effects.[6] Having now specified factored FOMDPs for both SYSADMIN and F-BOXWORLD, we proceed to generalize solution techniques from Chapters 4 and 5 to find (approximately) optimal solutions for them.

6.2 Factored Symbolic Dynamic Programming

We now proceed with an extension of symbolic dynamic programming for the factored FOMDP representation. Recalling Chapter 4, we note that symbolic dynamic programming (SDP) con-

[6] This verification is straightforward, but tedious.

Action	Aspects	Sub-actions	Probability	Effect Axioms
$unload(b,t,c)$	$unload_1(b,t,c)$	$unloadS(b,t,c)$	$On(b,t,s) \wedge TIn(t,c,s)$: .9 $\neg(On(b,t,s) \wedge TIn(t,c,s))$: 0	$a \sqsupseteq unloadS(b,t,c) \supset \neg On(b,t,do(a,s))$ $a \sqsupseteq unloadS(b,t,c) \supset BIn(b,c,do(a,s))$
		$unloadF(b,t,c)$	$On(b,t,s) \wedge TIn(t,c,s)$: .1 $\neg(On(b,t,s) \wedge TIn(t,c,s))$: 1	(note: $unloadF(b,t,c)$ equivalent to $noop$)
	$unload_2(b,b',t',c')$	$dropS(b,b',t',c')$	$b \neq b' \wedge On(b',t',s) \wedge TIn(t',c',s)$: .1 $\neg(b \neq b' \wedge On(b',t',s) \wedge TIn(t',c',s))$: 0	$a \sqsupseteq dropS(b,b',t',c') \supset \neg On(b',t',do(a,s))$ $a \sqsupseteq dropS(b,b',t',c') \supset BIn(b',c',do(a,s))$
	$\forall b',t',c'$	$dropF(b,b',t',c')$	$b \neq b' \wedge On(b',t',s) \wedge TIn(t',c',s)$: .9 $\neg(b \neq b' \wedge On(b',t',s) \wedge TIn(t',c',s))$: 1	(note: $dropF(b,t,c)$ equivalent to $noop$)
$load(b,t,c)$	$load_1(b,t,c)$	$loadS(b,t,c)$	$On(b,t,s) \wedge TIn(t,c,s)$: .9 $\neg(On(b,t,s) \wedge TIn(t,c,s))$: 0	$a \sqsupseteq loadS(b,t,c) \supset On(b,t,do(a,s))$ $a \sqsupseteq loadS(b,t,c) \supset \neg BIn(b,c,do(a,s))$
		$loadF(b,t,c)$	$On(b',t',s) \wedge TIn(t',c',s)$: .1 $\neg(On(b',t',s) \wedge TIn(t',c',s))$: 1	(note: $loadF(b,t,c)$ equivalent to $noop$)
	$load_2(b,b',t',c')$	$dropS(b,b',t',c')$	(note: same probabilities and effects as $dropS(b,b',t',c')$ for $unload_2$)	
	$\forall b',t',c'$	$dropF(b,b',t',c')$	(note: same probabilities and effects as $dropF(b,b',t',c')$ for $unload_2$)	
$drive(t,c_1,c)$	$drive_1(t,c_1,c)$	$driveS(t,c_1,c)$	$TIn(t,c_1,s)$: 1 $\neg TIn(t,c_1,s)$: 0	$a \sqsupseteq driveS(t,c_1,c) \supset TIn(t,c,s)$ $a \sqsupseteq driveS(t,c_1,c) \supset \neg TIn(t,c_1,s)$
		$driveF(t,c_1,c)$	$TIn(t,c_1,s)$: 0 $\neg TIn(t,c_1,s)$: 1	(note: $driveF(t,c)$ equivalent to $noop$)
	$drive_2(b',c',t')$	$dropS(b',c',t')$	$BIn(b',c',s) \wedge TIn(t',c',s)$: .1 $\neg(BIn(b',c',s) \wedge TIn(t',c',s))$: 0	$a \sqsupseteq dropS(b',c',t') \supset \neg On(b',t',do(a,s))$ $a \sqsupseteq dropS(b',c',t') \supset BIn(b',c',do(a,s))$
	$\forall b',t',c'$	$dropF(b',c',t')$	$BIn(b',c',s) \wedge TIn(t',c',s)$: .9 $\neg(BIn(b',c',s) \wedge TIn(t',c',s))$: 1	(note: $dropF(b',c',t')$ equivalent to $noop$)

Table 6.1: Factored FOMDP formulation of F-BOXWORLD. Predicates $TruckIn$, $BoxIn$, $BoxOn$ have been shortened to fit the table on one page. Variables start with the same letter of their type (i.e. Box, $Truck$, $City$) and unused action parameters are omitted from the second aspects.

sisted of two steps: (1) first-order decision-theoretic regression and (2) symbolic maximization. As we will see, it is easy to symbolically define these steps for factored FOMDPs, but it often requires some ingenuity to derive a compact and simplified result for SDP.

6.2.1 Exploiting Irrelevance

As we discussed in Chapter 3, an important aspect of efficiency in the dynamic programming solution of propositionally factored MDPs is exploiting probabilistic independence in the DBN representation of the transition distribution. The same will be true for factored FOMDPs except that now we must provide a novel first-order generalization of probabilistic independence:

Definition 6.2.1. *An aspect $A_i(\vec{x}, \vec{y})$ having a set of deterministic sub-action outcomes $N_i(\vec{x}, \vec{y})$ is* irrelevant *to a formula $\phi(s)$ (abbreviated $Irr[\phi(s), A_i(\vec{x}, \vec{y})]$)* iff

$$\forall\, n_i(\vec{x}, \vec{y}) \in N_i(\vec{x}, \vec{y}).\ \ [Regr(\phi(do(n_i(\vec{x}, \vec{y}), s)))\ \equiv\ \phi(s)]\ . \tag{6.27}$$

This definition simply states that an aspect is irrelevant to a formula $\phi(s)$ if all deterministic sub-action outcomes of that aspect cannot affect the regression of a formula. In general, we can prove the case equivalence needed to show irrelevance by converting the case representation to its logical equivalent and querying an off-the-shelf theorem prover. Most often though, a simple analysis of the effect axioms and the removal of superficial inconsistencies such as equality tests of distinct terms will allow us to show equivalence by syntactic comparison without the need for theorem proving.

For example, in F-BOXWORLD, we can say that the aspect $drive_1(t, c)$ that governs whether a truck t is successfully driven to city c (c.f., Table 6.1) is relevant to $TruckIn(t, c, s)$, but irrelevant to $BoxIn(b, c, s)$. This follows from an analysis of the effect axioms for the $driveS(t, c)$ and $driveF(t, c)$ sub-action outcomes of $drive_1(t, c)$ where we see that $driveS(t, c)$ clearly affects $TruckIn(t, c, s)$, but neither sub-action affects $BoxIn(b, c, s)$. Furthermore, the more constant substitutions we have in a formula or action, the more irrelevance we can detect. Assuming we have unique constants $\dot{t}_1, \dot{t}_2, \dot{c}_1, \dot{c}_2$ and a ground aspect $drive_1(\dot{t}_1, \dot{c}_1)$, then $drive_1(\dot{t}_1, \dot{c}_1)$ is relevant to $\exists t, c.\ TruckIn(t, c, s)$, but irrelevant to $TruckIn(\dot{t}_2, \dot{c}_2)$. A simple analysis of the effect axioms and the removal of inconsistent equalities and conjunctions easily allows us to show this.

Once we have detected irrelevant action aspects w.r.t. a formula $\phi(s)$, we can *drop any irrelevant sub-actions for these aspects from a joint action a* when performing $Regr(\phi(do(a, s)))$.

More formally we can state this with the following proposition where we note that the domain of c is restricted by the condition $a = n_i(\vec{x}, \vec{y}) \circ c$:

Proposition 6.2.2 (Removal of Irrelevant Aspects)**.**

$$Irr[\phi(s), A_i(\vec{x}, \vec{y})] \supset$$
$$\Big\{\forall\, n_i(\vec{x}, \vec{y}) \in N_i(\vec{x}, \vec{y}),\ \ \forall a \in N_A(\vec{x}),\ \ \forall c.\ (a = n_i(\vec{x}, \vec{y}) \circ c) \supset$$
$$Regr[\phi(s), n_i(\vec{x}, \vec{y}) \circ c] \equiv Regr[\phi(s), c]\Big\}$$

Proof. See Section B.3 of Appendix B.

6.2.2 Backup Operators

Now that we have specified a compact representation of Nature's probability distribution over deterministic actions, we will exploit this structure in the backup operators that perform the first-order decision-theoretic regression step of symbolic dynamic programming.

Following our definitions of the $B^{A(\vec{x})}[\cdot]$ and $B^{A}[\cdot]$ operators from Chapter 5, Section 5.1.2, we extend these definitions to incorporate factored transition models that are now possible in factored FOMDPs. We start with the $B^{A(\vec{x})}[\cdot]$ operator with updated notation for factored FOMDPs:

$$B^{A(\vec{x})}[V(s)] = \gamma \bigoplus_{a \in N_A(\vec{x})} \Big[P(a|A(\vec{x})) \otimes Regr(V(do(a, s)))\Big] \tag{6.28}$$

Of course, $P(a|A(\vec{x}))$ is implicitly factored according to Equation 6.16 and the factored action representation, so we can expand this out using previous notation where (1) we substitute $a = \bigcirc_{\dot{\vec{y}},\, i=1\ldots p} n_i(\vec{x}, \dot{\vec{y}})$, (2) we express the marginalization over all $a \in N_A(\vec{x})$ as an equivalent sum over all factored sub-actions $n_i(\vec{x}, \dot{\vec{y}})$, and (3) we substitute $P(a|A(\vec{x}))$ with its factored

representation from Equation 6.16:

$$B^{A(\vec{x})}[V(s)] = \gamma \bigoplus_{n_1(\vec{x},\dot{\vec{y}}_1)\in N_1(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}),\ldots,n_1(\vec{x},\dot{\vec{y}}_1)\in N_1(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})}$$
$$\left[\prod_{\vec{y}\in\{\dot{\vec{y}}_1,\ldots,\dot{\vec{y}}_{|\vec{y}|}\}} \prod_{i=1\ldots p} P(n_i(\vec{x},\vec{y})|A_i(\vec{x},\vec{y}),s) \right.$$
$$\left. \otimes\, Regr(V(do(n_1(\vec{x},\vec{y}_1)\circ\cdots\circ n_p(\vec{x},\vec{y}_{|\vec{y}|})\circ\cdots\circ n_1(\vec{x},\vec{y}_1)\circ\cdots\circ n_p(\vec{x},\vec{y}_{|\vec{y}|}),s))) \right] \tag{6.29}$$

Fortunately, we will often find that many sub-actions can be deemed irrelevant to a value function or aggregated in some compact manner that will prevent the need for regression through the fully specified joint action. We will see this borne out in two very different ways in the SYSADMIN and F-BOXWORLD examples.

First, however, let us recall the $B^A[\cdot]$ operator from Equation 5.8 in Chapter 5:[7]

$$B^A[V(s)] = \exists\vec{x}.\ \left\{B^{A(\vec{x})}[V(s)]\right\} \tag{6.30}$$

If the result of $B^{A(\vec{x})}[\cdot]$ can be represented as an expression using $\oplus$, $\otimes$, $\ominus$ over standard case statements as defined in Equation 4.8 *without* product or sum aggregators then we can apply $\exists\vec{x}$ directly, taking care to exploit (linear) structure in our value function if it exists as discussed in Section 5.1.2 of Chapter 5. However, if the result of $B^{A(\vec{x})}[\cdot]$ contains indefinite sum or product aggregator structure, the application of $\exists\vec{x}$ is less straightforward and we must leave this operator in purely symbolic form; in some cases, we will be able to directly evaluate $\exists\vec{x}$ in conjunction with symbolic maximization that we describe next.

6.2.3 Symbolic Maximization

We note that the result of the $B^A[V(s)]$ operator is close to our definition of a Q-function from Equation 4.25 except that we have omitted the reward. So let us add in the reward now to obtain a proper Q-function representation for a factored FOMDP. Assuming we are given a $(t\text{-}1)$-stage-go-to value function, we can compute a t-stage-to-go Q-function for action A as

[7] As done previously, we assume that the reward is not dependent on the action and thus omit it here. However, if this was not the case, we could easily insert it in this equation and make appropriate adjustments to our later equations.

follows:

$$Q^t(s, A) = R(s) \oplus B^A[V^{t-1}(s)] \tag{6.31}$$

As before, this $Q^t(s, A)$ represents a logical description of the Q-function for action A indicating the values that could be achieved by *any* instantiation of $A(\vec{x})$ if acting so as to obtain $V^{t-1}(s)$ after the action is performed.

And now, given the Q-functions, we want to maximize over them to compute $V^t(s)$ just as we did in the original definition of symbolic dynamic programming from Chapter 4. Notwithstanding computational difficulties owing to the structure of the $Q^t(s, A)$, we recall Equation 4.26 and represent this computation symbolically assuming we have actions $A^1, \ldots, A^m$:[8]

$$V^t(s) = \text{ casemax } \left[Q^t(s, A^1) \cup \ldots \cup Q^t(s, A^m)\right] \tag{6.32}$$

In the case where we are fortunate enough to represent $Q^t(s, A)$ as a standard case statement from Equation 4.8 where the case partition values are numerical constants, we can compute $\cup$ and casemax as defined previously and this completes one step of symbolic dynamic programming. However, we note that $R(s)$ may contain indefinite sum aggregator structure as in Equation 6.4 and $V^{t-1}(s)$ may also contain indefinite sum or product aggregator structure. Furthermore as mentioned previously and as we will show for F-BOXWORLD, $V^{t-1}(s)$ or its backup may be specified as case statements with case partition values that are a function of domain properties rather than simple numerical constants (see further discussion below). Both the indefinite and parameterized case structure that may arise in these case statements will almost always propagate to the representation of $Q^t(s, A)$, thus preventing the direct application of the $\cup$ and casemax case operators as previously defined in Chapter 4. We provide remedies to exploit some of this additional structure in the following sections.

Parameterized Case Structure

In some cases, the result of $B^{A(\vec{x})}[\cdot]$ may be parameterized by properties of the domain such as the sizes of individual object classes, or more generally, counts of the number of satisfying instances of some domain property. Thus, the result of $B^{A(\vec{x})}[\cdot]$ may have the following generic

[8] We superscript the indices for different action templates in this chapter to avoid confusion with the action subscripts that can be used to denote aspect indices for a particular action.

parameterized case format:

$$(t \;=\; case[\phi_1, t_1; \cdots; \phi_n, t_n]) \;\equiv\; \left(t \;=\; \begin{array}{|c|} \hline \phi_1 : f_1(\cdot) \\ \hline : \;\; : \;\; : \\ \hline \phi_n : f_n(\cdot) \\ \hline \end{array} \right) \tag{6.33}$$

Here $f_i(\cdot)$ may be any arithmetic function and its argument may range over a variety of domain and state properties. For example, we will see precisely such a structure when we provide an example backup computation for F-BOXWORLD in Section 6.2.4 where some of the case partition values of $B^{unload(b^*,t^*)}[\cdot]$ will be a function of the count of boxes on trucks in Paris in situation s. We note that such a parameterized case format does not pose any complications for the computation of $B^A[\cdot]$ as previously noted (the $\exists \vec{x}$ can still be pushed directly into the partitions of the case representation), but it will pose a few moderate complications for symbolic maximization that we discuss now.

If $Q^t(s, A) = B^{A(\vec{x})}[V^{t-1}(s)]$ takes on the parameterized structure of Equation 6.33 then we can still perform a $\cup$ as usual in Equation 6.32, but we need to modify casemax to explicitly check that a value is greater than all other values. To avoid cumbersome notation, we simply provide a small example:

$$\text{casemax} \begin{array}{|c|} \hline \psi_1 : f_1(\cdot) \\ \hline \psi_2 : f_2(\cdot) \\ \hline \end{array} \;=\; \begin{array}{|ll|} \hline \psi_1 \wedge \psi_2 \wedge [f_1(\cdot) \geq f_2(\cdot)] & : \; f_1(\cdot) \\ \hline \psi_1 \wedge \psi_2 \wedge [f_2(\cdot) \geq f_1(\cdot)] & : \; f_2(\cdot) \\ \hline \psi_1 \wedge \neg\psi_2 & : \; f_1(\cdot) \\ \hline \neg\psi_1 \wedge \psi_2 & : \; f_2(\cdot) \\ \hline \end{array} \tag{6.34}$$

In general, to perform the casemax of n parameterized case partitions, we need to look at the powerset of the configurations $\{\phi_i, \neg\phi_i\}$ of each case partition $\langle \phi_i, t_i \rangle$ in the operand. Then within each element of that powerset, we need to assign the functional value that is maximal for that region of state space. While such a parameterized casemax does produce a case statement with $n \cdot 2^n$ case partitions in the worst case, we note that often the functional form of $f_i(\cdot)$ is such that many inequalities can actually be determined without knowing the exact values of the function arguments. For example, if $f_i(\cdot)$ takes a value that is a linear function of its arguments then it is easy to determine that $f_1(x, y) = 2x + 3y$ strictly dominates $f_2(x, y) = x + y$ for $x > 0$ and $y > 0$. If these conditions held, we could then remove the second case partition of the casemax result in Equation 6.34 since it would be inconsistent.

As such, we now can perform the full symbolic dynamic programming update on parame-

terized case statements using the above modification to casemax. Unfortunately, the task will not be as simple for indefinite case structure as we will see next.

Indefinite Case Structure

If $Q^t(s, A)$ contains indefinite sum aggregator structure[9], then we can neither perform the $\cup$ nor the casemax directly as specified in Equation 6.32. Both these problems arise from the fact that the indefinite sum aggregator structure leads to an indefinite number of cases to analyze for each of these operations. When does such structure occur in our $Q^t(s, A)$ representation? If the reward was additively defined with a sum aggregator, then by the linearity of the $B^{A(\vec{x})}[\cdot]$ operator, its result will retain this sum aggregator structure.

In this case, we indirectly maximize over $Q^t(s, A^1) \cup \ldots \cup Q^t(s, A^m)$ by deriving an explicit policy and using this to restrict the value function, much the same as we did in FOAPI. To do this, we must first define the $\geq$ case operator[10], which produces an indicator function for states where the inequality holds:[11]

$$\begin{aligned} case[\phi_i, t_i : i \leq n] \geq case[\phi_j, v_j : j \leq m] \\ = case[\phi_i \wedge \psi_j, \mathbb{I}[t_i \geq v_j] : i \leq n, j \leq m] \end{aligned} \tag{6.35}$$

Intuitively, to perform a $\geq$ operation on case statements, we simply perform the corresponding operation on the intersection of all case partitions of the operands:

$$\begin{array}{|c|} \hline \phi_1 : 10 \\ \hline \phi_2 : \ \ 5 \\ \hline \end{array} \quad \geq \quad \begin{array}{|c|} \hline \psi_1 : \ \ 5 \\ \hline \psi_2 : 10 \\ \hline \end{array} \quad = \quad \begin{array}{|c|} \hline \phi_1 \wedge \psi_1 : 1 \\ \hline \phi_1 \wedge \psi_2 : 1 \\ \hline \phi_2 \wedge \psi_1 : 1 \\ \hline \phi_2 \wedge \psi_2 : 0 \\ \hline \end{array}$$

Second, following the policy construction of Guestrin *et al* [2002], we can assume that our default policy is to apply $noop$, and that we only want to execute action $A(\vec{x})$ in a state for instantiations of $\vec{x}$ that offer maximum advantage over the $noop$. Thus we can write an advantage function $ADV^t_{A \succ noop}(s)$ as the following difference computation:

$$ADV^t_{A \succ noop}(s) = \text{casemax}\, \exists \vec{x} \ \left(R(s) \oplus B^{A(\vec{x})}[V^{t-1}(s)] \ominus Q^t(s, noop) \right) \tag{6.36}$$

[9] We do not discuss symbolic maximization when $Q^t(s, A)$ contains indefinite product aggregator structure.

[10] Based on the $\geq$ definition, we can easily define similar procedures holding for $>$,$<$, and $\leq$ case operators.

[11] As in previous chapters, $\mathbb{I}$ is an indicator function taking the value 1 when its argument is true and 0 otherwise.

Here we see that $ADV_{A \succ noop}(s)$ represents the maximal Q-value advantage that can be had by taking some instantiation of action A over a $noop$. This may seem like a small step, but this step is crucial for obtaining a finite representation of a policy even in circumstances where the Q-functions retain indefinite sum aggregator structure. The key insight is that an action will typically only have (finite) local effects and thus the rest of the state will evolve according to it's default distribution, presumably the same as the $noop$ distribution. Thus, by looking at the advantage function, most of the indefinite structure should be identical and thus cancel out in the $\ominus$. While we are not claiming this will occur for all factored FOMDPs with sum aggregators in their Q-functions, such structure is not uncommon and we will see an example in Section 6.2.4 of how it can be exploited for the SYSADMIN domain.

Now, from advantage functions, we need to derive a policy indicator function indicating in which states each action should be applied. We can compute this using a first-order generalization of the policy generation technique used for factored MDPs in Section 3.2.2 of Chapter 3. We assume we are given actions $A^1, \ldots, A^m$, and for the purpose of breaking ties, a total preference ordering (perhaps random) over actions, that is, for all actions A^i and A^j such that $A^i \neq A^j$, we have either $A^i \succ A^j$ or $A^j \succ A^i$. From this, we can then compute $\pi^t_{A^i}$ as follows:

1. Initialize $\pi^t_{A^i} = \boxed{\top : 1}$
2. For each A^j s.t. $A^i \neq A^j$ update $\pi^t_{A^i}$:

$$\pi^t_{A^i} := \begin{cases} A^i \succ A^j : & \pi^t_{A^i} \otimes \left(ADV^t_{A^i \succ noop}(s) \geq ADV^t_{A^j \succ noop}(s) \right) \\ A^j \succ A^i : & \pi^t_{A^i} \otimes \left(ADV^t_{A^i \succ noop}(s) > ADV^t_{A^j \succ noop}(s) \right) \end{cases}$$

To calculate the $noop$ policy, we simply perform the following calculation:

$$\pi^t_{noop} = \boxed{\top : 1} \ominus \pi^t_{A^1} \ominus \cdots \ominus \pi^t_{A^m} \tag{6.37}$$

Finally, following the methodology used to extract a free variable policy as done in Equation 5.20 of Chapter 5, we can do likewise to obtain $\pi^t_{A(\vec{x})}$ from π^t_A. Note that we previously stated that we could not explicitly compute $B^A[V(s)]$ for indefinitely structured $V(s)$. However, we can now compute $B^{\pi^t}[V(s)]$ if we have a collection of policies $\pi^t_{A^1(\vec{x})}, \ldots, \pi^t_{A^m(\vec{x})}$ as follows:

$$B^{\pi^t}[V(s)] = \bigoplus_{i=1}^{m} \exists \vec{x}.\ \left[\pi^t_{A^i(\vec{x})} \otimes B^{A(\vec{x})}[V(s)] \right] \tag{6.38}$$

And in turn, we can finally note that this gives us an indirect method for computing $V^t(s)$ via the policy for $V^{t-1}(s)$:

$$V^t(s) = rCase(s) \oplus B^{\pi^t}[V^{t-1}(s)] \tag{6.39}$$

Also as a side benefit of specifying $B^{\pi^t}[V(s)]$, we also have a method for computing the value of a policy under successive approximation. This provides us with a method for performing modified policy iteration, or as we will see in a future section, a factored extension of first-order approximate policy iteration.

6.2.4 Examples

Up to this point, our discussion has been quite abstract so we pause for a moment to provide a few insightful examples of the previously defined operations applied to the F-BOXWORLD and SYSADMIN problems.

Backups and Parameterized Structure

To demonstrate a situation where this parameterized case stucture occurs, we demonstrate an application of the $B^A[\cdot]$ operator for the F-BOXWORLD problem. Specifically, we compute $B^{unload(b^*,t^*,c^*)}[vCase^0(s)]$ where $vCase^0(s) = rCase(s)$ as defined in Equation 4.9 from Chapter 4.

