First Order Markov Decision Processes

A Dissertation

submitted by

Chenggang Wang

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

May 2007

ADVISOR: Professor Roni Khardon

Abstract

Relational Markov Decision Processes (RMDP) are a useful abstraction for complex reinforcement learning problems and stochastic planning problems since one can develop abstract solutions for them that are independent of domain size or instantiation. This thesis develops compact representations for RMDPs and exact solution methods for RMDPs using such representations. One of the core contributions of the thesis is development of the First Order Decision Diagram (FODD), a representation that captures functions over relational structures, together with a set of operators to manipulate FODDs. FODDs offer a potentially compact representation for complex functions over relational structures and can therefore serve as underlying engine for efficient algorithms with relational structures. The second core contribution is developing exact solution methods for RMDPs based on FODD representations. In particular FODDs are used to represent value functions, transition probabilities, and domain dynamics of RMDPs. Special operations are developed to implement exact value iteration and a novel variant of policy iteration and the algorithms are shown to calculate optimal solutions for RMDPs. Finally we show how the algorithms for RMDPs using FODDs can be extended to handle relational Partially Observable MDPs.

Acknowledgments

I would like to thank my advisor, Professor Roni Khardon, for giving me the opportunity to complete my Ph.D. This thesis would never have been possible without his guidance, insight, and encouragement. He has been a great example and inspiration of how much one can accomplish: a devoted parent, an excellent researcher, and a great teacher and advisor.

I would like to thank my former advisor, the late Professor Jim Schmolze, for his incredible understanding, support, and wise advice, during a very difficult time in my life when my daughter had medical issues. Without his help and belief in me, I would not have been able to continue with my study.

I would like to thank Professor Souvaine for her support and encouragement during all these years. I would also like to thank the members of my committee, Professors Anselm