To compute $B^{unload(b^*,t^*,c^*)}[vCase^0(s)]$, we begin by writing out the full backup using the template from Equation 6.40 and the F-BOXWORLD definition from Table 6.1 where we assume $Box = \{b_1, \ldots, b_{|B|}\}$, $Truck = \{t_1, \ldots, t_{|T|}\}$, and $City = \{c_1, \ldots, c_{|C|}\}$ cities in our problem domain leading to a joint action composed of $|B| \cdot |C| \cdot |T| + 1$ sub-actions that we need to marginalize over:

$$B^{unload(b^*,t^*,c^*)}[vCase^0(s)] = \gamma \cdot \bigoplus_{n_1(b^*,t^*,c^*) \in \{unloadS(b^*,t^*,c^*), unloadF(b^*,t^*,c^*)\}} \tag{6.40}$$

$$\bigoplus_{n_2(b^*,b_1,t_1,c_1) \in \{dropS(b^*,b_1,t_1,c_1), dropF(b^*,b_1,t_1,c_1)\}} \cdots \bigoplus_{n_2(b^*,b_{|B|},t_{|T|},c_{|C|}) \in \{dropS(b^*,b_{|B|},t_{|T|},c_{|C|}), dropF(b^*,b_{|B|},t_{|T|},c_{|C|})\}}$$

$$\Bigg\{ pCase(n_1(b^*,t^*,c^*)|unload(b^*,t^*,c^*)) \otimes \prod_{b_i \in Box} \prod_{c_j \in City} \prod_{t_k \in Truck} [pCase(n_2(b^*,b_i,t_j,c_k)|unload(b^*,t^*,c^*))]$$

$$\otimes Regr\left[\begin{array}{|l|} \hline \exists b. BoxIn(b, paris, do(n_1(b^*,t^*,c^*) \circ n_2(b^*,b_1,t_1,c_1) \circ \cdots \circ n_2(b^*,b_{|B|},t_{|T|},c_{|C|}), s)) : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right] \Bigg\}$$

For completeness, we note the $pCase$ definitions that follow from Table 6.1 with appropriate

variable renamings:

$$pCase(unloadS(b^*, t^*, c^*)|unload(b^*, t^*, c^*)$$
$$= \begin{array}{|l|} \hline BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, c^*, s) \quad : \ .9 \\ \hline \neg(BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, c^*, s)) : \ 0 \\ \hline \end{array}$$

$$pCase(unloadF(b^*, t^*, c^*)|unload(b^*, t^*c^**)$$
$$= \begin{array}{|l|} \hline BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, c^*, s) \quad : \ .1 \\ \hline \neg(BoxOn(b^*, t^*, s) \wedge TruckIn(t^*, c^*, s)) : \ 1 \\ \hline \end{array}$$

$$pCase(dropS(b^*, b_i, t_j, c_k)|unload(b^*, t^*, c^*)$$
$$= \begin{array}{|l|} \hline b^* \neq b_i \wedge BoxOn(b_i, t_j, s) \wedge TruckIn(t_j, c_k, s) \quad : \ .1 \\ \hline \neq (b^* \neq b_i \wedge BoxOn(b_i, t_j, s) \wedge TruckIn(t_j, c_k, s)) : \ 0 \\ \hline \end{array}$$

$$pCase(dropF(b^*, b_i, t_j, c_k)|unload(b^*, t^*, c^*)$$
$$= \begin{array}{|l|} \hline b^* \neq b_i \wedge BoxOn(b, t_j, s) \wedge TruckIn(t_j, c_k, s) \quad : \ .9 \\ \hline \neg(b^* \neq b_i \wedge BoxOn(b, t_j, s) \wedge TruckIn(t_j, c_k, s)) : \ 1 \\ \hline \end{array}$$

Now, our first task to simplify this computation is to identify irrelevant structure. Unfortunately, given the existentially quantified $BoxIn(b, paris, \cdot)$ fluent, we note that many actions are relevant — we could unload a box that makes $BoxIn(b, paris, \cdot)$ true, or any box $b_1, \ldots, b_{|B|}$ could drop off a truck to make this fluent true. However, an analysis of irrelevance does show that all $n_2(b^*, b_i, t_j, c_k)$ for $c_k \neq paris$ are irrelevant to $\exists b.BoxIn(b, paris, do(a, s))$, so we can remove these aspects from the joint action and aspect marginalization.

Nonetheless, we still have an indefinite number of actions that are not irrelevant. However, we can make one powerful observation. For all possible combinations of sub-actions $n_2(b_1, c_1, t_1) \circ \cdots \circ n_2(b_{|B|}, c_{|C|}, t_{|T|})$ from which a can be composed, only one aspect of the form $n_2(b^*, b_i, t_j, paris) = dropS(b^*, b_i, t_j, paris)$ for any b_i and t_j (i.e., only one arbitrary box falling off some truck) is required to make $\exists b.BoxIn(b, paris, do(a, s))$ true since it is easy to verify the following:

$$\forall b_i, t_j.\ a \sqsupseteq dropS(b^*, b_i, t_j, paris) \ \supset \ Regr[\exists b.BoxIn(b, paris, do(a, s))] \equiv \top\,.$$

Likewise if all $n_2(b^*, b_i, t_j, paris) = dropF(b^*, b_i, t_j, paris)$, then the regression is equivalent to a $noop$.

Based on this analysis, we can aggregate all relevant aspects $n_2(b^*, b_i, t_j, paris)$ into two

sets: set (a) where all $n_2(b^*, b_i, t_j, paris) = dropF(b^*, b_i, t_j, paris)$ and set (b) where at least one $n_2(b^*, b_i, t_j, paris) = dropS(b^*, b_i, t_j, paris)$. Now, it is easy to compute the probability of set (a), it is just the product of probabilities of the $dropF$ sub-action for all of these aspects which is $p = 0.9^{|\{\langle b_i,t_j\rangle | b_i \in Box \wedge b_i \neq b^*.\ BoxOn(b_i,t_j,s) \wedge TruckIn(t_j,paris,s)\}|}$, or in words, 0.9 to the power of the number of boxes on trucks in Paris excluding the box being unloaded. And the probability of set (b) is obviously $1 - p$. Since all joint actions in sets (a) and (b) regress uniformly, we can obtain the following (somewhat ad-hoc) simplified representation of the backup:

$$B^{unload(b^*,t^*,c^*)}[vCase^0(s)] = \gamma \cdot \tag{6.41}$$

$$\bigoplus_{n_1(b^*,t^*,c^*) \in \{unloadS(b^*,t^*,c^*), unloadF(b^*,t^*,c^*)\}} \quad \bigoplus_{n_2 \in \{\text{set(a)}, \text{set(b)}\}}$$

$$\Big[pCase(n_1(b^*, t^*, c^*) | unload(b^*, t^*, c^*)) \otimes pCase(n_2 | unload(b^*, t^*, c^*))$$

$$\otimes\ Regr \left[\begin{array}{|l|} \hline \exists b. BoxIn(b, paris, do(n_1(b^*, t^*, c^*) \circ n_2, s)) : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right] \Big]$$

Given that we know all of the probabilities for the $pCase$ and the results of regression through the four possible joint actions, we can explicitly perform the regressions (simplifying equalities where possible in the process), and write out the sum over these four cases:

$$B^{unload(b^*,t^*,c^*)}[vCase^0(s)] = \gamma \cdot \tag{6.42}$$

$$\oplus\ .9 \cdot p \cdot \left[\begin{array}{|l|} \hline \exists b. \left[(t = t^* \wedge BoxOn(b, t^*, s) \wedge TruckIn(t^*, paris, s)) \vee BoxIn(b, paris, s) \right] : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right]$$

$$\oplus\ .1 \cdot (1 - p) \cdot \left[\begin{array}{|l|} \hline \exists b. \left[(\exists t.\ BoxOn(b, t, s) \wedge TruckIn(t, paris, s)) \vee BoxIn(b, paris, s) \right] : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right]$$

$$\oplus\ .9 \cdot (1 - p) \cdot \left[\begin{array}{|l|} \hline \exists b. \left[(t = t^* \wedge BoxOn(b, t^*, s) \wedge TruckIn(t^*, paris, s)) \right. \\ \left. \vee (\exists t.\ BoxOn(b, t, s) \wedge TruckIn(t, paris, s)) \vee BoxIn(b, paris, s) \right] : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right]$$

$$\oplus\ .1 \cdot p \cdot \left[\begin{array}{|l|} \hline \exists b. BoxIn(b, paris, s) : 10 \\ \hline \neg \text{“} \qquad\qquad : \ 0 \\ \hline \end{array} \right]$$

Of course to determine the joint action that led to each case, one need only examine the probabilities that led to them. Here .9 is for $unloadS(b^*, t^*, c^*)$, .1 is for $unloadF(b^*, t^*, c^*)$, p is for set (a) of actions, and $(1 - p)$ is for set (b).

Finally, we perform the explicit sum to obtain the following simplified representation of the backup:

$$B^{unload(b^*,t^*,c^*)}[vCase^0(s)] = \gamma \cdot \tag{6.43}$$

$\exists b. BoxIn(b, paris, s)$	:	10
$\neg$“ $\wedge \{\exists b. [(t = t^* \wedge BoxOn(b, t^*, s) \wedge TruckIn(t^*, paris, s))$ $\vee (\exists t.\, BoxOn(b, t, s) \wedge TruckIn(t, paris, s)) \vee TruckIn(t, paris, s)]\}$	:	$10 - p$
$\neg$“ $\wedge \exists b. [t = t^* \wedge BoxOn(b, t^*, s) \wedge TruckIn(t^*, paris, s)]$	:	9
$\neg$“ $\wedge \exists b. [\exists t.\, BoxOn(b, t, s) \wedge TruckIn(t, paris, s)]$	:	$10 - 10p$
$\neg$“	:	0

This is a fascinating result that symbolically represents the relevant information for this backup for any domain size. That is, as defined previously $(1 - p)$ represents the chance that a box falls off a truck in Paris and by definition approaches 1 as the number of boxes on trucks in Paris approaches ∞. This exactly reflects the fact that the more boxes there are on trucks in Paris, the higher the chance that any one of them can independently fall off the truck. This result is in the original spirit of symbolic dynamic programming, which intended to compute and represent such information symbolically. In addition, we note that due to the dependence on p, whose value is domain-instance dependent, we must treat this as a parameterized case statement as defined previously.

In concluding this example, we note that most of the reasoning to obtain this result was ad-hoc in the sense that we have not provided formal algorithms for automating it. However, we remark at this early stage of investigation into factored FOMDPs that our goal is simply to demonstrate various types of structure that can be exploited.

Example of Backup with Indefinite Structure

In the case of SYSADMIN, we have a reward defined with a sum aggregator, which gives us an entirely different kind of structure than we had in the case of F-BOXWORLD. We illustrate this notion with an example of the backup $B^{reboot(x)}[vCase^0(s)]$ for the SYSADMIN problem. Recall that $vCase^0(s) = rCase(s)$ from Equation 6.4. Then we can write the backup as follows

where n is the number of computers in the domain:[12]

$$B^{reboot(x)}[vCase^0(s)] = \gamma \bigoplus_{a_1 \in \{rebootS(c_1), rebootF(c_1)\}, \ldots, a_n \in \{rebootS(c_n), rebootF(c_n)\}} \left[\left(\prod_{i=1}^{n} pCase(a_i | reboot(x)) \right) \otimes \sum_{c_i \in Comp} \boxed{\begin{array}{l} Regr[Up(c_i, s), a_1 \circ \cdots \circ a_n] \ : \ 1 \\ Regr[\neg Up(c_i, s), a_1 \circ \cdots \circ a_n] : \ 0 \end{array}} \right]$$

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

$$B^{reboot(x)}[vCase^0(s)] = \gamma \sum_{c_i \in Comp} \left[\bigoplus_{a_1 \in \{rebootS(c_1), rebootF(c_1)\}, \ldots, a_n \in \{rebootS(c_n), rebootF(c_n)\}} \left(\prod_{i=1}^{n} pCase(a_i | reboot(x)) \right) \otimes \boxed{\begin{array}{l} Regr[Up(c_i, s), a_1 \circ \cdots \circ a_n] \ : \ 1 \\ Regr[\neg Up(c_i, s), a_1 \circ \cdots \circ a_n] : \ 0 \end{array}} \right]$$

This last step is extremely important because it captures the factored probability distribution $\prod$ inside a specific c_i being summed over. Thus, for all SYSADMIN problems, we now exploit the fact that we can prove $Irr[Up(c_i, s), reboot_1(c_j)]$ for all $i \neq j$. This gives substantial simplification via Axiom 6.2.2:

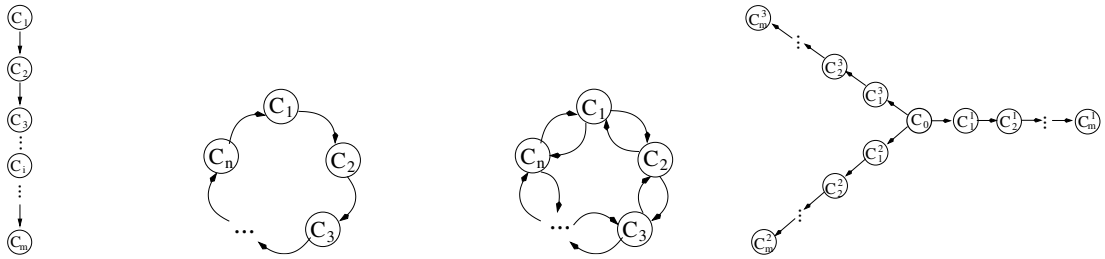
$$B^{reboot(x)}[vCase^0(s)] = \gamma \sum_{c_i \in Comp} \left[\bigoplus_{a \in \{rebootS(c_i), rebootF(c_i)\}} pCase(a | reboot(x)) \otimes \boxed{\begin{array}{l} Regr[Up(c_i, s), a] \ : \ 1 \\ Regr[\neg Up(c_i, s), a] : \ 0 \end{array}} \right]$$

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

$$B^{reboot(x)}[vCase^0(s)] = \gamma \sum_{c_i \in Comp} \left[pCase(rebootS(c_i) | reboot(x))) \otimes \boxed{\begin{array}{l} Regr[Up(c_i, s), rebootS(c_i)] \ : \ 1 \\ Regr[\neg Up(c_i, s), rebootS(c_i)] : \ 0 \end{array}} \right.$$
$$\left. \oplus \ pCase(rebootF(c_i) | reboot(x)) \otimes \boxed{\begin{array}{l} Regr[Up(c_i, s), rebootF(c_i)] \ : \ 1 \\ Regr[\neg Up(c_i, s), rebootF(c_i)] : \ 0 \end{array}} \right]$$

Recalling the results of regression from Eqs. 6.24 and 6.25, we see that the regressed top product simplifies to 1 and the regressed bottom product simplifies to 0. Now, recalling the defi-

[12] Here, we write Nature's choice sub-actions with a since for this problem we have previously assumed the number of computers is n.



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

Figure 6.1: Diagrams of the example SYSADMIN connection topologies that we focus on in this document.

nition of $pCase(rebootS(c)|reboot(x))$ from Eqs. 6.10 and 6.11, we obtain the final pleasing result:

$$B^{reboot(x)}[vCase(s)] = \gamma \sum_{c_i \in Comp} pCase(rebootS(c_i)|reboot(x)) \tag{6.44}$$

$$= \gamma \sum_{c_i \in Comp} \begin{cases} x = c_i & \boxed{\top : 1} \\ x \neq c_i & \begin{array}{|l} \hline Up(c_i, s) \;\; : \; 0.95 \\ \hline \neg Up(c_i, s) : \; 0.05 \\ \hline \end{array} \end{cases} \otimes \frac{1+\sum_d \left(\begin{array}{|l} \hline Conn(d, c_i) \wedge Up(d, s) \;\; : \; 1 \\ \hline \neg Conn(d, c_i) \vee \neg Up(d, s) : \; 0 \\ \hline \end{array} \right)}{1+\sum_d \left(\begin{array}{|l} \hline Conn(d, c_i) \;\; : \; 1 \\ \hline \neg Conn(d, c_i) : \; 0 \\ \hline \end{array} \right)}$$

Example of Symbolic Maximization with Indefinite Structure

To complete the factored symbolic dynamic programming (SDP) step we would first seek to compute $B^{reboot}[vCase^0(s)]$. However, noting our previous discussion, this is difficult to do in a simplified closed form since the result of $B^{reboot(x)}[vCase^0(s)]$ contains indefinite additive structure. Consequently, we need to take the indirect route of determining $B^{reboot}[vCase^0(s)]$ via a policy as previously described.

To simplify the example, we make additional domain constraints that restrict our network configuration to the simple unidirectional ring topology from Figure 6.1(b) where each computer c_{i-1} is connected to c_i (where subtraction is modulo n).

In this case, we can then simplify Equation 6.44 down to the following representation:

$$B^{reboot(x)}[vCase^0(s)] = \gamma \sum_{c_i \in Comp} \begin{array}{|lr|} \hline x = c_i & : \ 1.0 \\ Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.475 \\ \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.025 \\ \hline \end{array} \tag{6.45}$$

And assuming we have a $noop$ action, which takes the same transition distribution as $reboot(x)$ minus the case of Equation 6.10 (since $noop$ has no parameter), we can compute $B^{noop}[vCase^0(s)]$:

$$B^{noop}[vCase^0(s)] = \gamma \sum_{c_i \in Comp} \begin{array}{|lr|} \hline Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.475 \\ \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.025 \\ \hline \end{array} \tag{6.46}$$

Thus, we seek to compute an $aCase(s)$ instance of an advantage function $ADV(s)$ as defined previously. For $reboot(x)$, we can do this as follows:

$$aCase^t_{reboot(x) \succ noop}(s) = \text{casemax}\, \exists \vec{x} \ \left(B^{A(\vec{x})}[vCase^{t-1}(s)] \ominus qCase^t(s, noop) \right)$$

$$= \exists x. \sum_{c_i \in Comp} \begin{array}{|lr|} \hline x = c_i \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.975 \\ x = c_i \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ x = c_i \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.525 \\ x = c_i \wedge Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ x \neq c_i \wedge Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0 \\ x \neq c_i \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0 \\ x \neq c_i \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0 \\ x \neq c_i \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0 \\ \hline \end{array}$$

$$= \begin{array}{|lr|} \hline (\exists x. \, x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.975 \\ \neg \text{“} \wedge (\exists x. \, x = c_i) \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ \neg \text{“} \wedge (\exists x. \, x = c_i) \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.525 \\ \neg \text{“} \wedge (\exists x. \, x = c_i) \wedge Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ \hline \end{array}$$

While initially we had the $\exists x.$ on the outside of the $\sum_c$, we noted that only one of the cases

could have $x = c_i$, with the rest where $x \neq c_i$ contributing 0 and therefore removable from the $\sum_c$. This leaves us with one remaining case (when $x = c_i$), which we then existentially quantified. And as a reality check, we note that this advantage functions makes sense — the more computers that are down in a single unidirectional connection, the more advantageous it is to reboot one of those computers.

Since we only have a $reboot(x)$ and a $noop$ action in SYSADMIN, we note that by the previous discussion, the policy for $reboot(x)$ can then be represented in the following manner using the action precedence $noop \succ reboot$:

$$\pi Case^1(s)_{reboot} = aCase^t_{reboot(x) \succ noop}(s) > aCase^t_{noop \succ noop}(s) \tag{6.47}$$

$$= aCase^t_{reboot(x) \succ noop}(s) > \boxed{\top : 0} \tag{6.48}$$

$$= \begin{array}{|l|} \hline (\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) \quad : 1 \\ \hline \neg \text{“} \wedge (\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s) : 1 \\ \hline \neg \text{“} \wedge (\exists x.\, x = c_i) \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s) : 1 \\ \hline \neg \text{“} \wedge (\exists x.\, x = c_i) \wedge Up(c_i, s) \wedge Up(c_{i-1}, s) \quad : 1 \\ \hline \end{array} \tag{6.49}$$

This policy is essentially a decision list that prioritizes rebooting computers based on their status and the status of their upstream neighbor.

We note that in computing the $noop$ policy as previously described, we will get the following result

$$\pi Case^1(s)_{noop} = \boxed{\top : 1} \ominus \pi Case^1(s)_{reboot} \tag{6.50}$$

$$= \boxed{\top : 1} \ominus \boxed{\top : 0} \tag{6.51}$$

$$= \boxed{\top : 0} \tag{6.52}$$

since $\pi Case^1(s)_{reboot}$ exhaustively partitions the entire state space.

Now, we need to extract a free variable policy for $\pi Case^1(s)_{reboot(x)}$ as described previ-

ously, which gives us the following representation:

$$\pi Case^1(s)_{reboot(x)} = \begin{array}{|l|} \hline (x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) \quad : \ 1 \\ \hline \neg[(\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s)] \\ (x = c_i) \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s) \quad : \ 1 \\ \hline \neg[(\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s)] \\ \neg[(\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s)] \\ (x = c_i) \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s) \quad : \ 1 \\ \hline \neg[(\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s)] \\ \neg[(\exists x.\, x = c_i) \wedge \neg Up(c_i, s) \wedge Up(c_{i-1}, s)] \\ \neg[(\exists x.\, x = c_i) \wedge Up(c_i, s) \wedge \neg Up(c_{i-1}, s)] \\ (x = c_i) \wedge Up(c_i, s) \wedge Up(c_{i-1}, s) \quad : \ 1 \\ \hline \end{array} \tag{6.53}$$

And finally, based on Equation 6.39, we arrive at the following representation for $vCase^1(s)$ for SYSADMIN under the unidirectional ring constraints:

$$\begin{aligned} vCase^1(s) &= rCase(s) \oplus B^{\pi Case^1}[vCase^0(s)] \\ &= rCase(s) \oplus \exists \vec{x}.\ \left[\pi Case^1_{reboot(x)} \otimes B^{reboot(x)}[vCase(s)]\right] \\ &= \sum_{c \in Comp} \begin{array}{|l|} \hline Up(c, s) \ : \ 1 \\ \hline \neg Up(c, s) : \ 0 \\ \hline \end{array} \oplus \exists x. \left[\gamma \sum_{c \in Comp} \begin{array}{|ll|} \hline x = c_i & : \ 1.0 \\ \hline Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ \hline Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.475 \\ \hline \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ \hline \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.025 \\ \hline \end{array} \right] \end{aligned}$$

Note that we cannot get rid of the sum aggregator structure as this inherently defines the value function structure of SYSADMIN. Nonetheless, this is a purely symbolic representation of the maximum value achievable for $vCase^1(s)$ that has explicitly computed the casemax and thus is amenable to further factored symbolic dynamic programming steps.

6.3 Linear-value Approximation for (some) Factored FOMDPs

We note that in factored SDP, the value function representation often blows up uncontrollably in only a few steps. This is especially true when the results of factored SDP iterations introduce parameterized or sum/product aggregator structure since simplifying this structure and

maintaining a compact representation is difficult under these circumstances. Furthermore, it is not always clear how to explicitly compute the result of factored SDP without leaving the casemax and possibly the $\exists \vec{x}$ operators in symbolic form, which can complicate subsequent factored SDP iterations.

Given this representational blowup and inability to simplify, this suggests that we might want to use linear-value function approximation methods. Perhaps the most important advantage of such an approach w.r.t. factored FOMDPs as previously noted in Chapter 5 is that it does not require simplification, just the estimation of good weights. And as we will see, even when we can't explicitly compute the casemax or $\exists \vec{x}$ and must leave these operators in their original symbolic form, we may still be able to efficiently evaluate the linear programs at the heart of linear-value approximation methods.

While we will consider indefinite structure in our discussion of linear-value approximation, we note that the linear-value approximation approaches that we consider here currently *prohibit* the solution of problems such as F-BOXWORLD that introduce parameterized case structure in their backups. A linear-value approximation solution for such problems would require solving *parameterized (first-order) linear programs* that specify LPs in terms of free parameters such as domain and state properties (e.g., the number of boxes on trucks in Paris in a given state). In general the solution to parameterized LPs appears to be an open problem in the literature and we do not attempt to address this issue in this thesis. Consequently, from this point on, we only consider factored FOMDPs with indefinite sum and product aggregator structure that do not induce parameterized case structure. And for this reason, we exclude F-BOXWORLD from future examples and focus solely on SYSADMIN.

6.3.1 Linear-value Representation

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

$$V(s) = \bigoplus_{i=1}^{k} w_i \sum_{c} b_i(c, s) \tag{6.54}$$

For example, we can use the following $vCase(s)$ instance of $V(s)$ to represent the value

function

$$vCase(s) = w_1 \sum_c bCase_1(c, s) \oplus w_2 \sum_c bCase_2(c, s), \tag{6.55}$$

which accounts for the single (unary) and pair (binary) basis functions commonly used in the SYSADMIN literature [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001] where $bCase_i(c, s)$ are instances of $b_i(c, s)$ and parameters are tied for each of the unary and pair basis function classes:

$$bCase_1(c, s) = \boxed{\begin{array}{|l} Up(c, s) \quad : \ 1 \\ \hline \neg Up(c, s) : \ 0 \end{array}} \tag{6.56}$$

$$bCase_2(c, s) = \boxed{\begin{array}{|l} Up(c, s) \wedge \exists c_2 Conn(c, c_2) \wedge Up(c_2) \quad : \ 1 \\ \hline \neg(\, Up(c, s) \wedge \exists c_2 Conn(c, c_2) \wedge Up(c_2)) : \ 0 \end{array}} \tag{6.57}$$

There are a few motivations for this value representation:

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

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

As far as automatically constructing these basis functions is concerned, we note that we can attempt to generalize regression-based techniques from Section 5.4 of Chapter 5. In this case, however, the selection of which deterministic (joint) actions to regress through is a bit more challenging. In Chapter 5, we typically had a few deterministic outcomes per stochastic action whereas here we can have an indefinite number of deterministic joint actions. If we do not know which subset of joint actions to consider for basis function generation, we may just choose to use the $B^A[\cdot]$ backup operator to derive potential basis functions (when it can be computed). Since the most difficult and combinatorically explosive part of factored SDP

is computing the symbolic maximization over multiple actions, generating our basis functions using the backup operators in this way would still save us from performing this task.

For example, we note that the previous two basis functions we specified for SYSADMIN were (1) the reward $rCase(s)$ and (2) the result of computing $B^{noop}[rCase(s)]$. In general, the single, pairwise, triple, etc. basis functions used in the SYSADMIN literature can be derived by the $B^{noop}[\cdot]$ operator in this way. While this approach seems effective for SYSADMIN, it is not clear to what extent such a simple approach will generalize to other problems.

6.3.2 Factored First-order Approximate Linear Programming

Now, we generalize the first-order approximate linear programming (FOALP) approach from Chapter 5 to the case of factored FOMDPs. Simply substituting appropriate factored FOMDP structure into Equation 5.13 we can specify the following factored FOALP (fFOALP) solution for factored FOMDPs in terms of a first-order linear program where $R(s)$ is our case representation of reward:

$$\begin{aligned}
&\text{Variables: } w_i \; ; \;\; \forall i \le k \\
&\text{Minimize: } \sum_s \bigoplus_{i=1}^{k} w_i \cdot \sum_{\vec{c}} b_i(\vec{c}, s) \\
&\text{Subject to: } 0 \ge R(s) \oplus B^A \left[\bigoplus_{i=1}^{k} \cdot \sum_{\vec{c}} b_i(\vec{c}, s) \right] \\
&\qquad\qquad \ominus \bigoplus_{i=1}^{k} \cdot \sum_{\vec{c}} b_i(\vec{c}, s) \; ; \; \forall \, A, s
\end{aligned} \tag{6.58}$$

As before we note that our objective is an indefinite summation (over all situations s) and we have an indefinite number of constraints (one for each situation s). However, in the factored FOMDP formalism, we can now also have indefinite sum and product aggregator structure in the objective and constraints owing to the possibility of such structure in the reward and value representations. For example, for SYSADMIN, we are using a reward $R(s) = rCase(s)$ from Equation 6.4 defined with a sum aggregator and we are using a linear-value approximation of $vCase(s)$ from Equation 6.55 also based on the sum aggregator.

If the objective has no sum or product aggregator structure then we can simply use the same approximation as used for FOALP in Chapter 5. We note that if our linear-value representation does not contain product aggregators (i.e., we do not use them in the basis functions $b_i(s)$ that

we have chosen), we need not consider such structure in the objective. We can handle the remaining problem of indefinite sum aggregator structure in the objective fairly easily with an approximation. We compute an approximation of the objective in fFOALP following the approach for FOALP given in Chapter 5, Section 5.2.1. Exploiting commutativity of $\oplus$ with $\sum$, we approximate the above fFOALP objective as follows:

$$\sum_s \bigoplus_{i=1}^{k} w_i \cdot \sum_{\vec{c}} b_i(\vec{c}, s) = \bigoplus_{i=1}^{k} w_i \cdot \sum_s \sum_{\vec{c}} b_i(\vec{c}, s)$$
$$\sim \sum_{i=1}^{k} w_i \cdot |\vec{c}| \cdot \sum_{\langle \phi_j, t_j \rangle \in b_i} \frac{t_j}{|b_i|} \tag{6.59}$$

In the last step, we made an additional assumption that each of the $\vec{c}$ in the $\sum_{\vec{c}} b_i(\vec{c}, s)$ should be weighted uniformly and thus remove the $\sum_{\vec{c}}$ and replace it with a constant multiplier $|\vec{c}|$, which represents the number of ground basis functions for the basis function class b_i. If all basis functions use the same $\sum_{\vec{c}}$, then $\vec{c}$ becomes a constant multiplier that we can factor out of the objective. Otherwise, to determine $|\vec{c}|$, we need to know the actual size of domain object classes.[13] Here, $|b_i|$ represents the number of partitions in b_i and for each basis function, we sum over the value t_j of each partition $\langle \phi_j, t_j \rangle \in b_i$ normalized by $|b_i|$. This gives an approximation of the importance of each weight w_i in proportion to the overall value function.

Before we tackle the problem of solving an LP with indefinitely sized constraints, we introduce the factored generalization of first-order approximate policy iteration that will specify a constraint form similar to fFOALP.

6.3.3 Factored First-order Approximate Policy Iteration

Defining factored first-order approximate policy iteration (fFOAPI) turns out to be trivial given that we previously had to define the $B^\pi[\cdot]$ operator in our efforts to define symbolic maximization for factored FOMDPs with indefinite structure. The policy manipulation procedures to perform $B^\pi[\cdot]$ in Equation 6.38 exactly reflect what we need to do for fFOAPI and thus, we can immediately generalize the procedure given in Section 5.2.2 of Chapter 5.

As for API and FOAPI, in fFOAPI we calculate successive iterations of weights $w_j^{(i)}$ that represent the best approximation of the fixed-point value function for policy $\pi^{(i)}(s)$ at iteration i. We do this by performing the following two steps at every iteration i after initializing $\vec{w}^{(0)} =$

[13] As we will discuss later when we evaluate the constraints, we will actually have this information since we make domain size assumptions in our fFOALP solution.

$\vec{0}$ and $i = 1$:

1. Obtain the policy $\pi^{(i)}(s)$ from the weights $\vec{w}^{(i-1)}$ using the procedure outlined in Section 6.2.3. From this, we can easily derive $\pi^{(i)}_{A(\vec{x}^*)}(s)$ for all actions A. We replace the 1 values in the $\pi^{(i)}_{A(\vec{x}^*)}(s)$ case partition values with 0 and discard the 0 value case partitions (as for FOAPI as discussed in Section 5.2.2, we need not generate and test constraints where the policy does not apply).

2. Solve the following first-order LP that determines the weights $\vec{w}^{(i)}$ for the L_∞ minimizing projection of the approximate value function for policy $\pi^{(i)}(s)$:

$$\begin{aligned} &\text{Variables: } w_1^{(i)}, \ldots, w_k^{(i)} \\ &\text{Minimize: } \beta^{(i)} \\ &\text{Subject to: } \beta^{(i)} \geq \left| \mathcal{R}(s) \oplus \exists \vec{x}^* \left(\pi^{(i)}_{A(\vec{x}^*)}(s) \oplus B^{A(\vec{x}^*)} \left[\bigoplus_{j=1}^{k} w_j^{(i)} \cdot \sum_{\vec{c}} b_j(\vec{c}, s) \right] \right) \right. \\ &\qquad \left. \ominus \bigoplus_{j=1}^{k} w_j^{(i)} \cdot \sum_{\vec{c}} b_j(\vec{c}, s) \right| ; \ \forall A, s \end{aligned} \tag{6.60}$$

3. If $\pi^{(i)}(s) = \pi^{(i)}(s)$ or $\beta^{(i)}$ is less than a prespecified tolerance then exit, else increment i and goto step (1).

We've reached convergence *if* $\pi^{(i)}(s) = \pi^{(i-1)}(s)$ (or equivalently $\vec{w}^{(i)} = \vec{w}^{(i-1)}$). And if convergence is reached, we conjecture that the loss bounds for API (Equation 2.19) generalize to this case through a generalization of Theorem 5.2.1.

On a final note, we observe that our constraints could contain both indefinite sum and product aggregator structure just as for fFOALP. As such, we need some method for solving an LP with such constraints, which we tackle next.

6.3.4 Constraint Generation with Indefinite Constraints

Now we turn to solving for maximally violated constraints in a constraint generation solution to the first-order LPs given in Eqs. 6.58 and 6.60. We make two assumptions here: (a) each basis function takes the form $\sum_{\vec{c}} b_i(\vec{c}, s)$ and (b) the reward takes the form $\sum_{\vec{c}} R(\vec{c}, s)$ where the $\sum_{\vec{c}}$ in (a) and (b) refer to the same object domain $\vec{c}$.[14]

[14]While we are not necessarily restricted to have such symmetrical structure in our reward and linear-value approximations, we note that the constraint generation methods outlined in this section can only generally be

Under these assumptions, we will often find that the resulting constraint structure of fFOALP (6.58) and fFOAPI (6.60) for each action A has the following generalized format where we have replaced the $\forall s$ with $\max_s$:

$$0 \geq \max_s \exists \vec{x} \left\{ case_1(\vec{x}, s) \oplus \ldots \oplus case_p(\vec{x}, s) \oplus \sum_{\vec{c}} \left[case_{p+1}(\vec{c}, \vec{x}, s) \oplus \ldots \oplus case_q(\vec{c}, \vec{x}, s) \right] \right\} \tag{6.61}$$

We cannot guarantee that this structure holds for a given problem as (c) we may not be able to fully reduce the indefinite product aggregator structure in the transition distribution to a finite product (by exploiting irrelevance or other properties) when computing the backup operators, (d) the backups may induce parameterized case structure, or (e) the policy may take the form of a single case statement with indefinite sum or product aggregator structure or parameterized structure. However, if (a) and (b) hold and *none* of (c), (d), or (e) occurs then we note that the above general constraint structure arises in the following way:

1. Recall that the case statement and the result of all operators applied to case statements can be written as a first-order formula (albeit a potentially indefinitely long formula if sum or product aggregator structure are present). Thus, we can "Prenex" the $\exists \vec{x}$ quantifier (implicit from the backup operations in fFOALP or explicitly stated in the constraint for fFOAPI) from each constraint to the front of our constraint representation.

2. Any non-sum aggregator structure in the constraints such as the policy in fFOAPI can be represented by $case_i(\vec{x}, s)$ for $1 \leq i \leq p$.

3. Under our assumptions, the backup can be distributed through the $\sum_{\vec{c}}$ into each basis function and any residual finite product structure $\otimes$ from the transition function can be explicitly computed, thus yielding sum aggregator structure that can be represented in the form $\sum_c case_j(\vec{c}, \vec{x}, s)$ for $(p+1) \leq j \leq q$. Then we can exploit commutativity of $\oplus$ to rewrite $\sum_{\vec{c}} case_{p+1}(\vec{c}, \vec{x}, s) \oplus \ldots \oplus \sum_{\vec{c}} case_q(\vec{c}, \vec{x}, s)$ in the representation of Equation 6.61.

We note that such a constraint structure holds for fFOALP applied to problems like SYSADMIN for which we provide a concrete example at the end of this section.

This constraint form is very similar to that solved for the first-order linear programs in Chapter 5 (c.f., Equation 5.22) with one important exception—here we have the addition of the

made to work under such assumptions.

sum aggregator which prevents us from achieving a finite representation of the constraints in all cases (recall that $\sum_{\vec{c}}$ is an indefinitely large sum).

While we could conceive of trying to find a finite number of constraints that closely approximate the form in Equation 6.61, it is not clear how to ensure a good approximation for all domain sizes. In fact, it is very easy to construct examples that have very different solutions for, say, even vs. odd sized domains so it is not clear that a generic domain-size independent solution should always be desired.[15] On the other hand, grounding these constraints for a specific domain instantiation is clearly not a good idea since this approach would scale proportionally to the domain size.

Fortunately, there is a middle ground that has received a lot of research attention very recently—first-order probabilistic inference (FOPI) [Poole, 2003; de Salvo Braz *et al.*, 2005]. In this approach, rather than making a *domain closure* assumption and grounding, a much less restrictive *domain size* assumption is made. This allows the solution to be carried out in a lifted manner and the solutions to be parameterized by the domain size. Recent work [de Salvo Braz *et al.*, 2006] has explicitly examined a "first-order" $\max$-$\sum$ cost network similar to Equation 6.61 that we would need to evaluate during constraint generation.[16]

Inversion Elimination

Braz *et al.* [de Salvo Braz *et al.*, 2005; de Salvo Braz *et al.*, 2006] introduce the FOPI techniques of *(partial) inversion elimination* and *counting elimination*. We do not use counting elimination here and thus do not cover it. However, we do use inversion elimination for constraint simplification, which we describe next.

Assuming that $case_P(c, s)$ only mentions relations or fluents in set P and $case_R(c, s)$ only mentions relations or fluents in set R s.t. $P \cap R = \emptyset$, we can perform the following inversion

[15] Imagine a logistics domain where a truck and a plane can deliver an item between cities connected in a straight line with the truck and plane at one end and the goal at the other. The truck, for some reason, can only move forward exactly two cities at a time (i.e., no odd-length moves) and the plane although having a very high cost, can move freely between any two cities. In the optimal solution, the truck is used for domains with an even number of cities and the plane used for an odd number of cities leading to very different values.

[16] It turns out that our $\max$-$\sum$ formalism is actually more general than FOPI in that we can represent not only *parameterized factors* over propositional variables, but also general first-order formulae within our case statements. Nonetheless, these FOPI techniques can be made to generalize to our framework.

elimination transform:

$$0 \geq \max_s \sum_c \left[case_P(c,s) \oplus case_R(c,s) \right] \tag{6.62}$$

$$= \left[\max_s \sum_c case_P(c,s) \right] \oplus \left[\max_s \sum_c case_R(c,s) \right] \tag{6.63}$$

This result follows simply from the fact that $case_P(c,s)$ and $case_R(c,s)$ can be maximized independently as they have no common structure which may constrain the joint evaluation of their maximal values.

Now we introduce two additional elimination techniques in order to demonstrate an efficient solution to SYSADMIN.

Existential Elimination

We introduce *existential elimination* in order to exploit a powerful transformation for rewriting an $\exists \vec{x}$ operator in a concise $\sum_{\vec{c}} case(\vec{c})$ format. In what follows, we handle the case for a single $\exists x$ since it can be applied sequentially for each quantified variable in the case of $\exists \vec{x}$.

We assume that we are given a constraint of the following form

$$0 \geq \max_s \left\{ \exists x. \sum_{c_i \in C} case(c_i, x, s) \right\} \tag{6.64}$$

where each $case(c_i, x, s)$ is restricted to reference x in its case partition formulae using only the test $x = c_i$ (or by negation $x \neq c_i$). For example, $case(c_i, x, s)$ could have the following structure:

$$case(c_i, x, s) = \begin{cases} x = c_i \wedge \phi_1(s) & : \quad t_1 \\ \vdots & : \quad \vdots \\ x = c_i \wedge \phi_i(s) & : \quad t_i \\ x \neq c_i \wedge \phi_{i+1}(s) & : \quad t_{i+1} \\ \vdots & : \quad \vdots \\ x \neq c_i \wedge \phi_n(s) & : \quad t_n \end{cases} \tag{6.65}$$

Our ultimate goal is to be able to rewrite the constraint in Equation 6.64 in an equivalent form that does not contain the $\exists x$ operator. We describe how to do this in the following discussion by introducing additional variables and case summands into the constraint structure and making

substitutions in our original $case(c_i, x, s)$ representation.

To solve this problem, we will assume that we have a transitive ordering $\succ$ (with equality: $\succeq$) on all of the elements in the domain of $C \cup \{c_{n+1}\}$ where $|C| = \{c_1, \ldots, c_n\}$ such that $c_1 \succ c_2 \succ \cdots \succ c_n \succ c_{n+1}$. We define a function $next(\cdot)$ that specifies the next element in the order such that $next(c_i) = c_{i+1}\ ;\ 1 \leq i \leq n$. And we introduce a new variable $b(c_i)$ whose definition is the following:

$$\forall c \in C.\ b(c_i) \equiv x = c \wedge c \succ c_i$$

In words, $b(c_i)$ is defined as equivalent to the statement "$x = c$ for some c coming *before* c_i in the ordering." The beauty of this definition is that we can use it to redefine $x = c_i$ in the following manner:

$$\forall c \in C.\ [x = c] \equiv [\neg b(c) \wedge b(next(c))]$$

This equivalence is most obvious in words: "$x = c$ is the same as c being chosen before $next(c)$, but not before c in the ordering." Now we can rewrite $case(c_i, x, s)$ as $case'(c_i, c_{i+1}, s)$ where we substitute every occurrence of $x = c$ and $x \neq c$ with this equivalent definition (that does not contain the variable x, thus allowing us to remove the vacuous quantifier $\exists x$). Rewriting the $case(c, x, s)$ statement from Equation 6.65, we would obtain the following:

$$case'(c_i, c_{i+1}, s) = \begin{array}{|lcl|}\hline \neg b(c_i) \wedge b(next(c_i)) \wedge \phi_1(s) & : & t_1 \\ \hline : & : & : \\ \hline \neg b(c_i) \wedge b(next(c_i)) \wedge \phi_i(s) & : & t_i \\ \hline \neg(\neg b(c_i) \wedge b(next(c_i))) \wedge \phi_{i+1}(s) & : & t_{i+1} \\ \hline : & : & : \\ \hline \neg(\neg b(c_i) \wedge b(next(c_i))) \wedge \phi_n(s) & : & t_n \\ \hline \end{array} \tag{6.66}$$

Now, if only we could enforce the definition of $b(c_i)$ while ensuring that at least one $x = c_i$ was chosen for $c_1 \succeq c_i \succ c_{n+1}$ (to enforce the semantics of $\exists x$), then we would have an equivalent rewrite of our constraint without the variable x. To do this, we begin by defining the

following two axioms:

$$\forall c \in C.\ (b(c) \supset b(next(c)))$$
$$\neg b(c_1) \wedge b(c_{n+1})$$

The first axiom states in words that if $x = c$ for some c that occurs before c_i then c also occurs before $next(c_i)$. If this axiom is satisfied for all $c \in C$, it ensures that the definition of $b(c_i)$ is satisfied. The second axiom states in words that $x = c_i$ for some c_i where $c_1 \succeq c_i \succ c_{n+1}$, thus enforcing that $x = c_i$ holds true for at least one c_i.

Now we introduce an expression that encodes the above two axioms and takes the value 0 when both of these axioms are satisfied:

$$\begin{array}{|l:r|}\hline \neg b(c_1) \wedge b(c_{n+1}): & 0 \\ \hline b(c_1) \vee \neg b(c_{n+1}): & -\infty \\ \hline \end{array} \oplus \sum_{c_i \in C} \begin{array}{|l:r|}\hline b(c_i) \supset b(next(c_i) & : \quad 0 \\ \hline \neg(b(c_i) \supset b(next(c_i)): & -\infty \\ \hline \end{array}$$

Having done this, we achieve our goal by rewriting the original constraint without $\exists x$ where by construction, if its maximal value is greater than $-\infty$, then the original form of the constraint with the $\exists x$ is satisfied:

$$0 \geq \max_s \left\{ \left(\sum_{c_i \in C} case'(c_i, c_{i+1}, s) \ \oplus \sum_{c_i \in C} \begin{array}{|l:r|}\hline b(c_i) \supset b(next(c_i)) & : \quad 0 \\ \hline \neg(b(c_i) \supset b(next(c_i))): & -\infty \\ \hline \end{array} \right) \right.$$
$$\left. \oplus \begin{array}{|l:r|}\hline \neg b(c_1) \wedge b(c_{n+1}): & 0 \\ \hline b(c_1) \vee \neg b(c_{n+1}): & -\infty \\ \hline \end{array} \right\}$$

Linear Elimination

We next introduce an elimination technique intended to exploit symmetry in special cases of the $\max$-$\sum$ problem. In Figure 6.2, we are given a sum over c_i of case statements of the form $case(c_i, c_{i+1})$ (where c_i and c_{i+1} are consecutive w.r.t. some total order). We generally refer to this summation form as *linearly connected* since each c_i co-occurs with c_{i-1} in $case(c_{i-1}, c_i)$ and c_{i+1} in $case(c_i, c_{i+1})$ (except for the first and last variables which each only occur in one $case(c_i, c_{i+1})$ summand). Our goal is to compute the max over all variables except the first and last, each previous solution can be used to *double* the size of the next solution due to the *symmetry* inherent in the elimination. As shown, $r(2) = \max_{x_2} \sum_{i=1}^{2} case_i$ and is structurally identical to $\max_{x_4} \sum_{i=3}^{4} case_i$ modulo variable renaming, thus leaving x_3 to be eliminated from the sum to obtain $r(4)$. Applying the same elimination again to $r(4)$ yields $r(8)$ and so on. In

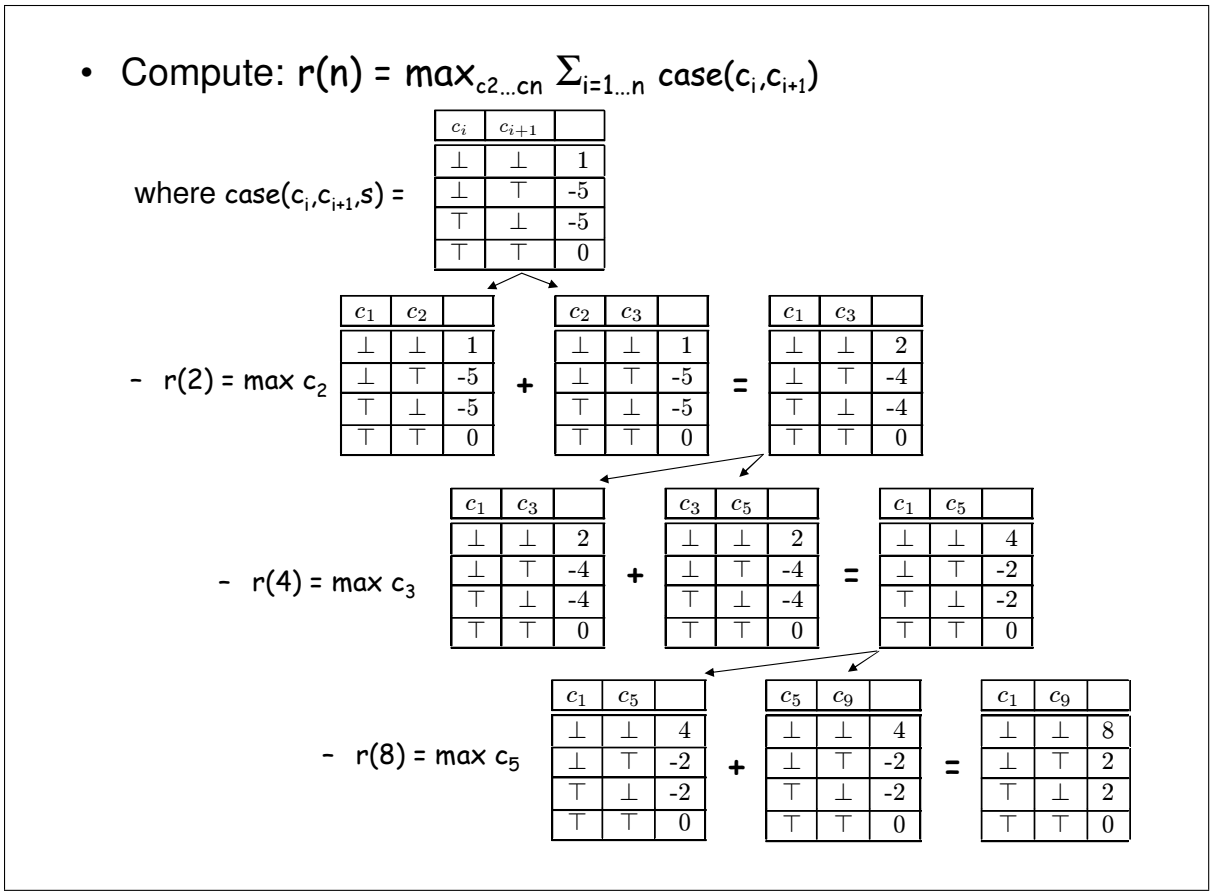


Figure 6.2: An example of *linear elimination*.

general, $r(2^n)$ can be computed directly from $r(2^{n-1})$ in this manner. Thus, the elimination can be done in $O(\log n)$ space and time and the maximizing instantiations can be represented in $O(\log n)$ space also due to the inherent symmetry of the variable assignments.

Example of Linear and Existential Elimination

Having described a generic constraint structure and various elimination techniques intended to efficiently find the maximally violated constraint under domain size assumptions, we now provide an example of this applied to SYSADMIN. To simplify our exposition, we use the uni-directional ring topology constraints of Figure 6.1(b) and we use just the unary basis function class $bCase_1(c, s)$ from Equation 6.56.

We note that the following techniques efficiently generalize to the pairwise, triple, etc. basis functions discussed previously since they all make a linearly connected assumption that leads to symmetry that can be exploited by both linear and existential elimination. And for more complex network topologies that include lines (or rings) as a fundamental building block, these

networks can be decomposed and linear and existential elimination applied to each line with the results easily pieced together.

With the prior assumptions, we now examine the fFOALP constraint structure that we would get for the $reboot(x)$ action for SYSADMIN:

$$0 \geq rCase(s) \oplus B^{reboot} \left[\bigoplus_{i=1}^{k} w_i \cdot bCase_i(s) \right] \ominus \bigoplus_{i=1}^{k} w_i \cdot bCase_i(s) \; ; \; \forall s$$

Now we expand out each of the case statements into their actual representation and substitute our single basis linear value function representation. We also rewrite our constraints in terms of $\max_s$ to get rid of the $\forall s$:

$$0 \geq \max_s \left\{ \sum_{c_i \in Comp} \begin{array}{|l} Up(c_i, s) \;\; : \; 1 \\ \hline \neg Up(c_i, s) : \; 0 \end{array} \oplus B^{reboot} \left[w_1 \sum_{c_i \in Comp} \begin{array}{|l} Up(c_i, s) \;\; : \; 1 \\ \hline \neg Up(c_i, s) : \; 0 \end{array} \right] \right.$$
$$\left. \oplus w_1 \sum_{c_i \in Comp} \begin{array}{|l} Up(c_i, s) \;\; : \; 1 \\ \hline \neg Up(c_i, s) : \; 0 \end{array} \right\}$$

Noting that the result of $B^{reboot(x)}$ for this particular case was given in Equation 6.45, we can easily derive B^{reboot} by existentially quantifying it, so we substitute it in to obtain:

$$0 \geq \max_s \left\{ \sum_{c_i \in Comp} \begin{array}{|l} Up(c_i, s) \;\; : \; 1 \\ \hline \neg Up(c_i, s) : \; 0 \end{array} \oplus w_i \cdot \gamma \, \exists x. \sum_{c_i \in Comp} \begin{array}{|ll} \hline x = c_i & : \; 1.0 \\ \hline Up(c_i, s) \wedge Up(c_{i-1}, s) & : \; 0.95 \\ \hline Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \; 0.475 \\ \hline \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \; 0.05 \\ \hline \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \; 0.025 \\ \hline \end{array} \right.$$
$$\left. \oplus w_1 \sum_{c_i \in Comp} \begin{array}{|l} Up(c_i, s) \;\; : \; 1 \\ \hline \neg Up(c_i, s) : \; 0 \end{array} \right\}$$

Next we apply existential elimination to get rid of the $\exists x$ assuming that we have defined

$next(c_i) = c_{i-1}$ and exploit commutativity of $\oplus$ to reorganize the $\sum_{c_i \in Comp}$:

$$0 \geq \max_s \left\{ \sum_{c_i \in Comp} \left(\begin{array}{|l|} \hline Up(c_i, s) \ : \ 1 \\ \hline \neg Up(c_i, s) : \ 0 \\ \hline \end{array} \oplus w_i \cdot \gamma \begin{array}{|l|r|} \hline \neg b(c_i) \wedge b(c_{i-1}) & : \ 1.0 \\ \hline Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.95 \\ \hline Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.475 \\ \hline \neg Up(c_i, s) \wedge Up(c_{i-1}, s) & : \ 0.05 \\ \hline \neg Up(c_i, s) \wedge \neg Up(c_{i-1}, s) & : \ 0.025 \\ \hline \end{array} \right. \right.$$

$$\left. \left. \oplus w_1 \begin{array}{|l|} \hline Up(c_i, s) \ : \ 1 \\ \hline \neg Up(c_i, s) : \ 0 \\ \hline \end{array} \oplus \begin{array}{|l|r|} \hline b(c) \supset b(c_{i-1}) & : \ 0 \\ \hline b(c) \wedge \neg b(c_{i-1}) & : \ -\infty \\ \hline \end{array} \right) \oplus \begin{array}{|l|r|} \hline \neg b(c_n) \wedge b(c_1) & : \ 0 \\ \hline b(c_n) \vee \neg b(c_1) & : \ -\infty \\ \hline \end{array} \right\}$$

From this point, we have a representation that is directly amenable to the application of linear elimination. To see this, we explicitly compute the "cross-sum" of the four case statements inside the parens $(\cdot)$. Then we merge each $b(c_i)$ and $Up(c_i, s)$ into a single 4-valued variable v_i with domain as follows

$$\begin{aligned} v_i \in \{ &\langle b(c_i) = \top, Up(c_i, s) = \top \rangle, \langle b(c_i) = \top, Up(c_i, s) = \bot \rangle, \\ &\langle b(c_i) = \bot, Up(c_i, s) = \top \rangle, \langle b(c_i) = \bot, Up(c_i, s) = \bot \rangle \} \end{aligned}$$

and rewrite this sum as $case_2(v_i, v_{i+1})$. For uniformity, we can easily rewrite the last non-$\sum_c$ term over $b(c_n), b(c_1)$ in the form $case_1(v_1, v_n)$. Thus, we obtain the following rewrites of the above constraint culminating in the final constraint form where we push the $\max$ in using the principles of variable elimination:

$$\begin{aligned} 0 &\geq \max_{v_1, v_n} \left\{ case_1(v_n, v_1) \oplus \left(\max_{v_2, \ldots, v_{n-1}} \sum_{i=1}^{n} case_2(v_i, v_{i-1}) \right) \right\} \\ &\geq \max_{v_1, v_n} \left\{ case_1(v_n, v_1) \oplus case_2(v_1, v_n) \oplus \left(\max_{v_2, \ldots, v_{n-1}} \sum_{i=2}^{n} case_2(v_i, v_{i-1}) \right) \right\} \\ &\geq \max_{v_1, v_n} \left\{ case_1(v_n, v_1) \oplus case_2(v_1, v_n) \oplus \left(\max_{v_2, \ldots, v_{n-1}} \sum_{i=1}^{n-1} case_2(v_{i+1}, v_i) \right) \right\} \end{aligned}$$

Now, we can determine the inner $\max_{v_2, \ldots, v_{n-1}} \sum_{i=1}^{n-1} case_2(v_{i+1}, v_i)$ by linear elimination in $O(\log n)$ time since this is essentially the form we evaluated in Figure 6.2 except that the variables v_i are quaternary rather than binary (which can be easily accommodated). The final and outer $\max_{v_1, v_n}$ is only over 2 variables and can be computed in constant time.

Therefore, once we have a weight instantiation during constraint generation, we can linearly evaluate the maximizing variable assignment for this constraint in $O(\log n)$ time. This gives us the maximizing value for the constraint and if it is a violation, we can easily extract the structure of this constraint to add it to our LP. Thus, we see that for fFOALP on this particular SYSADMIN problem, constraint generation takes $O(\log n)$ time per iteration.

As noted at the beginning of this subsection, these techniques generalize to larger sets of basis functions (pairs, triples, etc...) since these all exhibit the symmetry that can be exploited by linear and existential elimination. Furthermore, these techniques can also be applied to more complex network topologies that can be decomposed into lines by solving each decomposed piece separately and then piecing the solutions together.

6.4 Empirical Results

All of the solution techniques that we have described so far are targeted to very specific types of problem structure and we note that the collection of techniques we have presented is far from a universal solution. Our goal in our solution approaches was to scale sub-linearly in the FOMDP representation size, however, as we will discuss in the conclusion, the results of Jaeger [2000] imply this is generally impossible. As a consequence, our current implementation of linear-value approximation methods is geared specifically towards the types of problem structure that we have exploited above, i.e., SYSADMIN problems with network topologies consisting of linearly connected structure.

Given the general difficulty of automatically finding a compact policy representation in the policy-driven approaches of factored SDP and fFOAPI for SYSADMIN, the only practical first-order approach to solving this problem was fFOALP. We applied ALP and fFOALP solutions to the SYSADMIN problem configurations from Figure 6.1(a,b,d) using unary basis functions; each of these network configurations represents a distinct class of MDP problems with its own optimal policy. Solution times and empirical performance are shown in Figure 6.3. We did not tie parameters for ALP in order to let it exploit the properties of individual computers; had we done so, ALP would have generated the same solution as fFOALP.

The most striking feature of the solution times is the scalability of fFOALP over ALP. ALP's time complexity is $\Omega(n^2)$ since each constraint generation iteration must evaluate n ground constraints (i.e., n ground actions), each of length n (i.e., n basis functions). fFOALP avoids this complexity by using one backup to handle *all* possible action instantiations at once *and* exploiting the symmetric relational structure of the constraints by using existential and lin-

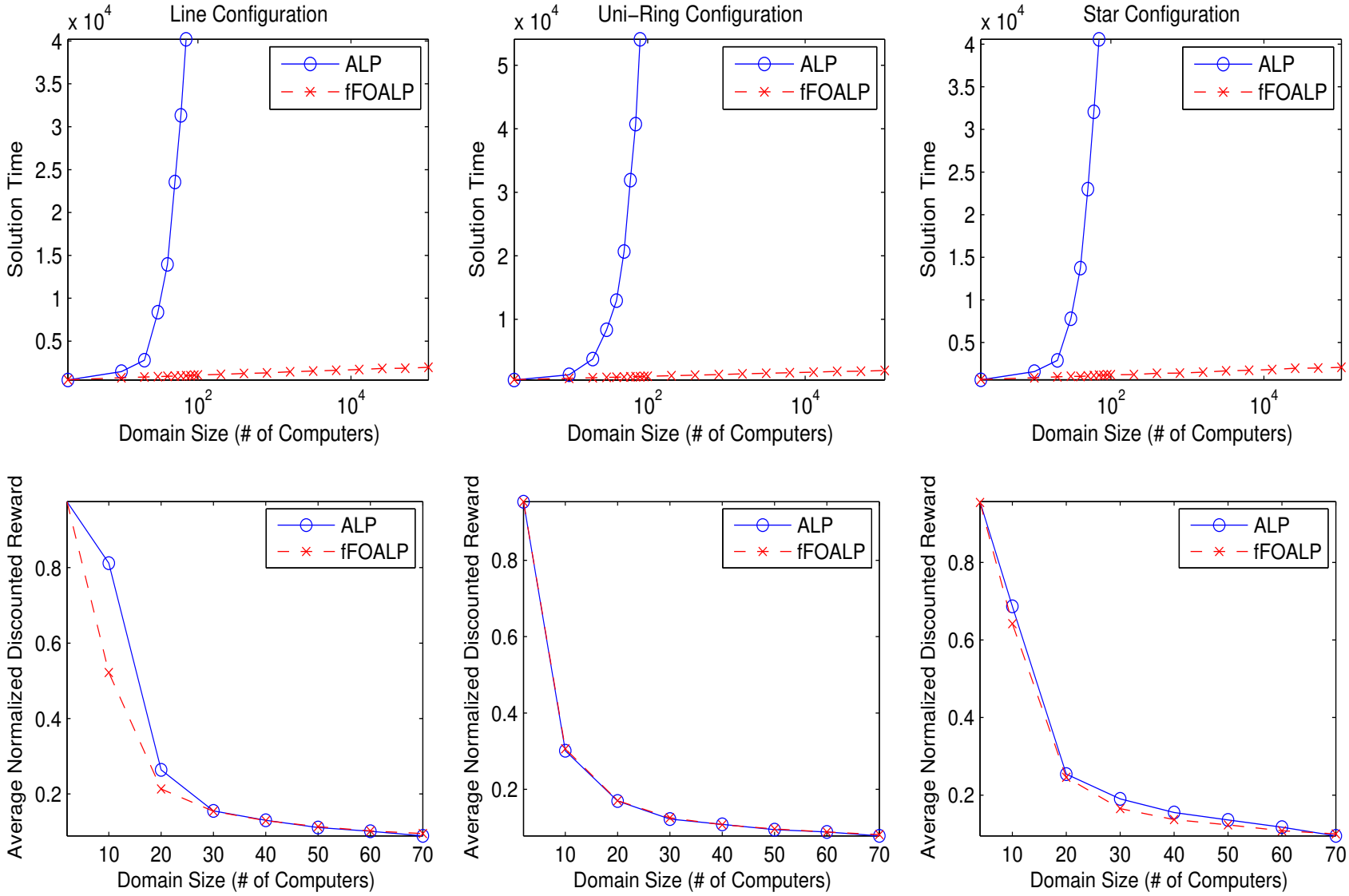


Figure 6.3: Factored FOALP and ALP solution times (top) and expected discounted reward divided by the maximum possible reward (bottom) averaged over 200 trials of 200 steps vs. domain size for various network configurations (left:line, middle:unidirectional-ring, right:star) in the SYSADMIN problem.

ear elimination (plus inversion elimination for the star network) to evaluate them in $O(\log n)$ time. Empirically, the fFOALP solutions to these SYSADMIN problems generate a constant number of constraints and since LPs are polynomial-time solvable, the complexity is thus polynomial in $\log n$.

In terms of performance, as the number of computers in the network increases, it becomes difficult to obtain a high reward since on average, more than one computer will fail on a given time step (for sufficiently large problems), yet only one computer can be rebooted at each time step. This leads to a necessary degradation of even the optimal policy value as the domain size increases. Comparatively though, the implicit parameter tying of fFOALP's basis function classes does not hurt it considerably in comparison to ALP; certainly, the difference becomes negligible for the networks as the domain size grows. This indicates that tying parameters across basis function classes may be a reasonable approach for large domains. Secondly, for completely symmetric cases like the unidirectional ring, we see that ALP and fFOALP produce exactly the same policy—albeit with fFOALP having produced this policy using much less computational effort.

We note that the FOMDP formalism from Chapter 4 (originally appearing in Boutilier *et al.* [2001]), the extensions for linear-value representation in Chapter 5, and all other FOMDP formalisms [Hölldobler and Skvortsova, 2004; Kersting *et al.*, 2004; Wang *et al.*, 2007] cannot compactly represent factored structure in FOMDPs. Other non-first-order approaches [Fern *et al.*, 2003; Gretton and Thiebaux, 2004; Guestrin *et al.*, 2003] require sampling where in the best case these approaches could never achieve sub-linear complexity in the sampled domain size.

6.5 Concluding Remarks

We have contributed the sum and product aggregator language extension for the specification of factored FOMDPs that were previously impossible to represent in a domain-independent manner as FOMDPs. And we have generalized symbolic dynamic programming to exploit novel definitions of first-order independence and sum/product aggregator structure. We have also shown how parameterized structure can arise in the solution of factored FOMDPs as it did for the F-BOXWORLD problem.

In addition to exact solution methods, we have generalized linear-value approximation solution techniques to handle factored FOMDPs. In many cases, the presence of sum and product aggregator structure in our factored FOMDP definition prevents us from obtaining a finite-

length form of the constraints in the linear program representations that arise from these solutions. Nonetheless, we showed that we can make a mildly restrictive domain size assumption and exploit the resulting constraint structure to efficiently evaluate them in a constraint generation framework without grounding. To do this, we borrowed from the first-order probabilistic inference (FOPI) framework [Poole, 2003; de Salvo Braz *et al.*, 2005] and introduced the novel existential and linear elimination techniques for respectively exploiting existential and linear structure in the evaluation of cost networks during constraint generation. Using these techniques, we empirically demonstrated that we can solve the SYSADMIN factored FOMDPs in time and space that scales polynomially in the logarithm of the domain size—results that were *impossible* to obtain for previous techniques that relied on grounding.

Unfortunately, while we have provided the representation and basic symbolic dynamic programming equations for factored FOMDPs, we have only begun to scratch the surface of their solution methods. Thus, this chapter should be viewed more as an introduction to the factored FOMDP representation and potential solution methods rather than a guide to its generic solution. A lot of the underlying theories are in development and the methods for manipulating and simplifying first-order case expressions are just beginning to be explored. It is beyond the scope of this thesis to address all of these topics comprehensively, nonetheless, it was our objective to provide an idea of what is possible with these novel solution approaches.

In conclusion, we do note one negative result that may partially explain our inability to provide comprehensive algorithms for factored FOMDP solutions. Jaeger [2000] proved that lifted inference in relational Bayes nets (i.e., those equivalent in expressive power to our $pCase$ representation for transition DBNs in factored FOMDPs, which condition probabilities on first-order formulae) cannot always be done exactly for trivial queries (i.e., determining the probability of a single ground atom) in a manner whose inferential complexity is less than that of performing inference in a fully grounded problem. Since inference in the backup operator for factored FOMDPs is at least as expressive as these trivial queries, this implies that we will not always be able to symbolically evaluate a factored transition distribution (or cost network constraint structure built from this distribution) in an asymptotically faster manner than that obtained by fully grounding it out — thus losing the benefits of the compact factored FOMDP representation.

Nonetheless, this does not preclude the possibility of restricted classes of structure for which we can obtain efficient solutions. As an example, for problem structures similar to the F-BOXWORLD and SYSADMIN problems discussed in this chapter, we note that there is much hope for efficient solutions. And one additional idea is that if we are willing to approximate

our model in order to fit into a class of efficiently solvable FOMDPs, then we can use a solution to this approximated model as guidance for other more computationally expensive algorithms ranging from seed values for value iteration to ground heuristic search to reward shaping in reinforcement learning [Ng *et al.*, 1999; Marthi, 2007]. Or we may just choose to act according to this approximated model, especially if we can obtain error bounds on performance [Dearden and Boutilier, 1997]. We revisit this idea in our concluding chapter since it outlines an effective framework for making use of FOMDPs and factored FOMDPs where they excel, while avoiding their use where they are problematic.

Chapter 7

Conclusions

We have come a long way since we first introduced the ground enumerated state MDP and its variant solutions as the basic model for representing and solving decision-theoretic planning problems. Since that point, we have introduced factored structure into MDPs and we have covered a variety of exact and approximate solution algorithms that exploit this factored MDP structure. We have also introduced the FOMDP representation to exploit some of the relational and first-order structure inherent in many planning problem representation languages such as PPDDL. And we have combined factored and first-order structure into a factored FOMDP model that combines the representational advantages of both.

Not only have we shown that various forms of factored and relational structure can be exploited in the concise and natural representation of MDPs — we have also demonstrated that this structure can be exploited in solution methods as well. We have introduced a variety of methods for exploiting structure in exact solution approaches and we have heavily motivated the linear-value approximation approach for exploiting all levels of MDP structure. For this approach, we have presented comparative results from the ICAPS International Probabilistic Planning Competitions that demonstrate that our first-order linear-value approximation approach is competitive with other state-of-the-art planners. And finally, we have also presented encouraging empirical results showing vast reductions in solution complexity on certain types of problems over less-structured approaches through the exploitation of various forms of structure — be it context-specific, additive or multiplicative independence, or the exploitation of factored MDP structure, first-order MDP structure, decomposable goal structure or combinations of the above.

Here, we review the major contributions of the thesis, outline some interesting directions for future work, and part with some concluding remarks on the framework of first-order decision-

theoretic planning in structured relational environments.

7.1 Summary of Contributions

In a thesis such as this one that draws on so much background work, it can be difficult to discern new contributions from prior work. Consequently, we briefly review some of the major contributions of this thesis:

1. **Affine Algebraic Decision Diagrams:** Having identified various shortcomings with the algebraic decision diagram (ADD) [Bahar *et al.*, 1993] representation, we introduced the affine ADD (AADD) to simultaneously exploit additive, multiplicative and context-specific independence in factored MDP representation and solution methods. We proved that the AADD never performs more than a constant factor worse in time and space than an ADD and can lead to an exponential-to-linear reduction in time and space over the ADD. And we presented a variety of empirical results suggesting that AADDs are often as good as or better than ADDs or tabular representations in the solution of factored MDPs. Unfortunately, a preliminary investigation into the use of AADDs for approximate inference in MDPs has not proved fruitful; yet approximation approaches seem crucial for scaling beyond the limits of exact inference while simultaneously mitigating issues of representational blowup that may occur due to numerical precision errors in the AADD computations.

2. **First-order Decision Diagrams:** In the same way that ADDs and AADDs exploit structure to compactly represent factored MDPs, we introduced first-order ADDs and first-order AADDs (collectively termed FO(A)ADDs) to represent structure in the case representation of FOMDPs. We introduced an additional type of structure — first-order context-specific independence — that can be exploited in these diagrams and we discussed how to perform case operations directly on this representation. We explored the difficulties that can occur with using FO(A)ADDs for all FOMDP solution computations and thus advocated a hybrid approach for their use that enabled the fully automated solution of a few example FOMDPs.

   One of the key components for producing compact FO(A)ADDs is the use of equality simplification procedures to (a) remove unneccessary variables and quantifiers and (b) to distribute quantifiers as deeply into a formula as possible in an effort to expose propositional structure in a first-order formula. Our current approach for simplification is based

on the heuristic search-based application of rewrite rules outlined in Chapter 4, but it is likely that there are better and more efficient methods for doing this.

3. **Additive Decomposition of Universal Rewards:** FOMDPs with universally quantified rewards pose a number of interesting difficulties for solution techniques based on the case representation or its decision diagram extensions. Having discussed these issues, we then proceeded to propose an additive goal decomposition approach to handling universal rewards motivated in part by the work of [Boutilier *et al.*, 1997; Singh and Cohn, 1998; Meuleau *et al.*, 1998b; Poupart *et al.*, 2002a]. This approach required an offline generic solution combined with an online additive decomposition of Q-values w.r.t. goals specified at run-time. Then we outlined an approach for exploiting the additivity of Q-values to efficiently perform run-time policy evaluation. We used these techniques in the FOALP and FOAPI planners — FOALP, in particular, proved to be a capable planner, due in part to its ability to efficiently exploit universal reward structure. However, we remark that additive goal decomposition is only a heuristic approach and it is relatively easy to construct examples where it will fail; it is not clear to what extent enhancements and a deeper analysis of goal structure can mitigate these problems.

4. **Linear-value Approximation for FOMDPs:** We showed how to generalize linear-value approximation techniques from factored MDPs [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001] to the case of FOMDPs. This solution involved a number of novel contributions w.r.t. the introduction of the first-order linear programming paradigm, a generalization of the variable elimination algorithm to relation elimination, and the use of this algorithm in an efficient constraint generation approach to solving the first-order linear program. Additionally, we showed how the generation of orthogonal basis functions could be exploited in our solution algorithms. Together, all of these contributions made first-order generalizations of approximate linear programming (FOALP) and approximate policy iteration (FOAPI) possible. And in combination with additive goal decomposition as mentioned previously, FOALP managed to outperform state-of-the art stochastic planners on certain classes of problems due to its exploitation of relational and goal-oriented structure.

   We note that the generalization of approximate linear programming (ALP) to the first-order case of FOALP was only heuristic in that we could not directly represent the ALP objective in FOALP. It would be useful to revisit our assumptions in modeling the FOALP objective to determine if there are better approaches. While we did manage to

directly generalize approximate policy iteration (API) to the first-order case of FOAPI — thus obtaining a first-order generalization of the API error bounds — we note that FOAPI proved to be a difficult method to apply in practice. The main problem with FOAPI was that its policy representation could not be maintained in a compact form and the growth of the policy representation as more basis functions were added quickly outstripped the ability of our theorem prover to detect inconsistency. Thus, for FOAPI to be a viable approach in the future, we need to focus on compact ways to derive and represent the policy. Finally, for both FOALP and FOAPI, we remark that our basis function generation was highly heuristic and geared towards a specific set of probabilistic planning problems. More work is needed to identify general, automated methods of producing first-order basis functions.

5. **Representation and Solution of Factored FOMDPs:** We contributed the factored FOMDP formalism to permit FOMDPs to domain-independently represent factored structure such as additive rewards and factored actions that scale with the domain size. We provided a compact formalization of effect axioms and discussed a number of their properties that could be exploited in solution approaches.

   Beyond the representation, our investigation of solution methods was highly exploratory and we were only able to provide small examples and ad-hoc approaches for exploiting some of the structure that may occur in factored FOMDPs. For example, we identified a case where parameterized case structure may arise and we modified our case operators to handle such additional structure. We also contributed specialized symbolic dynamic programming and linear-value approximation techniques to solve certain factored FOMDPs. Even though our linear-value approximation approach was specific to a domain size, we introduced a framework that allowed the constraints to be evaluated without requiring domain grounding. These ideas built on the first-order probabilistic inference (FOPI) work of [Poole, 2003; de Salvo Braz *et al.*, 2005; de Salvo Braz *et al.*, 2006] where we also introduced the two novel elimination methods of *existential elimination* and *linear elimination* for performing variable elimination in this framework without grounding. Together, these ideas permitted the factored FOALP solution of the SYSADMIN problem in space and time that scaled sublinearly in the domain size — a result that is impossible to obtain for the corresponding grounded ALP approach. However this work is just the tip of the iceberg and leaves many important open questions such as "what classes of factored FOMDP structure can be solved efficiently?" We provide some guidance on

this problem in the next section covering directions for future work.

6. **Correspondence of Symbolic Dynamic Programming for FOMDPs and Dynamic Programming for MDPs:** We provided a proof of correspondence between symbolic dynamic programming (SDP) for FOMDPs and Dynamic Programming for MDPs. The key to this proof was showing that when an SDP solution to FOMDPs is grounded w.r.t. a domain closure assumption, the result is equivalent to the solution obtained by first grounding the FOMDP and then applying standard ground MDP solution techniques. We remark that this was an alternate proof approach than that given in [Boutilier *et al.*, 2001]. There the emphasis was on proving the correctness of the SDP algorithm at a purely logical level (including the case of infinite models). In our proof, we focused on proving correspondence between the first-order and well-known ground MDP solutions. Among other things, this allowed us to lift the results for approximation error bounds to the first-order case.

7.2 Future Directions

With respect to this thesis work, there are a number of open ends that are worth further exploration. Here 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. And for ideas, we need only look at the range of successful planners used in planning competitions. Perhaps one of the most useful approaches would be to use offline methods for solving FOMDPs to generate a domain-independent 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]. 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 fully evaluate all of the possible combinations of structural exploitation in FOMDPs. For example, we introduced both FOADDs and FOAADDs, but we only

used FOADDs in our experiments since preliminary experiments involving FOAADDs demonstrated that there was little additive or multiplicative structure to exploit in these problems. Furthermore, we discussed the APRICODD extension of SPUDD for approximate value iteration with ADDs in Chapter 3, but we did not consider similar extensions for approximate value iteration with first-order ADDs for FOMDPs. Given the success of APRICODD, this approach is quite appealing for first-order approximate value iteration; when the FOADD representing the value function becomes too large we can simply prune out nodes in the FOADD (as demonstrated in APRICODD for ADDs in Figure 3.6) in an effort to reduce the size of the value function while minimizing the approximation error.

3. We only skimmed the surface of research on factored FOMDPs. Perhaps the single greatest unanswered question for factored FOMDPs is how to identify structure that can be efficiently exploited by solution methods — and furthermore, how to automate these solutions. One approach to addressing this would analyze specific classes of problem structure, their efficient solution (if possible), and how these classes of problem structure could be combined while still permitting efficient solutions. In many ways, this is similar to the field of description logics in its nascency, when researchers sought to determine which combinations of logical constructors permitted efficient subsumption reasoning. However, the well-known result of Levesque and Brachman [1987] for description logics (showing that very minor changes in logical representation can lead to major changes in tractability) portends a similar negative result for factored FOMDPs due to the underlying connections at an abstract logical level. That is, very simple factored FOMDP structures that are efficiently solvable in isolation may interact such that their combination is no longer efficiently solvable. However, such discussion is only hypothetical and a clear formal analysis is needed to verify this. Nonetheless, in light of the results of Jaeger [2000] as discussed in the conclusion of Chapter 6, it does become clear that exact solutions to factored FOMDPs will not always be tractable and thus we discuss additional ideas for approximate solution approaches momentarily.

   One other interesting avenue for future research is on methods that extend first-order probabilistic inference (FOPI) ideas [Poole, 2003; de Salvo Braz *et al.*, 2005; de Salvo Braz *et al.*, 2006] to the relation elimination approach of Chapter 5 to permit a more general application of FOPI to linear-value approximation techniques for factored FOMDPs. At the current time, FOPI-based inference focuses on non-quantified relational structure

in the form of parameterized factors (i.e., *parfactors* in the FOPI lexicon), yet general factored FOMDPs clearly permit the use of quantifiers in the case statement representation that generalizes parfactors to full first-order logic. However, while the representational generalization is clear, the algorithmic generalization is much less clear since the lifted propositional ordered resolution used at the parfactor level in FOPI has much better computational properties than the first-order ordered resolution used at the case level in relation elimination (i.e., the propositional variant is guaranteed to terminate on any single elimination step whereas the first-order variant may not). But perhaps the use of specially restricted languages and more complex ordered resolution methods based on reduction orderings may resolve some of these difficulties; a fascinating thesis on this latter topic is given by Motik [2006].

4. One very fascinating idea and perhaps one of the most promising uses of FOMDPs and factored 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 w.r.t. the optimal policy in the non-abstracted version. If we lift such results to FOMDPs and factored FOMDPs, then this offers a very appealing paradigm for their use: we can approximate a general (factored) 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 shaped rewards [Ng *et al.*, 1999; Marthi, 2007] or seed values [Wiewiora, 2003] for value iteration in the non-abstracted MDP model.

In addition to the immediate open ends of our current research, 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, we began to uncover situations where we might obtain parameterized case structure in Chapter 6. We could continue this line of inquiry into parameterized value representations in the context of modeling continuous state properties, perhaps combined with discrete state properties in a *hybrid (FO)MDP*. This idea is intriguing in that it permits the specification of FOMDPs with the state represented in terms of continuous quantities where actions may range over continuous variables and rewards may scale continuously with state variables (or relations). It is quite easy to formulate some simple problems in this domain, such as moving quantities of water between tanks by opening and closing valves [Hauskrecht and Kveton, 2004; Guestrin *et al.*, 2004]. However, solving such a problem domain-independently and efficiently will likely require a significant extension of current methods.

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 w.r.t. 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; Andre and Russell, 2002] to the case of solving FOMDPs w.r.t. 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; Soutchanski, 2001; Ferrein *et al.*, 2003] have provided an initial investigation into these ideas.

Altogether, it should be clear that this thesis only represents the tip of the iceberg for first-order decision-theoretic planning in structured relational environments. And the above suggestions are but a few of the many possible avenues in this fecund field of research.

7.3 Concluding Remarks

For a few years immediately succeeding the publication of first-order MDPs and their symbolic dynamic programming solution [Boutilier *et al.*, 2001], this 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 applications, this thesis 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, lifted solutions *can* work in practice. And at the present point in time, not only can they work, but they can scale well beyond that of grounded approaches in many notable cases.

Despite these successes, many researchers will continue to avoid lifted first-order methods and resort to grounded methods for the very practical reason that grounded methods are both easier to understand and easier to implement. But there is a great potential payoff to be gained by understanding and working at the first-order level when approaching decision-theoretic planning problems. And there is a world of relational and symbolic structure waiting to be exploited. This thesis represents just a few points in that space of ideas and there are likely a plethora of breakthroughs in first-order decision-theoretic planning patiently awaiting discovery. Our hope is that this thesis lays out the foundations for further exploration of this space and helps move this nascent field further along the path of practical impact.

Appendix A

Proof of Correctness of Symbolic Dynamic Programming

In this appendix, we provide a proof of correspondence between symbolic dynamic programming (SDP) for the FOMDP model of Chapter 4 and dynamic programming (DP) for the MDP model of Chapter 2 under finite domain assumptions.

A.1 General Proof Approach

The key to this proof is showing that when a SDP solution to FOMDPs is grounded w.r.t. a finite domain via domain closure axioms[1], the result is equivalent to the solution obtained by first grounding the FOMDP and then applying standard ground MDP solution techniques (see Figure A.1 for a visual representation of this proof).

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. However, they do not provide an explicit correspondence between FOMDPs formalized with the deterministic situation calculus and MDPs as formalized in Chapter 2 with explicit stochastic actions. While the correspondence is not difficult to show, it is nonetheless useful to make this explicit. Thus, we provide a direct correspondence between FOMDPs and MDPs in this Appendix and provide an alternate proof of correctness of first-order value iteration based on this correspondence

[1]Throughout this appendix, when we say ground, we mean to restrict the interpretations to a finite domain via domain closure axioms. Domain closure will be formally defined in the next section.

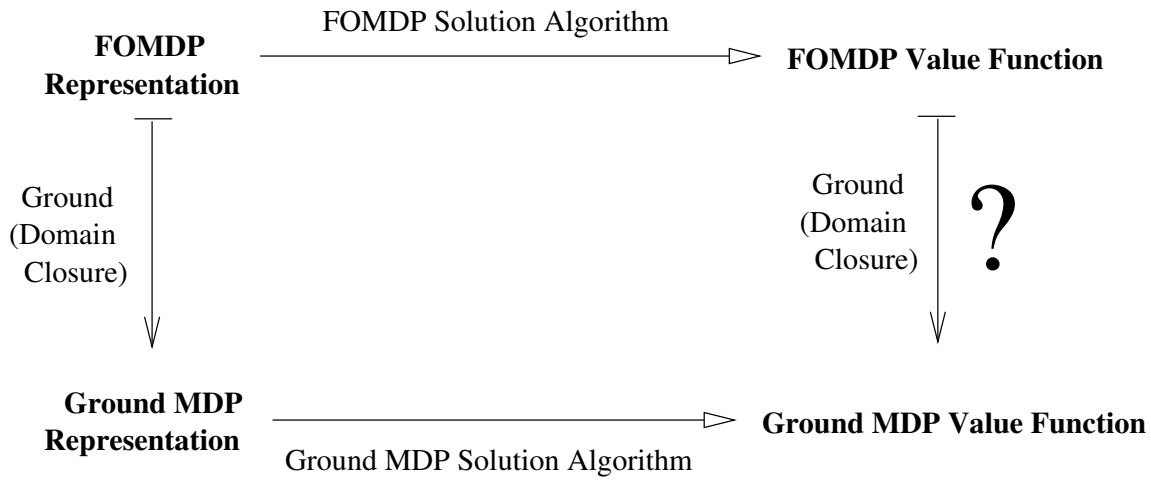


Figure A.1: Proving correspondence between FOMDPs and MDPs.

A.2 Correspondence of Case and Ground Representations

For simplicity, we will assume an unsorted first-order logic with equality. While we have previously assumed a sorted representation, we will assume that sort information has been compiled into an unsorted logical form where $\forall Sort : c\ \phi(c)$ has been rewritten as $\forall c.\ Sort(c) \supset \phi(c)$ and likewise $\exists Sort : c\ \phi(c)$ has been rewritten as $\exists c.\ Sort(c) \wedge \phi(c)$. In the following presentation we draw on the logical notation and semantics for unsorted first-order logic given in Brachman and Levesque [2004]. It is particularly important to note the following two restrictions:

- *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 "$=$"$\in P_2$.
- *Function Symbols:* We assume a set of function symbols f_j of each arity $0 \leq j \leq n$ for some finite maximum n.

We recapitulate the case notation introduced in Chapter 4 and used in the first-order MDPs in this thesis. We use the notation:

$$t \ = \ case[\phi_1, t_1; \cdots ; \phi_n, t_n] \tag{A.1}$$

as an abbreviation for the logical formula:

$$\bigvee_{i \leq n} \{\phi_i \wedge t = t_i\} \tag{A.2}$$

Here, the $\phi_j(s)$ are *state formulae* (whose situation term does not use do) and we assume in these proofs that the t_j are real-valued constants such that $\forall i.\, t_i \in \mathbb{R}$. We will always assume an implicit case element $\langle \neg\phi_1 \wedge \ldots \wedge \neg\phi_n, -\infty \rangle$ to ensure that the case statement exhaustively assigns a value to all possible ground states and we assume that all case partitions are mutually exclusive[2] unless otherwise noted. We assume that the free variable t in the logical formula for which $t = case[\phi_1, t_1; \cdots; \phi_n, t_n]$ is an abbreviation does not appear in any of the ϕ_j.

A finite interpretation is a pair $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$ where $\mathcal{D}$ is a finite, nonempty set of domain elements $\{c_1, \ldots, c_k\}$ and $\mathcal{I}$ is a mapping from all predicate names in P_i for $0 \leq i \leq m$ into a subset of $\mathcal{D}^i$ and from function names in f_j for $0 \leq j \leq n$ into a map of $\mathcal{D}^j \rightarrow \mathcal{D}$. We represent variable bindings for free variables of a formula as a variable assignment μ, which is a set of substitutions $\{v/c\}$ of $c \in \mathcal{D}$ for variable v. We use $\phi\mu$ to represent the formula resulting from making the substitutions of μ for free variables in ϕ. We say an interpretation $\Im$ and a variable binding μ satisfy a formula ϕ, written $\Im, \mu \models \phi$, if $\phi\mu$ can be recursively evaluated to be true under the model-theoretic interpretation of first-order logic syntax given $\Im$ [Brachman and Levesque, 2004].

A *domain closure assumption* is an axiom that restricts the universe of objects to those explicitly in a finite domain $\mathcal{D}$. Throughout this appendix, we will refer to a case statement under a domain closure assumption, written generically as $case^{\mathcal{D}}$. We will assume that the following domain closure axiom is *implicit* in the background theory:

$$\forall x.\, x = c_1 \vee \ldots \vee x = c_k \tag{A.3}$$

A.3 Correspondence of Representations and Operations

Although we typically use a ground representation expressed as $case^{\mathcal{D}}$, we note that there is a simple transformation between $case^{\mathcal{D}}$ and the more familiar ground representation of propositional factors used in the ground factored MDP representation from Chapter 3. Here we present a simple method for converting $case^{\mathcal{D}}$ to a propositional factor where $case^{\mathcal{D}}$ that we call *grounding*:

Definition A.3.1 (Grounding)**.** *To ground $case^{\mathcal{D}}$ to a propositional factor C as demonstrated in Figure A.2, we perform the following steps where we assume the partitions of $case^{\mathcal{D}}$ are mutually exclusive and exhaustive:*

[2] Mutual exclusivity can be easily enforced with the unary casemax operator from Chapter 4.

$case^D =$

$\exists x.\ A(x)$	: 10
$\neg\exists x.\ A(x)$:	5

$\longmapsto$ Prop. Factor C:

$A(c_1)$	$A(c_2)$	
$\bot$	$\bot$	5
$\bot$	$\top$	10
$\top$	$\bot$	10
$\top$	$\top$	10

Figure A.2: Given the tabular representation of a case statement $case^D$, its grounded representation as a propositional factor for $\mathcal{D} = \{c_1, c_2\}$ is given on the RHS. If our language had included function symbols, we would have included extra columns in the factor C representing all truth-value of all possible function equalities.

1. *Expand all quantifiers into finite conjunctions ($\forall$) or disjunctions ($\exists$) over all elements of $\mathcal{D}$. It is easy to see that the resulting case statement will only consist of connectives over ground atoms and terms (with the exception of variable t).*

2. *Build a tabular representation of a propositional factor C that enumerates all truth assignments of all ground atoms w.r.t. $\mathcal{D}$ (the ground atoms for equality suffice to represent all function valuations) referenced by the $case^{\mathcal{D}}$ formula in the* rows *of the table.*

3. *Each row r of C represents a set of interpretations w.r.t. domain $\mathcal{D}$ that are consistent with the truth assignments to ground atoms in that row. Each row r of C is also assigned a value t_r. Because C forms an exhaustive truth table over relevant ground atoms, the rows of C disjointly and exhaustively partition the set of all interpretations for a fixed domain $\mathcal{D}$ (no interpretation could satisfy the assignments from two different rows). We denote the set of all interpretations for row r of C to be ρ_r. We assume there are no free variables of $case^{\mathcal{D}}$ other than t, which appears in the formal logical representation of $t = case^{\mathcal{D}}$. If $\langle \phi_j, t_j \rangle$ is a partition of $case^{\mathcal{D}}$ and $\forall \Im \in \rho_r . \Im \models \phi_j$ then we assign value $t_r = t_j$ for row r.*

Next we define a binary relation $\longmapsto$ that allows us to establish a correspondence between $case^{\mathcal{D}}$ and a ground propositional factor C:

Definition A.3.2 ($\longmapsto$)**.** *Let all symbols be defined as in Definition A.3.1. Define the ground atoms* relevant *to $case^{\mathcal{D}}$ as all of the ground atom truth assignments that could be potentially required during the model-theoretic evaluation of the logical representation of $case^{\mathcal{D}}$. Assume that the rows of C exhaustively enumerate all truth assignments to relevant ground atoms of*

$case^{\mathcal{D}}$. Then the correspondence $case^{\mathcal{D}} \longmapsto C$ holds iff for all rows r of C, there exists exactly one partition $\langle \phi_j, t_j \rangle$ of $case^{\mathcal{D}}$ s.t. $\forall \Im \in \rho_r.\Im, \{t/t_r\} \models \phi_j \wedge t = t_j$.

Lemma A.3.3. *If we ground $case^{\mathcal{D}}$ to obtain C using Definition A.3.1, then $case^{\mathcal{D}} \longmapsto C$.*

Proof. Let all symbols be defined as in Definition A.3.1. By the definition of the grounding procedure, the rows of C exhaustively partition all possible relevant ground atoms of $case^{\mathcal{D}}$ (an examination of the rules for model-theoretic evaluation reveal that we need only know truth assignments to the ground atoms for the relations and function symbols w.r.t. $\mathcal{D}$ in $case^{\mathcal{D}}$, which the grounding procedure indeed provides).

Now, choose any particular row r of C. We know that every row r (viewed as an interpretation) is a model of exactly one ϕ_j. This follows from the mutually exclusive and exhaustive nature of partitions $\langle \phi_j, t_j \rangle$ in $case^{\mathcal{D}}$ and the fact that row r makes truth assignments to all relevant ground atoms of $case^{\mathcal{D}}$. Since all $\Im \in \rho_r$ are just augmentations of the truth assignments to r, we then know $\forall \Im \in \rho_r.\Im \models \phi_j$. Furthermore, by the definition of the grounding procedure, we can infer $t_r = t_j$. From this, we can then infer from Definition A.3.2 that $case^{\mathcal{D}} \longmapsto C$ must hold. □

We provide a simple example of a case statement under domain closure and its grounded representation as a propositional factor in Figure A.2. Note that the top partition of the $case^{\mathcal{D}}$ statement is modeled by the three bottom rows of the propositional factor C and that each row of C corresponds to exactly one partition of $case^{\mathcal{D}}$.

For a binary operation $op \in \{\oplus, \otimes, \ominus, \max\}$, the application of op on tabular representations of propositional factors can be expressed in exactly the same form as a binary operation on the case representation: a cross-product operation op of all rows with inconsistency removal (and optional simplification).

We note that performing operations on case statements under domain closure is equivalent to transforming the case statements to propositional tables and performing the same operations on propositional tables. To show that this is correct, we prove the following theorem:

Theorem A.3.4. *Using Definition A.3.1 to ground $case_1^{\mathcal{D}}$ to C_1 and $case_2^{\mathcal{D}}$ to C_2, let $case_R^{\mathcal{D}} = (case_1^{\mathcal{D}}\ op\ case_2^{\mathcal{D}})$ and let $C_R = C_1\ op\ C_2$. Then $case_R^{\mathcal{D}} \longmapsto C_R$.*

Proof. From Lemma , we know that $case_1^{\mathcal{D}} \longmapsto C_1$ and $case_2^{\mathcal{D}} \longmapsto C_2$. Our goal is to show that $case_R^{\mathcal{D}} \longmapsto C_R$ by Definition A.3.2.

By the construction of C_R as the result of a binary operation on ground propositional factors (previously discussed), we know that it exhaustively enumerates truth assignments to all ground

atoms of C_1 and C_2, which in turn include all relevant ground atoms of $case_1^{\mathcal{D}}$ and $case_2^{\mathcal{D}}$. As the only possible relevant ground atoms of $case_R^{\mathcal{D}}$ are those corresponding to the grounding of function symbols and predicates of $case_1^{\mathcal{D}}$ and $case_2^{\mathcal{D}}$ (recall that $case_R^{\mathcal{D}}$ simply contains a conjunction of formulae from $case_1^{\mathcal{D}}$ and $case_2^{\mathcal{D}}$), we can infer that C_R exhaustively enumerates truth assignments to all relevant ground atoms of $case_R^{\mathcal{D}}$.

By the construction of $case_R^{\mathcal{D}}$, we know that all partitions of $case_R^{\mathcal{D}}$ are exhaustive and mutually exclusive. This follows from the fact that each of $case_1^{\mathcal{D}}$ and $case_2^{\mathcal{D}}$ were exhaustive and mutually exclusive and thus the resulting cross-product of case partitions in $case_R^{\mathcal{D}}$ likewise retains this property.

Now, choose a row r of C_R. By construction, we know that row r was formed from a row i in C_1 and a row j in C_2. Let ρ_i^1 and ρ_j^2 be the set of respective intepretations corresponding to a row i of C_1 and row j of C_2. We note the following: (1) $\forall \Im_1 \in \rho_i^1 . \Im_1 \models \phi_1$ and (2) $\forall \Im_2 \in \rho_j^2 . \Im_2 \models \phi_2$.

By definition of the case operations there will be a case element $\langle \phi_r, t_r \rangle$ in $case_R^{\mathcal{D}}$ where $\phi_r \equiv \phi_1 \wedge \phi_2$ and $t_r = t_1 \ op \ t_2$ if $\phi_1 \wedge \phi_2 \nvdash \bot$. To complete the proof, we first need to show that $\forall \Im_R \in \rho_i^1 \cap \rho_j^2 . \Im_R, \{t/(t_1 \ op \ t_2)\} \models \phi_r \wedge t = t_r$ for $\langle \phi_r, t_r \rangle$ in $case_R^{\mathcal{D}}$ (i.e., we need to show that all interpretations of row r of C_R model the correct value w.r.t. $case_R^{\mathcal{D}}$). Replacing ϕ_r with the equivalent $\phi_1 \wedge \phi_2$, and recursively decomposing the model-theoretic interpretation of $\wedge$, this requires us to show (3) $\forall \Im_R \in \rho_i^1 \cap \rho_j^2 . \Im_R, \{t/(t_1 \ op \ t_2)\} \models \phi_1$, (4) $\forall \Im_R \in \rho_i^1 \cap \rho_j^2 . \Im_R, \{t/(t_1 \ op \ t_2)\} \models \phi_2$, and (5) $\forall \Im_R \in \rho_i^1 \cap \rho_j^2 . \Im_R, \{t/(t_1 \ op \ t_2)\} \models t = t_r$. (3) and (4) follow respectively from (1) and (2); (5) follows from the definition of $t_r = t_1 \ op \ t_2$. Finally we need to show $\phi_1 \wedge \phi_2 \nvdash \bot$, but this follows easily since (3) and (4) imply $\forall \Im \in \rho_i^1 \cap \rho_j^2 . \Im \models \phi_1 \wedge \phi_2$ and we know $\rho_i^1 \cap \rho_j^2$ is non-empty because it contains the model corresponding to row r of C_R. □

A.4 Correspondence of a FOMDP and an MDP

To begin the correspondence proofs given in Figure A.1, we must first define the FOMDP and its grounded MDP variant.

To define a FOMDP, we slightly modify notation from Chapter 4 such that we use the $case$ specification both for a generic FOMDP and a specific instance to avoid confusion with the ground MDP notation. A FOMDP is described by a reward case statement $rCase(s)$, case statements representing Nature's choice distribution over deterministic action outcomes $pCase(n_{i,j}(\vec{y}), A_i(\vec{y}), s)$ where for each stochastic action term $A_i(\vec{y})$ for $1 \leq i \leq p$ there is

a corresponding set of Nature's choice deterministic action outcomes $n_{i,j}(\vec{y})$ for $1 \leq i \leq p$, $1 \leq j \leq q$, and a set of successor state axioms (SSAs) for each fluent and Nature's choice action. Often, when we are referring to a specific action, we will drop the index i, e.g., $pCase(n_j(\vec{x}), A(\vec{x}))$. Somewhat more importantly, we note the following notational convention:

- Since state properties can be recovered from situation terms due to the Markovian assumptions of an MDP, we drop the situation term s from all FOMDP case statements from here out. The use of the stage-to-go index t will allow us to track the state that a case statement is referring to.

To define an MDP and derive it from a FOMDP grounded w.r.t. domain $\mathcal{D}$, we use the factored propositional MDP notation from Chapter 3. We define the set of states $\mathcal{S}^{\mathcal{D}}$ in terms of all possible truth assignments to a vector of binary state variables $\vec{x}$ consisting of the following variables:

- For all i ($0 \leq i \leq m$), for all predicate names $P \in P_i$, and for all $\vec{c} \in \mathcal{D}^i$, there is a binary variable representing whether the atom $P(\vec{c})$ is true. Again, due to the inclusion of the equality predicate, this suffices to handle all function valuations.

In essence, under a domain assumption $\mathcal{D}$, the state variables $\vec{x}$ are capable of representing all possible mappings $\mathcal{I}$ for a finite interpretation $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$.

With these definitions, we define the MDP reward as a factor $R(\vec{x})$ for $\vec{x} \in \mathcal{S}^{\mathcal{D}}$, and for all i, j ($1 \leq i \leq p$, $1 \leq j \leq q$), and we define the MDP value function as $V^t(\vec{x})$ for $t \geq 0$ and $\vec{x} \in \mathcal{S}^{\mathcal{D}}$.

The set $\mathcal{A}^{\mathcal{D}}$ contains all grounded versions of stochastic actions $A_i(\vec{y})$ ($1 \leq i \leq p$) for a FOMDP grounded w.r.t. $\mathcal{A}^{\mathcal{D}}$. Likewise, the set $\mathcal{N}^{\mathcal{D}}$ contains all grounded Nature's choice deterministic actions $n_{i,j}(\vec{y})$ ($1 \leq i \leq p$,$1 \leq j \leq q$). We define $\mathcal{A}^{\mathcal{D}}$ and $\mathcal{N}^{\mathcal{D}}$ in the following way where we assume α_i is the arity of action A_i:

- $\mathcal{A}^{\mathcal{D}}$: For all i ($1 \leq i \leq p$) and for all $\vec{c} \in \mathcal{D}^{\alpha_i}$, there is a stochastic action symbol for the term $A_i(\vec{c})$.

- $\mathcal{N}^{\mathcal{D}}$: For all i ($1 \leq i \leq p$), for all j ($1 \leq j \leq q$), and for all $\vec{c} \in \mathcal{D}^{\alpha_i}$, there is a Nature's choice deterministic action symbol for the term $n_{i,j}(\vec{c})$.

We note that the FOMDP and MDP *share* symbols at a syntactic level, namely those in $\mathcal{A}^{\mathcal{D}}$ and $\mathcal{N}^{\mathcal{D}}$, which are function terms in the FOMDP case and enumerated symbols in the ground

MDP case. However, the "meaning" of the symbols can be disambiguated by context in the following presentation unless otherwise noted.

Next we show how to build the transition functions $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$ for all actions $A_i(\vec{c}) \in \mathcal{A}^{\mathcal{D}}$ in the MDP given $P(n_{i,j}(\vec{c})|A_i(\vec{c}), \vec{x})$ in the FOMDP where $n_{i,j}(\vec{c}) \in \mathcal{N}^{\mathcal{D}}$, $A_i(\vec{c}) \in A^{\mathcal{D}}$, and $\vec{x} \in \mathcal{S}^{\mathcal{D}}$.

To do this, we need to show how a next-state $\vec{x}'$ can be determined given a current state $\vec{x}$ and Nature's choice deterministic action $n_{i,j}(\vec{c})$. Recalling the definition of successor-state axioms (SSAs) from Chapter 4, Section 4.2.2, this is easy. For each relational fluent $F(\vec{y}, s)$[3], we have an SSA of the form $F(\vec{y}, do(a, s)) \equiv \Phi_F(\vec{y}, a, s)$. Given a ground action $a = n_{i,j}(\vec{c})$ and the corresponding set of SSAs, we can determine the truth-value of all atoms represented in $\vec{x}'$ directly from these SSAs; for each atom in $\vec{x}'$, we find its corresponding fluent — call it $F(\vec{d}, s)$ — then we evaluate $\Phi_F(\vec{d}, n_{i,j}(\vec{c}), s)$ on the pre-action state $\vec{x}$ to determine whether that atom should be set to true or false. We denote the $\vec{x}'$ that results from performing action $n_{i,j}(\vec{c})$ in $\vec{x}$ with the notation $\vec{x}' = \mathit{Progress}(\vec{x}, n_{i,j}(\vec{c}))$.

Given stochastic action $A_i(\vec{c})$ we want to build $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$ by summing the probability of reaching $\vec{x}'$ from $\vec{x}$ given one of Nature's choice deterministic outcomes $n_{i,j}(\vec{c})$ of $A_i(\vec{c})$ for each $\vec{x}'$ and $\vec{x}$. We implement this directly with the following calculation where $\mathbb{I}[\cdot]$ is an indicator function taking the value 1 when its argument is true and 0 otherwise and $pCase^{\mathcal{D}}(n_j(\vec{c}), A(\vec{c})) \longmapsto P(n_{i,j}(\vec{c})|A_i(\vec{c}), \vec{x})$:

$$P(\vec{x}'|\vec{x}, A_i(\vec{c})) = \sum_{j=1}^{q} \mathbb{I}[\vec{x}' = \mathit{Progress}(\vec{x}, n_{i,j}(\vec{c}))] \cdot P(n_{i,j}(\vec{c})|A_i(\vec{c}), \vec{x}) \tag{A.4}$$

Given the constructions above, we can now obtain an MDP instance from a particular FOMDP under domain instantiation $\mathcal{D}$:

Definition A.4.1 (Grounding a FOMDP w.r.t. $\mathcal{D}$ to obtain an MDP)**.** *A FOMDP and its grounding w.r.t. a domain $\mathcal{D}$ to obtain an MDP are given by the following ground correspondences where $\vec{x} \in \mathcal{S}^{\mathcal{D}}$ (the function may actually only be over a subset of variables in $\vec{x}$), $n_{i,j}(\vec{c}, \vec{x}) \in \mathcal{N}^{\mathcal{D}}$, and $A_i(\vec{c}) \in A^{\mathcal{D}}$ (as defined above):*

- *Obtain $R(\vec{x})$ from $rCase^{\mathcal{D}}$ by grounding according to Definition A.3.1*
- *Obtain $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$ from $pCase^{\mathcal{D}}(n_j(\vec{c}), A(\vec{c}))$ via Equation A.4.*
- *Use the discount γ from the FOMDP for the MDP.*

[3]We ignore functional fluents in this presentation as in Chapter 4, but they could be incorporated if needed.

Note that given a FOMDP t-stage-to-go value function $vCase^{t,\mathcal{D}}$, we can likewise obtain a ground representation $V^t(\vec{x})$ using Definition A.3.1.

Now we return to the overall goal of our proof as illustrated in Figure A.1. Definition A.4.1 gives us the correspondence between the FOMDP and its ground MDP representation illustrated as the vertical $\longmapsto$ on the LHS of Figure A.1. But it also gives us a means for grounding a FOMDP value function $vCase^{t,\mathcal{D}}$ to obtain $V^t(\vec{x})$ for any $t \geq 0$ so that we can verify the vertical $\longmapsto$ on the RHS of Figure A.1. Consequently, our task in the remaining sections is to show that symbolic dynamic programming and dynamic programming algorithms (respectively, the top and bottom horizontal $\longmapsto$ in Figure A.1) preserve the correspondence $vCase^{t,\mathcal{D}} \longmapsto V^t(\vec{x})$ of the respective representations for all $t \geq 0$.

A.5 Correspondence of FODTR and DTR

Given the correspondence between a FOMDP and an MDP w.r.t. domain $\mathcal{D}$ from Definition A.4.1, we now seek to show a correspondence betwen first-order decision-theoretic regression (FODTR) and decision-theoretic regression (DTR) for a specific action instantiation $A(\vec{x})$.

To recap, in Chapter 3, we introduced DTR as a crucial step in the dynamic programming solution of MDPs that yields the Q-function:

$$\begin{aligned} Q^{t+1}(\vec{x}, A(\vec{c})) &= DTR[V^t(\vec{x}), A(\vec{c})] \\ &= R(\vec{x}) + \gamma \left[\sum_{\vec{x}'} P(\vec{x}' | \vec{x}, A_i(\vec{c})) V^t(\vec{x}') \right] \end{aligned} \tag{A.5}$$

Here we have substituted appropriate notation from Definition A.4.1 on the RHS of this equation.

In Chapter 4, we introduced FODTR as a crucial step in the dynamic programming solution of FOMDPs that yields the first-order Q-function. We include it here in its original form with situation terms in order to ensure that $Regr$ is well-defined:

$$\begin{aligned} &qCase^{t+1}(s, A(\vec{y})) = FODTR[vCase^t(s), A(\vec{y})] \\ &= rCase \oplus \text{casemax} \exists y. \gamma \left[\bigoplus_{j=1}^{q} \{ pCase(n_{i,j}(\vec{c}), A_i(\vec{y}), s) \otimes Regr(vCase^t(do(n_{i,j}(\vec{y}), s))) \} \right] \end{aligned} \tag{A.6}$$

For this proof, we are only interested in showing correspondence for a specific action parameter substitution $\{\vec{y}/\vec{c}\}$ so we can use the slightly simplified form of FODTR:

$$qCase^{t+1}(s, A(\vec{c})) = FODTR[vCase^t(s), A(\vec{c})] \tag{A.7}$$
$$= rCase \oplus \gamma \left[\bigoplus_{j=1}^{q} \{ pCase(n_{i,j}(\vec{c}), A_i(\vec{c}), s) \otimes Regr(vCase^t(do(n_{i,j}(\vec{c}), s))) \} \right]$$

Note that in the following, we treat case statements as being state-oriented and thus drop situation terms for the most part. However, we must reintroduce situation terms to perform $FODTR$, but we assume they are stripped off once the result has been computed.

To prove correspondence of FODTR and DTR, we need to prove the following theorem:

Theorem A.5.1. *Given $rCase^{\mathcal{D}}$, $pCase^{\mathcal{D}}(n_j(\vec{c}), A(\vec{c}))$, SSAs, and γ from a FOMDP, obtain $R(\vec{x})$, $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$, and γ for an MDP w.r.t. domain $\mathcal{D}$ from Definition A.4.1. Let $qCase^{t+1,\mathcal{D}}(A(\vec{c})) = FODTR[vCase^t, A(\vec{c})]$ as given in Equation A.7 for some $vCase^t$. Obtain $V^t(\vec{x})$ from $vCase^t$ using Definition A.3.1. Let $Q^{t+1}(\vec{x}, A(\vec{c})) = DTR[V^t(\vec{x}), A(\vec{c})]$ be obtained by first grounding the FOMDP w.r.t. $\mathcal{D}$ as defined above and applying the ground computation as given in Equation A.5. Then $qCase^{t+1,\mathcal{D}}(A(\vec{c})) \longmapsto Q^{t+1}(\vec{x}, A(\vec{c}))$.*

Proof. First we establish the one-to-one correspondence of the case statements in Equation A.7 and the propositional factors in Equation A.5 as assumed in Definition A.4.1. Specifically, it is immediately obvious that $rCase^{\mathcal{D}} \longmapsto R(\vec{x})$ and $vCase^{t,\mathcal{D}} \longmapsto V^t(\vec{x})$ by Definition A.3.1 and Lemma A.3.

Now we prove correspondence of the expectation portions of Equation A.7 and Equation A.5 (i.e., the content in the square braces $[\cdot]$ in both equations). To make this clear, let us temporarily ignore the reward and the discount factor γ and define two new equations where we have substituted the definition of $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$ from Equation A.4 in the first equation:

$$Q^{t+1}_{-R,\gamma}(\vec{x}, A(\vec{c})) = \left[\sum_{\vec{x}'} \sum_{j=1}^{q} \left\{ \mathbb{I}[\vec{x}' = Progress(\vec{x}, n_{i,j}(\vec{c}))] \cdot P(n_{i,j}(\vec{c})|A_i(\vec{c}), \vec{x}) V^t(\vec{x}') \right\} \right] \tag{A.8}$$

$$qCase^{t+1}_{-R,\gamma}(s, A(\vec{c})) = \left[\bigoplus_{j=1}^{q} \left\{ pCase(n_{i,j}(\vec{c}), A_i(\vec{c}), s) \otimes Regr(vCase^t(do(n_{i,j}(\vec{c}), s))) \right\} \right] \tag{A.9}$$

Our goal is to show $qCase^{t+1}_{-R,\gamma}(A(\vec{c})) \longmapsto Q^{t+1}_{-R,\gamma}(\vec{x}, A(\vec{c}))$. Specifically, let ρ_r be the set of

respective intepretations corresponding to some row r of $Q^{t+1}_{-R,\gamma}(\vec{x}, A(\vec{c}))$ having state variable assignment $\vec{x}_r$ and taking value v_r. And let $\langle \phi, v_\phi \rangle$ be some partition in $qCase^{t+1}_{-R,\gamma}(A(\vec{c}))$. If $\forall \Im \in \rho_r . \Im \models \phi$ then our goal is to show that $v_r = v_\phi$.

We know by the definition of FODTR that for all Nature's choice outcomes $n_{i,j}(\vec{c})$ ($j = 1 \ldots q$) of $A_i(\vec{c})$ that $\phi \equiv Regr[\phi'_j(do(n_{i,j}(\vec{c}), s))]$ for some value partition $\langle \phi'_j, t_j \rangle$ of $vCase^t$. Thus, given ϕ, we can directly express $v_\phi = \sum_{j=1}^{q} t_j \cdot pCase(n_{i,j}(\vec{c}), A_i(\vec{c}))$.

Similarly for DTR, given $\vec{x}_r$ and $n_{i,j}(\vec{c}))$, let $t_{\vec{x}_r, n_{i,j}(\vec{c})}$ denote the value of $V^t(\vec{x}'_r)$ for the unique $\vec{x}'_r$ satisfying $\mathbb{I}[\vec{x}'_r = Progress(\vec{x}_r, n_{i,j}(\vec{c}))] = 1$. Then, we can directly express $v_r = \sum_{j=1}^{q} t_{\vec{x}_r, n_{i,j}(\vec{c})} \cdot P(n_{i,j}(\vec{c}) | A_i(\vec{c}), \vec{x}_r)$.

To prove $v_r = v_\phi$, we need to show (1) $pCase(n_{i,j}(\vec{c}), A_i(\vec{c})) \longmapsto P(n_{i,j}(\vec{c}) | A_i(\vec{c}), \vec{x}_r)$ (already shown at beginning of proof) and (2) $t_j = t_{\vec{x}_r, n_{i,j}(\vec{c})}$. To prove (2), we need to show that if ρ_r is a set of interpretations consistent with $\vec{x}_r$ and $\rho_r \models \phi$, then a set of interpretations ρ'_r consistent with $\vec{x}'_r$ (obtained by progressing $\vec{x}_r$ through $n_{i,j}(\vec{c})$) must satisfy $\forall \Im \in \rho'_r . \Im \models \phi'_j$.

We prove (2) by contradiction. Assume ρ_r is a set of interpretations consistent with $\vec{x}_r$ and $\rho_r \models \phi$. Also assume there exists an $\Im \in \rho'_r$ s.t. $\Im$ is consistent with $\vec{x}'_r$ and $\Im \not\models \phi'_j$. Briefly reintroducing situation terms, we derive $\phi(s)$ via the SSAs using $\phi(s) \equiv Regr[\phi'_j(do(n_{i,j}(\vec{c}), s))]$ and then drop the situation term s from $\phi(s)$ to obtain ϕ. We know that the models ρ_r of ϕ must be consistent with $\vec{x}_r$. Then the ground atom truth assignments of $\vec{x}_r$ must be consistent with the predecessor-state conditions ϕ that make ϕ'_j true. The interpretations in ρ' that are consistent with $\vec{x}'_r$ (by definition) must also satisfy $\forall \Im \in \rho'_r . \Im \models \phi'_j$ — this is the crucial step and follows from the fact that the truth assignments in $\vec{x}_r$ had to satisfy all of the preconditions that led to ϕ'_j and thus $\vec{x}'_r = Progress(\vec{x}_r, n_{i,j}(\vec{c}))$ (which simply uses a ground version of the SSAs) can only produce truth assignments in $\vec{x}'_r$ that are consistent with all interpretations of ϕ'_j. But this result contradicts our assumption and thus we can conclude that for all $\Im \in \rho'_r$ s.t. $\Im$ is consistent with $\vec{x}'_r$ that $\Im \models \phi'_j$

This proves correspondence of the expectations $qCase^{t+1}_{-R,\gamma}(A(\vec{c})) \longmapsto Q^{t+1}_{-R,\gamma}(\vec{x}, A(\vec{c}))$. From this result and the two equations

$$Q^{t+1}(\vec{x}, A(\vec{c})) = R(\vec{x}) + \gamma \cdot Q^{t+1}_{-R,\gamma}(\vec{x}, A(\vec{c}))$$
$$qCase^{t+1}(A(\vec{c})) = rCase \oplus \gamma \cdot qCase^{t+1}_{-R,\gamma}(A(\vec{c}))$$

we can easily infer that $qCase^{t+1,\mathcal{D}}(A(\vec{c})) \longmapsto Q^{t+1}(\vec{x}, A(\vec{c}))$ follows directly from the results of our case operation correspondence Theorem A.3.4. □

A.6 Correspondence of Symbolic and Ground Maximization

Having proved the correspondence $qCase^{t+1,\mathcal{D}}(A(\vec{c})) \longmapsto Q^{t+1}(\vec{x}, A(\vec{c}))$ for a ground action $A(\vec{c})$, we now proceed to tackle the maximization required to make the full Bellman backup and obtain the correspondence pair $vCase^{t+1,\mathcal{D}} \longmapsto V^{t+1}(\vec{x})$.

As previously defined, we assume the FOMDP has p action templates $A_1(\vec{y}), \ldots, A_p(\vec{y})$. To compute the Bellman backup in the ground case, we must take the maximum of $Q(\vec{x}, a)$ over all ground actions $a \in \mathcal{A}^{\mathcal{D}}$, where the elements of $\mathcal{A}^{\mathcal{D}}$ can be partitioned according to which of the p action templates they were derived from. Since maximization is commutative and associative, we first tackle the maximization for all ground action instantations of a single stochastic action template $A(\vec{y}) = A_i(\vec{y})$ $(1 \leq i \leq p)$. Recalling that α_i is the arity of A_i, we enumerate all possible ground instantiations $\vec{y} = \vec{c}$ as $\vec{c} \in \mathcal{D}^{\alpha_i}$ where $\mathcal{D}^{\alpha_i} = \{\vec{c}_1, \ldots, \vec{c}_{|\vec{y}|}\}$.

Theorem A.6.1. *Define the FOMDP, MDP w.r.t. domain $\mathcal{D}$, corresponding value functions $vCase^{t,\mathcal{D}}$ and $V^t(\vec{x})$, and Q-functions $qCase^{t+1,\mathcal{D}}(A(\vec{c}))$ and $Q^{t+1}(\vec{x}, A(\vec{c}))$ as in Theorem A.5.1. Let $qCase^{t+1,\mathcal{D}}(A(\vec{y}))$ be the result of computing Equation A.7 for $vCase^{t,\mathcal{D}}$ and action template $A(\vec{y})$ w.r.t. a given FOMDP. Then the following correspondence holds:*

$$\textit{casemax}\ \exists \vec{y}\ qCase^{t+1,\mathcal{D}}(A(\vec{y})) \quad \longmapsto \quad \max_{a \in \{A(\vec{c}_1), \ldots, A(\vec{c}_{|A(\vec{y})|})\}} Q^{t+1}(\vec{x}, a)$$

Proof. We can make the following equivalence transformations to rewrite the LHS until it directly corresponds to the RHS (we justify the steps below):

$$\text{casemax}\ \exists \vec{y}. \begin{array}{|c|}\hline \phi_1(\vec{y}) : t_1 \\ \hline : \quad : : \\ \hline \phi_n(\vec{y}) : t_n \\ \hline \end{array} = \text{casemax} \begin{array}{|c|}\hline \exists \vec{y}.\ \phi_1(\vec{y}) : t_1 \\ \hline : \quad : : \\ \hline \exists \vec{y}.\ \phi_n(\vec{y}) : t_n \\ \hline \end{array} \tag{A.10}$$

$$= \text{casemax} \begin{array}{|c|}\hline \phi_1(\vec{c}_1) \vee \ldots \vee \phi_1(\vec{c}_{|A(\vec{y})|}) : t_1 \\ \hline : \quad : : \\ \hline \phi_n(\vec{c}_1) \vee \ldots \vee \phi_n(\vec{c}_{|A(\vec{y})|}) : t_n \\ \hline \end{array} \tag{A.11}$$

$$= \text{casemax} \begin{array}{|l|} \hline \phi_1(\vec{c_1}) \quad : t_1 \\ \hline : \quad\quad\quad : \; : \\ \hline \phi_1(\vec{c}_{|A(\vec{y})|}) : t_1 \\ \hline : \quad\quad\quad : \; : \\ \hline \phi_n(\vec{c_1}) \quad : t_n \\ \hline : \quad\quad\quad : \; : \\ \hline \phi_n(\vec{c}_{|A(\vec{y})|}) : t_n \\ \hline \end{array} \tag{A.12}$$

$$= \max \left(\begin{array}{|l|} \hline \phi_1(\vec{c_1}) : t_1 \\ \hline : \quad\quad : \; : \\ \hline \phi_n(\vec{c_1}) : t_n \\ \hline \end{array} , \; \cdots \; , \begin{array}{|l|} \hline \phi_1(\vec{c}_{|A(\vec{y})|}) : t_1 \\ \hline : \quad\quad\quad : \; : \\ \hline \phi_n(\vec{c}_{|A(\vec{y})|}) : t_n \\ \hline \end{array} \right) \tag{A.13}$$

$$= \max_{a \in \{A(\vec{c_1}), \ldots, A(\vec{c}_{|A(\vec{y})|})\}} qCase^{t+1,\mathcal{D}}(a) \tag{A.14}$$

$$\longmapsto \max_{a \in \{A(\vec{c_1}), \ldots, A(\vec{c}_{|A(\vec{y})|})\}} Q^{t+1}(\vec{x}, a) \tag{A.15}$$

The LHS of Equation A.10 is just the LHS of the theorem. The first equivalence in Equation A.10 follows from the properties of the $\exists$ quantifier and the disjunction of case elements within a case statement. The second equivalence in Equation A.11 follows from the application of domain closure assumptions. The third equivalence in Equation A.12 is a rewrite of the case elements since they are already disjunctively defined and we can distribute the conjoined $t = t_i$ terms for each case element into the disjunction.

The rewrite between Eqs. A.12 and A.13 follows from the lemma immediately following this proof (note that we have switched from unary casemax to n-ary $\max$). The transformation from Equation A.13 to Equation A.14 simply involves a notational substitution and the final step follows from the correspondence of $qCase^{t+1,\mathcal{D}}(a) \longmapsto Q^{t+1}(\vec{x}, a)$ proved in Theorem A.5.1 for a ground action a and the correspondence of the case ($\max$) operation proved in Theorem A.3.4. Thus, the theorem follows. □

Lemma A.6.2. *The transformation between Eqs. A.12 and A.13 is an equivalence-preserving transform.*

Proof. The rewrite between Eqs. A.12 and A.13 follows from the semantics of the unary case

maximization operator and n-ary maximization operators discussed in Chapter 4. Assume for an interpretation $\Im$ that $\Im \models \phi_i$ where $\langle \phi_i, t_i \rangle$ is an element of case statement in Equation A.12 and for all other case elements $\langle \phi_j, t_j \rangle$ either $t_j \leq t_i$ or $\Im \not\models \phi_j$.

Let $\langle \phi_m, t_m \rangle$ be a case element in the resulting cross-product used to compute the n-ary maximization of Equation A.13. To show the equivalence of the semantics of this maximization with the previous from Equation A.12, we need to show that if $\Im \models \phi_m$ then $t_m = t_i$. We assume the condition and break this into two cases:

(1) ϕ_m was formed from a partition conjoined with $\langle \phi_i, t_i \rangle$, or

(2) ϕ_m was not formed from a partition conjoined with $\langle \phi_i, t_i \rangle$.

For (1), we know that if ϕ_m is consistent then all other $\langle \phi_j, t_j \rangle$ must have $t_j \leq t_i$, so the max yields $t_m = t_i$. For (2), we can show that $\Im \models \neg\phi_i$ due to the mutual exclusivity of all partitions in the n-ary cross-product. Since this violates our previous assumption that $\Im \models \phi_i$, we know this case is vacuous and (1) must hold. Thus, the equivalence of Eqs. A.12 and A.13 follows. □

Now we complete the (symbolic) dynamic programming step. For the ground case, we specify the completion of the dynamic programming step as the following maximization (where $a \in A_i^{\mathcal{D}}(\vec{y})$ denotes that a ranges over all ground instantiations $A(\vec{c})$ of action template $A_i(\vec{y})$ given domain $\mathcal{D}$):

$$V^{t+1}(\vec{x}) = \max_{i=1\ldots p} \max_{a \in A_i^{\mathcal{D}}(\vec{y})} Q^{t+1}(\vec{x}, a) \tag{A.16}$$

And for the FOMDP representation, we break the symbolic dynamic programming step into the following two step maximization given by the following:

$$vCase^{t+1} = \max_{i=1\ldots p} \text{casemax } \exists \vec{y}.\ qCase^{t+1}(A_i(\vec{y})) \tag{A.17}$$

Now we prove the final theorem that guarantees correspondence of dynamic programming and symbolic dynamic programming for one step:

Theorem A.6.3. *Define the FOMDP, MDP w.r.t. domain $\mathcal{D}$, corresponding value functions $vCase^{t,\mathcal{D}}$ and $V^t(\vec{x})$, and Q-functions $qCase^{t+1,\mathcal{D}}(A(\vec{y}))$ and $Q^{t+1}(\vec{x}, A(\vec{c}))$ as in Theorem A.6.1. Let $vCase^{t+1,D}$ be obtained from computing Equation A.16 and let $V^{t+1}(\vec{x})$ be computed from Equation A.17. Then $vCase^{t+1,\mathcal{D}} \longmapsto V^{t+1}(\vec{x})$.*

Proof. Based on the assumptions and Theorem A.6.1, we know

$$\text{casemax}\, \exists \vec{y}.\ qCase^{t+1}(A_i(\vec{y})) \longmapsto \max_{a \in A_i^{\mathcal{D}}(\vec{y})} Q^{t+1}(\vec{x}, a).$$

The remainder of the theorem follows from the correspondence of the $\max$ operands from Lemma A.3 and operator applications, for which correspondence has already been proved in Theorem A.3.4. □

A.7 Correspondence of Symbolic and Ground Value Iteration

Our previous theorems lead us to the following obvious result:

Theorem A.7.1. *Given $rCase^{\mathcal{D}}$, $pCase^{\mathcal{D}}(n_j(\vec{c}), A(\vec{c}))$, and γ from a FOMDP, obtain $R(\vec{x})$, $P(\vec{x}'|\vec{x}, A_i(\vec{c}))$, and γ for an MDP w.r.t. domain $\mathcal{D}$ from Definition A.4.1. Let $vCase^{t,\mathcal{D}}$ (for $t > 0$) be obtained by applying symbolic dynamic programming for t steps. Let $V^t(\vec{x})$ (for $t > 0$) be obtained by applying ground dynamic programming for t steps. Then $vCase^{t,\mathcal{D}} \longmapsto V^t(\vec{x})$ for all $t \geq 0$.*

Proof. The proof is by induction. As defined respectively in Chapters 3 and 4 $V^0(\vec{x}) = R(\vec{x})$ and $vCase^{0,\mathcal{D}} = rCase^{\mathcal{D}}$. By Lemma A.3, we know that $vCase^{0,\mathcal{D}} \longmapsto V^0(\vec{x})$ for any finite domain $\mathcal{D}$. This provides the base case for $t = 0$. For the inductive step, we have shown that given $vCase^{t,\mathcal{D}} \longmapsto V^t(\vec{x})$, we can prove $vCase^{t+1,\mathcal{D}} \longmapsto V^{t+1}$ after applying one step of (symbolic) dynamic programming to the respective value representations via Theorem A.6.3. This proves the inductive case for all $t > 0$. Thus the theorem follows. □

Appendix B

Remaining Proofs

B.1 Proofs from Chapter 3

Lemma 3.4.2. *Fix a variable ordering over $x_1, \ldots, x_n$. For any function $g(x_1, \ldots, x_n)$ mapping $\mathbb{B}^n \longrightarrow \mathbb{R}$, there exists a unique generalized AADD G over variable domain $x_1, \ldots, x_n$ satisfying the given variable ordering such that for all $\rho \in \mathbb{B}^n$ we have $g(\rho) = Val(G, \rho)$.*

Proof. We prove this lemma by induction on n. For $n = 0$, we have a function representing a constant C. The constraints imply that $G = C + 0 \cdot 0$ is the only legal representation of this function.

Now, for the inductive case, we assume that we have n variables in our function $f(x_1, \ldots, x_n)$ with variable x_1 first in the ordering. We inductively assume that the lemma holds for the representation of the functions $f_h(x_1 = \mathit{true}, x_2, \ldots, x_n)$ and $f_l(x_1 = \mathit{false}, x_2, \ldots, x_n)$ over $n - 1$ variables so that both of these functions are represented by unique generalized AADDs $G_h = c_h + b_h F_h$ and $G_l = c_l + b_l F_l$. If $G_h = G_l$, then this case is satisfied by our inductive assumption since $f(x_1, \ldots, x_n)$ technically ranges over $n - 1$ variables. Otherwise $G_h \neq G_l$, so the only way to represent $f(x_1, \ldots, x_n)$ in the grammar is to use an if node branching on x_1. Constraint (1) implies that we can have at most one if node above F_h and F_l branching on variable x_1. So we build $F = \mathit{if}\ (F^{var})\ \mathit{then}\ c'_h + b'_h F_h\ \mathit{else}\ c'_l + b'_l F_l$ and $G = c + bF$ to represent $f(x_1, \ldots, x_n)$. Let $r_{\min} = \min(c_h, c_l)$, $r_{\max} = \max(c_h + b_h, c_l + b_l)$, and $r_{\mathit{range}} = r_{\max} - r_{\min}$; these respectively denote the minimum, maximum, and value span of the child functions G_l and G_h, which allow us to normalize the newly constructed F node to have a range of $[0, 1]$, while at the same time providing us with the offset c and multiplier b for the newly constructed G node.

Now, we must solve for $c, b, c'_h, b'_h, c'_l, b'_l$ that satisfy constraints (2) and (3). This gives us the following six equations that must be simultaneously satisfied:

$$\begin{aligned} c &= r_{\min} \\ b &= r_{\min} + r_{range} \\ c_h &= b \cdot c'_h + c \\ b_h &= b \cdot b'_h \\ c_l &= b \cdot c'_l + c \\ b_l &= b \cdot b'_l \end{aligned}$$

In matrix form, this linear system is non-singular when $b > 0$, which follows from $r_{\min} + r_{range} > 0$ as implied by constraint (4). Thus, the matrix is full rank and the linear system has one unique solution. By simple Gaussian elimination, we can derive this unique solution as the following: $c = r_{\min}$, $b = r_{\min} + r_{range}$, $c'_h = \frac{c_h - r_{\min}}{r_{range}}$, $c'_l = \frac{c_l - r_{\min}}{r_{range}}$, $b'_h = \frac{b_h}{r_{range}}$, $b'_l = \frac{b_l}{r_{range}}$. This shows us that there is only one unique construction of G to represent $f(x_1, \ldots, x_n)$. Thus, the inductive case is satisfied and the statement of the lemma follows. □

Theorem 3.4.3. *For all functions $F_1 : \mathbb{B}^n \longrightarrow \mathbb{R}$ and $F_2 : \mathbb{B}^m \longrightarrow \mathbb{R}$ ($n \geq 0$ and $m \geq 0$), the time and space performance of $Reduce(F_1)$ and $Apply(F_1, F_2,\ op\ )$ for AADDs (operands and results represented as canonical AADDs) is within a multiplicative constant of $Reduce(F_1)$ and $Apply(F_1, F_2,\ op\ )$ for ADDs (operands and results represented as canonical ADDs) in the worst case assuming any fixed variable ordering.*

Proof. The ADD $Reduce$ and $Apply$ algorithms can be seen as analogs of the corresponding AADD algorithms without the overhead of progagating the affine transforms of edge weights during recursive calls and normalizing them when returning. However, a comparison of the ADD/AADD $Reduce$ and $Apply$ algorithms shows that there are only a constant number of additional constant time operations for manipulating edge weights in each AADD algorithm in comparison to the corresponding ADD algorithm. Thus each call to the AADD algorithm incurs an additional constant time overhead over the corresponding call to the ADD algorithm. We denote the respective constant time to evaluate one ADD $Reduce$ or $Apply$ call to be T_{ADD}^{Reduce} and T_{ADD}^{Apply}, respectively. Likewise, we denote the respective time to evaluate one AADD $Reduce$ or $Apply$ call to be $T_{AADD}^{Reduce} = T_{ADD}^{Reduce} + C_{AADD}^{Reduce}$ and $T_{AADD}^{Apply} = T_{ADD}^{Apply} + C_{AADD}^{Apply}$ where C represents the additional constant time overhead of the call for an AADD in compar-

ison to the ADD.

Now, we only need to show that the AADD makes equal or fewer calls to $Reduce$ and $Apply$ than the ADD version. First we note that under the same variable ordering, an ADD is equivalent to a non-canonical AADD with fixed edge weights $c = 0, b = 1$. Thus, if we did not normalize AADD nodes in $Reduce$ and $Apply$, then there would be a direct 1-1 mapping between each $Reduce$ and $Apply$ call for ADDs and the corresponding call for AADDs. Since normalization can only increase the number of $Reduce$ and $Apply$ cache hits and reduce the number of cached nodes, it is clear that an AADD must generate equal or fewer $Reduce$ and $Apply$ calls and have equal or fewer cached nodes than the corresponding ADD. This allows us to conclude that in the worst case, the AADD generates as many $Reduce$ and $Apply$ calls and cache hits as the ADD. Assuming n calls are made by both the ADD and AADD variants of $Reduce$ and $Apply$, then the ADD requires total time nT_{ADD}^{Reduce} and nT_{ADD}^{Apply} for each respective algorithm whereas the AADD requires time $n(T_{ADD}^{Reduce} + C_{AADD}^{Reduce})$ and $n(T_{ADD}^{Apply} + C_{AADD}^{Apply})$ for each respective algorithm. This verifies that the AADD operations are within a multiplicative constant of the time required by the corresponding ADD operations (specifically, $\frac{T_{ADD}^{Reduce}+C_{AADD}^{Reduce}}{T_{ADD}^{Reduce}}$ for $Reduce$ and $\frac{T_{ADD}^{Apply}+C_{AADD}^{Apply}}{T_{ADD}^{Apply}}$ for $Apply$).

An analogous proof for space can be obtained by substituting "space" for "time" above. □

Theorem 3.4.4. *There exist functions F_1 and F_2 and an operator op such that the running time and space performance of $Apply(F_1, F_2, op)$ for AADDs can be linear in the number of variables when the corresponding ADD operations are exponential in the number of variables.*

Proof. Two functions and $Apply$ operation examples where this holds true are $\sum_{i=1}^{n} 2^i x_i \oplus \sum_{i=1}^{n} 2^i x_i$ and $\prod_{i=1}^{n} \gamma^{2^i x_i} \otimes \prod_{i=1}^{n} \gamma^{2^i x_i}$. (Examples of these operands as ADDs and AADDs were given in Figures 3.7(c) and 3.8.) Because these computations result in a number of terminal values exponential in n, the ADD operations must require time and space exponential in n. On the other hand, it is known that the operands can be represented in linear-sized AADDs. Due to this structure, the $Apply$ algorithm will begin by recursing on the high branch of both operands to depth n. Then, at each step as it returns and recurses down the low branch of decision test x_i, the respective additive difference of 2^i and multiplicative coefficient of γ^{2^i} in the corresponding high-branch and low-branch $Apply$ operation calls will be normalized out for the respective operations of $\oplus$ and $\otimes$ due to the canonical caching scheme in Table 3.2, thus yielding cache hits for all low branches. For each operation on the specified pair of functions, this results in n cached nodes and $2n$ $Apply$ calls for the AADD operations. □

B.2 Proofs from Chapter 5

Theorem 5.2.1. *Let $V(s)$ be the approximated value function obtained by the weights $\vec{w}^{(i)}$ of the final LP solution of Equation 5.21 for FOAPI applied to a given FOMDP where FOAPI has converged. Let $\beta^{(i)}$ be the objective value of this final LP solution. Then the error bounds on $V_{\tilde{\pi}}(s)$ (the value function obtained by acting according to the greedy policy $\tilde{\pi}$ w.r.t. $V(s)$) derived from plugging $\beta^{(i)}$ in for β in Equation 2.19 hold for all possible finite ground domain instantiations of this FOMDP.*

Proof. At convergence, we know that $\vec{w}^{(i)} = \vec{w}^{(i-1)}$. Then in summary, we note that the constraints in the FOAPI LP provide a bound on the Bellman error of the value function corresponding to $\vec{w}^{(i)}$, which we can then use to derive a bound on the error of acting according to a greedy policy w.r.t. this value function as described for MDPs in Section 2.5.2 of Chapter 2.

Define the $SDP[\cdot]$ operator to take a value function $V(s)$ and compute its one-step symbolic dynamic programming backup under the FOMDP dynamics as defined in Section 4.4. Then we break the proof into two parts:

1. *At convergence*, the constraints can be transformed with the following series of rewrites explained below:

$$\begin{aligned}
\beta^{(i)} &\geq \left| R(s) \oplus \exists \vec{x}^* \ \left(\pi^{(i)}_{A(\vec{x}^*)}(s) \oplus B^{A(\vec{x}^*)} \left[\bigoplus_{j=1}^{k} w_j^{(i)} \cdot b_j(s) \right] \right) \ominus \bigoplus_{j=1}^{k} w_j^{(i)} \cdot b_j(s) \ \right| ; \ \forall A, s \\
&\geq \left| R(s) \oplus \exists \vec{x}^* \ \left(\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)] \oplus B^{A(\vec{x}^*)}[V(s)] \right) \ominus V(s) \ \right| ; \ \forall A, s \qquad \text{(B.1)} \\
&\geq |SDP[V(s)] \ominus V(s) \ | ; \ \forall s \qquad \text{(B.2)}
\end{aligned}$$

   To obtain Equation B.1, we simply substituted $V(s)$ in for its linear-value representation as defined in the statement of the theorem. To make it explicit that the policy case statements $\pi^{(i)}_{A(\vec{x}^*)}(s)$ were derived from $V(s)$, we make this explicit using the notation $\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)]$. To obtain Equation B.2 we apply Lemma B.2.1 below.

2. By Theorem A.7.1 (that proves the correspondence between symbolic dynamic programming and ground dynamic programming) and Theorem A.3.4 (that proves the correspondence of the case operator $\ominus$ for the first-order and grounded representations of a case statement) both from Appendix A, we know that the constraints in Equation B.2 hold for all ground domain instantiations. Since these ground constraints imply a Bellman error $\beta^{(i)}$ for all ground domain instantiations of the approximated value function $V(s)$,

the error bounds on $V_{\tilde{\pi}}(s)$ (the value function obtained by acting according to the greedy policy $\tilde{\pi}$ w.r.t. $V(s)$) derived from plugging $\beta^{(i)}$ in for β in Equation 2.19 hold for all ground domain instantiations.

The final result of step 2 proves the theorem. □

Lemma B.2.1. *Under the assumptions and notation of Theorem 5.2.1, if the constraints*

$$\beta^{(i)} \geq \left| R(s) \oplus \exists \vec{x}^* \ \left(\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)] \oplus B^{A(\vec{x}^*)}[V(s)] \right) \ominus V(s) \ \right| ; \ \forall A, s$$

hold then the constraints $\beta^{(i)} \geq |SDP[V(s)] \ominus V(s)\ |;\ \forall s$ *must also hold.*

Proof. We prove this through the following derivation, which we justify below:

$$\beta^{(i)} \geq \left| R(s) \oplus \exists \vec{x}^* \ \left(\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)] \oplus B^{A(\vec{x}^*)}[V(s)] \right) \ominus V(s) \ \right| ; \ \forall A, s \tag{B.3}$$

$$\geq \left| R(s) \oplus \bigcup_A \left[\exists \vec{x}^* \ \left(\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)] \oplus B^{A(\vec{x}^*)}[V(s)] \right) \right] \ominus V(s) \ \right| ; \ \forall s \tag{B.4}$$

$$\geq \left| R(s) \oplus \bigcup_A \left[\exists \vec{x}^* \ \left(\begin{array}{|c|} \hline \psi_{A,1} \wedge \phi_{A,1}(\vec{x}^*, s) \ : 0 \\ \hline \quad : \wedge : \qquad\qquad : : \\ \hline \psi_{A,n} \wedge \phi_{A,n}(\vec{x}^*, s) : 0 \\ \hline \end{array} \oplus \begin{array}{|c|} \hline \phi_{A,1}(\vec{x}^*, s) : t_{A,1} \\ \hline : \qquad\qquad : \ : \\ \hline \phi_{A,n}(\vec{x}^*, s) : t_{A,n} \\ \hline \end{array} \right) \right] \ominus V(s) \ \right| ; \ \forall s \tag{B.5}$$

$$\geq \left| R(s) \oplus \bigcup_A \begin{array}{|c|} \hline \psi_{A,1} \wedge \exists \vec{x}^* \phi_{A,1}(\vec{x}^*, s) \ : t_{A,1} \\ \hline \quad : \wedge : \qquad\qquad : \ : \\ \hline \psi_{A,n} \wedge \exists \vec{x}^* \phi_{A,n}(\vec{x}^*, s) : t_{A,n} \\ \hline \end{array} \ominus V(s) \ \right| ; \ \forall s \tag{B.6}$$

$$\geq |R(s) \oplus \pi(s)[V(s)] \ominus V(s)| ; \ \forall s \tag{B.7}$$

$$\geq \left| R(s) \oplus \text{casemax} \left(\bigcup_A B^A[V(s)] \right) \ominus V(s) \right| ; \ \forall s \tag{B.8}$$

$$\geq |SDP[V(s)] \ominus V(s)\ |;\ \forall s \tag{B.9}$$

We assume that the constraints in Equation B.3 hold and proceed to rewrite them in Equation B.4 where we have exploited the disjointness of the policy $\pi^{(i)}_{A(\vec{x}^*)}$ for each A; that is, for any situation s, if a partition of $\pi^{(i)}_{A(\vec{x}^*)}$ holds true then no partition of $\pi^{(i)}_{B(\vec{x}^*)}$ for $B \neq A$ can also hold true. This guarantees equivalence to Equation B.3 since the constraint is "active" for one A in each s and thus Equation B.4 self-selects which A constraint should apply for each s.

The transform to Equation B.5 is purely notational where we have written out the case

statements for $B^{A(\vec{x}^*)}[V(s)]$ and $\pi^{(i)}_{A(\vec{x}^*)}$. Note that the case statements share the formula prefixed with ϕ since the policy case statement $\pi^{(i)}_{A(\vec{x}^*)}(s)[V(s)]$ for action A was derived from $B^{A(\vec{x}^*)}[V(s)]$ (c.f., Section 4.4.4). The formulae prefixed with ψ represent the additional guard formulae added to each policy partition to prevent it from applying when higher valued policy partitions were possible. Then, performing the explicit "cross-sum" $\oplus$ (and removing all inconsistent partitions) and distributing the $\exists \vec{x}^*$ into the result, we obtain Equation B.6.

We recognize that the case statements within the $\cup$ in Equation B.6 are just the action-specific policy case statements $\pi_A(s)[V(s)]$, which were derived in Section 5.2.2 of Chapter 5 by partitioning the policy $\pi(s)[V(s)]$ into case statements for each action. Thus we apply the $\cup$ to reverse this derivation and obtain Equation B.7. Next we substitute $\pi(s)[V(s)]$ with its derivation from the RHS of Equation 4.28 (c.f. Section 4.4.4 of Chapter 4) to obtain Equation B.8. Noting that the first two terms of Equation B.8 are just the definition of $SDP[V(s)]$, we obtain the final result in Equation B.9 that proves the lemma. □

B.3 Proofs from Chapter 6

Proposition 6.1.1. *$P(a|A(\vec{x}), s)$ defines a proper probability distribution over a, i.e.,*

$$\sum_{a \in N_A(\vec{x})} P(a|A(\vec{x}), s) = \boxed{\top : 1}. \tag{B.10}$$

Proof. We can effectively sum over all $a \in N_A(\vec{x})$ by generating a summation over all possible instantiations of a according to its definition in Equation 6.14. Doing this and substituting the definition of $P(a|A(\vec{x}), s)$ from Equation 6.16, we obtain the following:

$$\sum_{a \in N_A(\vec{x})} P(a|A(\vec{x}), s) = \bigoplus_{n_1(\vec{x},\dot{\vec{y}}_1) \in N_1(\vec{x},\dot{\vec{y}}_1), \ldots, n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}) \in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}), \ldots, n_1(\vec{x},\dot{\vec{y}}_1) \in N_1(\vec{x},\dot{\vec{y}}_1), \ldots, n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}) \in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})} \left(\prod_{\vec{y} \in \{\dot{\vec{y}}_1, \ldots, \dot{\vec{y}}_{|\vec{y}|}\}} \prod_{i=1}^{p} P(n_i(\vec{x}, \vec{y})|A_i(\vec{x}, \vec{y}), s) \right)$$

Now, we can exploit the independence of the random variables $n_i(\vec{x}, \dot{\vec{y}})$ inherent in this factored representation of the joint probability distribution to marginalize over each $n_i(\vec{x}, \dot{\vec{y}})$ independently. Here we push the first marginalization into its only relevant factor, where the result

follows from the properties of $P(n_i(\vec{x},\dot{\vec{y}})|A_i(\vec{x},\dot{\vec{y}}),s)$ as defined in Equation 6.13:

$$\begin{aligned}
\sum_{a\in N_A(\vec{x})} P(a|A(\vec{x}),s) = & \bigoplus_{n_2(\vec{x},\dot{\vec{y}}_1)\in N_2(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}),\ldots,n_1(\vec{x},\dot{\vec{y}}_1)\in N_1(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})} \\
& \left[\left(\prod_{\vec{y}\in\{\dot{\vec{y}}_1,\ldots,\dot{\vec{y}}_{|\vec{y}|}\}} \prod_{\text{if } \vec{y}=\dot{\vec{y}}_1 \text{ then } i=1 \text{ else } i=2}^{p} P(n_i(\vec{x},\vec{y})|A_i(\vec{x},\vec{y}),s)\right)\right. \\
& \left.\left(\bigoplus_{n_1(\vec{x},\dot{\vec{y}}_1)\in N_1(\vec{x},\dot{\vec{y}}_1)} P(n_1(\vec{x},\dot{\vec{y}}_1)|A_1(\vec{x},\dot{\vec{y}}_1),s)\right)\right] \\
= & \bigoplus_{n_2(\vec{x},\dot{\vec{y}}_1)\in N_2(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|}),\ldots,n_1(\vec{x},\dot{\vec{y}}_1)\in N_1(\vec{x},\dot{\vec{y}}_1),\ldots,n_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})\in N_p(\vec{x},\dot{\vec{y}}_{|\vec{y}|})} \\
& \left[\left(\prod_{\vec{y}\in\{\dot{\vec{y}}_1,\ldots,\dot{\vec{y}}_{|\vec{y}|}\}} \prod_{\text{if } \vec{y}=\dot{\vec{y}}_1 \text{ then } i=1 \text{ else } i=2}^{p} P(n_i(\vec{x},\vec{y})|A_i(\vec{x},\vec{y}),s)\right) \boxed{\top : 1}\right]
\end{aligned}$$

Repeating this marginalization process indefinitely until none remain, we obtain the final result:

$$\sum_{a\in N_A(\vec{x})} P(a|A(\vec{x}),s) = \boxed{\top : 1}$$

This proves the proposition. □

Proposition 6.1.4. *Let $A_i(\vec{x},\dot{\vec{y}}_j)$ and $A_h(\vec{x},\dot{\vec{y}}_k)$ be two distinct aspects of stochastic action $A(\vec{x})$ (i.e., either $i \neq h$ or $\dot{\vec{y}}_j \neq \dot{\vec{y}}_k$) and recall that $N_i(\vec{x},\dot{\vec{y}}_j)$ and $N_h(\vec{x},\dot{\vec{y}}_k)$ are the respective sets of Nature's deterministic sub-action outcomes for each of these aspects. Then for all $A(\vec{x})$ and $i \neq h$ and $\dot{\vec{y}}_j \neq \dot{\vec{y}}_k$ where $n_i(\vec{x},\dot{\vec{y}}_j) \in N_i(\vec{x},\dot{\vec{y}}_j)$ and $n_h(\vec{x},\dot{\vec{y}}_k) \in N_h(\vec{x},\dot{\vec{y}}_k)$, if the following condition holds:*

$$\begin{aligned}
\forall\vec{x}.\ [\ P(n_i(\vec{x},\dot{\vec{y}}_j)|A(\vec{x}),s) > 0 \ \ \wedge \ \ P(n_h(\vec{x},\dot{\vec{y}}_k)|A(\vec{x}),s) > 0 \\
\supset\ (E_{n_i(\vec{x},\dot{\vec{y}}_j)} \wedge E_{n_h(\vec{x},\dot{\vec{y}}_k)})\ \textit{is consistent}\ ]
\end{aligned}$$

then Assumption 6.1.3 must hold for the given factored FOMDP.

Proof. By definition of inconsistency and the conjunctive effect representation, we note that the effect set $E_a = \bigcup_{\dot{\vec{y}},\, i=1\ldots p} E_{n_i(\vec{x},\dot{\vec{y}})}$ for a joint action a is inconsistent iff $\bigwedge_{\dot{\vec{y}},\, i=1\ldots p} E_{n_i(\vec{x},\dot{\vec{y}})}$ contains at least two respective conjoined fluents of the form $F(\vec{x},s)$ and $\neg F(\vec{x},s)$. Since we previously assumed that a single $E_{n_i(\vec{x},\dot{\vec{y}})}$ is consistent, these two fluents must have been

contributed by two sub-action effect sets $E_{n_i(\vec{x},\dot{\vec{y}}_j)}$ and $E_{n_h(\vec{x},\dot{\vec{y}}_k)}$ where either $i \neq h$ or $\dot{\vec{y}}_j \neq \dot{\vec{y}}_k$. In addition, since $P(a|A(\vec{x}), s) > 0$, it must trivially hold that

$$P(n_i(\vec{x}, \dot{\vec{y}}_j)|A(\vec{x}), s) > 0 \quad \wedge \quad P(n_h(\vec{x}, \dot{\vec{y}}_k)|A(\vec{x}), s) > 0$$

since both of these are multiplicative factors of $P(a|A(\vec{x}), s)$ by definition in Equation 6.16.

Since every inconsistent joint action a with $P(a|A(\vec{x}), s) > 0$ has at least two constituent sub-actions $n_i(\vec{x}, \dot{\vec{y}}_j)$ and $n_h(\vec{x}, \dot{\vec{y}}_k)$ with inconsistent effects and $P(n_i(\vec{x}, \dot{\vec{y}}_j)|A(\vec{x}), s) > 0$ and $P(n_h(\vec{x}, \dot{\vec{y}}_k)|A(\vec{x}), s) > 0$, this proves Proposition 6.1.4. □

Proposition 6.2.2. (Removal of Irrelevant Aspects)

$$\begin{aligned} &Irr[\phi(s), A_i(\vec{x}, \vec{y})] \supset \\ &\quad \big\{\forall\, n_i(\vec{x}, \vec{y}) \in N_i(\vec{x}, \vec{y}),\ \ \forall a \in N_A(\vec{x}),\ \ \forall c.\ (a = n_i(\vec{x}, \vec{y}) \circ c) \supset \\ &\qquad Regr[\phi(s), n_i(\vec{x}, \vec{y}) \circ c] \equiv Regr[\phi(s), c]\big\} \end{aligned}$$

Proof. From the definition of $Irr[\phi(s), A_i(\vec{x}, \vec{y})]$ we know that

$$\forall n_i(\vec{x}, \vec{y}) \in N_i(\vec{x}, \vec{y}).\ Regr(\phi(do(n_i(\vec{x}, \vec{y}), s))) \equiv \phi(s).$$

By this result and the construction of SSAs in terms of factored effect axioms derived from $E_{n_i(\vec{x},\vec{y})}$ (c.f., Section 6.1.3 and Equation 6.21), we can conclude that all disjoined elements of the SSAs relevant to fluents of $\phi(\cdot)$ contributed by $E_{n_i(\vec{x},\vec{y})}$ simplified to $\perp$ and were removed during $Regr[\phi(do(n_i(\vec{x}, \vec{y})))]$, thus yielding $\phi(s)$. Since the effects considered in the regression of $Regr(\phi(do(c \circ n_i(\vec{x}, \vec{y}), s)))$ are compiled from $E_c \cup E_{n_i(\vec{x},\vec{y})}$ and we know that the disjoined elements of the SSAs relevant to fluents of $\phi(\cdot)$ contributed by $E_{n_i(\vec{x},\vec{y})}$ simplify to $\perp$, the effects contributed by $E_{n_i(\vec{x},\vec{y})}$ can be ignored during this regression. Then the only effects relevant to the regression on the LHS and RHS of the final line of the proposition are E_c and thus the regressions are equivalent. □

Bibliography

[Andre and Russell, 2001] David Andre and Stuart Russell. Programmable reinforcement learning agents. In *In Advances in Neural Information Processing Systems*, volume 13, 2001.

[Andre and Russell, 2002] David Andre and Stuart Russell. State abstraction for programmable reinforcement learning agents. In *In Proc. AAAI-02*, Edmonton, Alberta, 2002. AAAI Press.

[Bacchus and Grove, 1995] Fahiem Bacchus and Adam Grove. Graphical models for preference and utility. In *Uncertainty in Artificial Intelligence. Proceedings of the Eleventh Conference (1995)*, pages 3–10, San Francisco, 1995. Morgan Kaufmann Publishers.

[Bacchus and Kabanza, 2000] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[Bacchus *et al.*, 1995] Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors in the situation calculus. In *IJCAI-95*, pages 1933–1940, Montreal, 1995.

[Bahar *et al.*, 1993] R. Iris Bahar, Erica Frohm, Charles Gaona, Gary Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic Decision Diagrams and their applications. In *IEEE /ACM International Conference on CAD*, 1993.

[Baier *et al.*, 2007] J. Baier, F. Bacchus, and S. McIlraith. A heuristic search approach to planning with temporally extended preferences. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 1808–1815, Hyderabad, India, January 2007.

[Barto and Sutton, 1998] Andrew Barto and Richard Sutton. *Reinforcement Learning*. MIT Press, 1998.

[Barto *et al.*, 1993] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, U. Mass. Amherst, , 1993.

[Bellman, 1957] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bertsekas and Tsitsiklis, 1996] Dmitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[Bertsekas, 1987] Dmitri P. Bertsekas. *Dynamic Programming*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[Bienvenu *et al.*, 2006] M. Bienvenu, C. Fritz, and S. McIlraith. Planning with qualitative temporal preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 134–144, Lake District, UK, June 2006.

[Blum and Furst, 1995] Avrim L. Blum and Merrick L. Furst. Fast planning through graph analysis. In *IJCAI 95*, pages 1636–1642, Montreal, 1995.

[Bonet and Geffner, 2004] Blai Bonet and Hector Geffner. 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/*, 2004.

[Boutilier *et al.*, 1995a] Craig Boutilier, Thomas Dean, and Steve Hanks. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the Third European Workshop on Planning*, Assisi, Italy, 1995.

[Boutilier *et al.*, 1995b] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *IJCAI 95*, San Francisco, 1995.

[Boutilier *et al.*, 1996] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *UAI-96*, pages 115–123, Portland, OR, 1996.

[Boutilier *et al.*, 1997] Craig Boutilier, Ronen I. Brafman, and Christopher Geib. Prioritized goal decomposition of Markov decision processes: Toward a synthesis of classical and decision theoretic planning. In *IJCAI-97*, pages 1156–1162, Nagoya, 1997.

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

[Boutilier *et al.*, 2000] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI-00*, pages 355–362, Austin, TX, 2000.

[Boutilier *et al.*, 2001] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*, pages 690–697, Seattle, 2001.

[Brachman and Levesque, 2004] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Brafman and Chernyavsky, 2005] Ronen I. Brafman and Yuri Chernyavsky. Planning with goal preferences and constraints. In *International Conference on Automated Planning and Scheduling*, 2005.

[Bryant and Chen, 1995] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Design Automation Conference*, pages 535–541, 1995.

[Bryant, 1986] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.

[Buffet and Aberdeen, 2006] Olivier Buffet and Douglas Aberdeen. The factored policy gradient planner (ipc-06 version). In *Proceedings of the Fifth International Planning Competition*, 2006.

[Chen and Bryant, 1997] Yirng-An Chen and Randal E. Bryant. PHDD: an efficient graph representation for floating point circuit verification. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 2–7, Washington, DC, 1997.

[Cimatti and Roveri, 1999] Alessandro Cimatti and Marco Roveri. Conformant planning via model checking. In *ECP*, pages 21–34, 1999.

[de Farias and Van Roy, 2003] DP de Farias and Ben Van Roy. The linear programming approach to approximate dynamic programming. *Operations Research*, 51:6:850–865, 2003.

[de Salvo Braz *et al.*, 2005] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *IJCAI-05*, Edinburgh, UK, 2005.

[de Salvo Braz *et al.*, 2006] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. MPE and partial inversion in lifted probabilistic variable elimination. In *AAAI-06*, Boston, USA, 2006.

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

[Dearden and Boutilier, 1997] Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(12):219–283, 1997.

[Dechter, 1999] Rina Dechter. Bucket elimination: A unifying framework for reasoning. In *Artificial Intelligence*, volume 113, pages 41–85, 1999.

[Drechsler and Sieling, 2001] R. Drechsler and D. Sieling. Binary decision diagrams in theory and practice. In *Software Tools for Technology Transfer*, volume 3, 2001.

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

[Ferrein *et al.*, 2003] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. Extending DTGolog with Options. In *IJCAI-2003*, Acupulco, Mexico, 2003.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AI Journal*, 2:189–208, 1971.

[Fox and Long, 2001] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains., 2001.

[Gardiol and Kaelbling, 2004] Natalia H. Gardiol and Leslie Pack Kaelbling. Envelope-based planning in relational MDPs. In *Advances in Neural Information Processing Systems 16 (NIPS-03)*, Vancouver, CA, 2004.

[Gerevini *et al.*, 2006] Alfonso Gerevini, Blai Bonet, and Bob Givan, editors. *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, 2006.

[Gretton and Thiebaux, 2004] Charles Gretton and Sylvie Thiebaux. Exploiting first-order regression in inductive policy selection. In *UAI-04*, pages 217–225, Banff, Canada, 2004.

[Gretton *et al.*, 2004] Charles Gretton, David Price, and Sylvie Thiebaux. NMRDPP: Decision-theoretic planning with control knowledge. In *Online Proceedings for The Probablistic Planning Track of IPC-04: http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*, 2004.

[Guestrin *et al.*, 2001] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projections for factored MDPs. In *IJCAI-01*, pages 673–680, Seattle, 2001.

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

[Guestrin *et al.*, 2003] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI-03*, Acapulco, Mexico, 2003.

[Guestrin *et al.*, 2004] Carlos Guestrin, Milos Hauskrecht, and Branislav Kveton. Solving factored MDPs with continuous and discrete variables. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pages 235–242, 2004.

[Haddawy and Hanks, 1998] Peter Haddawy and Steve Hanks. Utility models for goal-directed decision-theoretic planners. *Computational Intelligence*, 14(3), 1998.

[Hauskrecht and Kveton, 2004] Milos Hauskrecht and Branislav Kveton. Linear program approximations for factored continuous-state Markov decision processes. In *Advances in Neural Information Processing Systems 16*, pages 895–902, 2004.

[Heckerman, 1990] David Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proceedings of the 5th Annual Conference on Uncertainty in Artificial Intelligence (UAI-90)*, New York, NY, 1990. Elsevier Science.

[Hoey *et al.*, 1999] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *UAI-99*, pages 279–288, Stockholm, 1999.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intellence Research JAIR*, 14:253–302, 2001.

[Hölldobler and Skvortsova, 2004] Steffen Hölldobler and Olga Skvortsova. A logic-based approach to dynamic programming. In *In AAAI-04 Workshop on Learning and Planning in Markov Processes–Advances and Challenges*, pages 31–36. AAAI Press, Menlo Park, California, 2004.

[Howard and Matheson, 1984] Ronald A. Howard and James E. Matheson. Influence diagrams. In Ronald A. Howard and James E. Matheson, editors, *Readings on the Principles and Applications of Decision Analysis*. Strategic Decision Group, Menlo Park, CA, 1984.

[Howard, 1960] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[Jaeger, 2000] Manfred Jaeger. On the complexity of inference about probabilistic relational models. *Artificial Intelligence*, 117(2):297–308, 2000.

[Karabaev and Skvortsova, 2005] Eldar Karabaev and Olga Skvortsova. A heuristic search algorithm for solving first-order MDPs. In *UAI-2005*, pages 292–299, Edinburgh, Scotland, 2005.

[Keeney and Raiffa, 1976] R.L. Keeney and H. Raiffa. *Decisions with multiple objectives: Preferences and value tradeoffs*. J. Wiley, New York, 1976.

[Kersting *et al.*, 2004] Kristian Kersting, Martijn van Otterlo, and Luc de Raedt. Bellman goes relational. In *ICML-04*. ACM Press, 2004.

[Kjaerulff, 1990] U. Kjaerulff. Triangulation of graphs–algorithms giving small total state space. Technical Report Research Report R-90-09, Aalborg University, 1990.

[Koehler, 1976] G. J. Koehler. A case for relaxation methods in large scale linear programming. In *Large Scale Systems Theory and Applications*, pages 293–302, Pittsburgh, PA, 1976.

[Koller and Parr, 1999a] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *IJCAI-99*, pages 1332–1339, Stockholm, 1999.

[Koller and Parr, 1999b] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *UAI-2000*, pages 1332–1339, Stockholm, 1999.

[Kushmerick *et al.*, 1995] Neil Kushmerick, Steve Hanks, and Dan Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(12):239–286, 1995.

[Levesque and Brachman, 1987] Hector J. Levesque and Ronald J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93, 1987.

[Levesque *et al.*, 1997] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: a logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[Little, 2006] Iain Little. Paragraph: A Graphplan-based probabilistic planner. In *Proceedings of the Fifth International Planning Competition*, 2006.

[Littman and Younes, 2004a] Michael L. Littman and Hakan L. S. Younes. Introduction to the probabilistic planning track. In *Online Proceedings for The Probablistic Planning Track of IPC-04: http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*, 2004.

[Littman and Younes, 2004b] Michael L. Littman and Hakan L. S. Younes, editors. *Online Proceedings for The Probablistic Planning Track of IPC-04: http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/proceedings/*, Vancouver, Canada, 2004.

[Littman *et al.*, 1998] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *JAIR*, 9:1–36, 1998.

[Littman, 1997] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *AAAI-97*, pages 748–754, Providence, RI, 1997.

[Mahadevan, 2005] Sridhar Mahadevan. Samuel meets Amarel: Automating value function approximation using global state space analysis. In *AAAI-05*, pages 1000–1005, Pittsburgh, 2005.

[Marthi, 2007] Bhaskara Marthi. Automatic shaping and decomposition of reward functions. In *24th International Conference on Machine Learning (ICML)*, Portland, OR, 2007.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pages 410-417.

[McDermott *et al.*, 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Dan Weld, and David Wilkins. PDDL—The planning domain definition language, 1998.

[Meuleau *et al.*, 1998a] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *AAAI-98*, pages 165–172, Madison, WI, 1998.

[Meuleau *et al.*, 1998b] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *AAAI-98*, pages 165–172, Madison, WI, 1998.

[Motik, 2006] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[Ng *et al.*, 1999] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*, pages 278–287. Morgan Kaufmann, San Francisco, CA, 1999.

[Parr and Russell, 1998] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In M. Kearns M. Jordan and S. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, Cambridge, 1998.

[Pearl, 1986] J Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288, 1986.

[Pednault, 1989] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *KR*, pages 324–332, 1989.

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

[Poole, 2003] David Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.

[Poupart *et al.*, 2002a] Pascal Poupart, Craig Boutilier, Relu Patrascu, and Dale Schuurmans. Piecewise linear value function approximation for factored MDPs. In *AAAI-02*, pages 292–299, Edmonton, 2002.

[Poupart *et al.*, 2002b] Pascal Poupart, Relu Patrascu, Dale Schuurmans, Craig Boutilier, and Carlos Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *AAAI-02*, pages 285–291, Edmonton, 2002.

[Puterman and Shin, 1978] Martin L. Puterman and M.C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137, 1978.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[R. Drechsler *et al.*, 1997] R. Drechsler, B. Becker, and S. Ruppertz. Manipulation algorithms for K*BMDs. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 4–18, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

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

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

[Riazanov and Voronkov, 2002] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2):91–110, 2002.

[Rintanen, 2003] Jussi Rintanen. Expressive equivalence of formalisms for planning with sensing. In *ICAPS*, pages 185–194, 2003.

[Rossi *et al.*, 1990] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *ECAI-90: Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, 1990.

[Rudell, 1993] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *(ICCAD)*, 1993.

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

[Sanner and Boutilier, 2006] Scott Sanner and Craig Boutilier. Practical linear evaluation techniques for first-order MDPs. In *UAI-2006*, Boston, Mass., 2006.

[Sanner and Boutilier, 2007] Scott Sanner and Craig Boutilier. Approximate solution techniques for factored first-order MDPs. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, 2007.

[Sanner and McAllester, 2005] Scott Sanner and David McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *IJCAI 2005*, 2005.

[Schuurmans and Patrascu, 2001] Dale Schuurmans and Relu Patrascu. Direct value approximation for factored MDPs. In *NIPS-2001*, pages 1579–1586, Vancouver, 2001.

[Schweitzer and Seidmann, 1985] Paul Schweitzer and Abraham Seidmann. Generalized polynomial approximations in markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110:568–582, 1985.

[Singh and Cohn, 1998] Satinder P. Singh and David Cohn. How to dynamically merge Markov decision processes. In *NIPS-98*, pages 1057–1063. MIT Press, Cambridge, 1998.

[Soutchanski, 2001] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *IJCAI-2001*, Seattle, Washington, 2001.

[St-Aubin *et al.*, 2000] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *NIPS-2000*, pages 1089–1095, Denver, 2000.

[Stergiou and Walsh, 1999] Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI-99*, pages 163–168, Orlando, FL., July 1999.

[Tafertshofer and Pedram, 1997] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Form. Methods Syst. Des.*, 10(2-3), 1997.

[Teichteil and Fabiani, 2006] Florent Teichteil and Patrick Fabiani. Symbolic stochastic focused dynamic programming with decision diagrams. In *Proceedings of the Fifth International Planning Competition*, 2006.

[Trick and Zin, 1997] Michael A. Trick and Stanley E. Zin. Spline approximations to value functions: A linear programming approach. In *Macroeconomic Dynamics*, volume 1, pages 255–277, 1997.

[Tsitsiklis and Van Roy, 1996] John N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[Veloso, 1992] Manuela Veloso. *Learning by analogical reasoning in general problem solving*. PhD thesis, Carnegie Mellon University, August 1992.

[Wang and Khardon, 2007] Chenggang Wang and Roni Khardon. Policy iteration for relational MDPs. In *UAI*, Vancouver, Canada, 2007.

[Wang *et al.*, 2007] Chenggang Wang, Saket Joshi, and Roni Khardon. First order decision diagrams for relational MDPs. In *IJCAI*, Hyderabad, India, 2007.

[Weld, 1999] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[Wiewiora, 2003] Eric Wiewiora. Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.

[Williams and Baird, 1994] Ronald Williams and Leeman Baird. Tight performance bounds on greedy policies based on imperfect value functions. In *Eighth Yale Workshop on Adaptive and Learning Systems*, pages 108–113, New Haven, CT, 1994.

[Williamson and Hanks, 1994] Michael Williamson and Steve Hanks. Utility-directed planning. In *Artificial Intelligence and Planning Systems*, 1994.

[Yianilos, 1993] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[Yoon *et al.*, 2002] SungWook Yoon, Alan Fern, and Robert Givan. Inductive policy selection for first-order Markov decision processes. In *UAI-02*, Edmonton, 2002.

[Yoon *et al.*, 2004] SungWook Yoon, Alan Fern, and Robert Givan. 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/*, 2004.

[Yoon *et al.*, 2007] Sungwook Yoon, Alan Fern, and Robert Givan. Ff-replan: A baseline for probabilistic planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, 2007.

[Younes and Littman, 2004] Hakan Younes and Michael Littman. PPDDL: The probabilistic planning domain definition language: *http://www.cs.cmu.edu/~lorens/papers/ppddl.pdf*, 2004.

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

[Zhang and Poole, 1996] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *J. Artif. Intell. Res. (JAIR)*, 5:301–328, 1996.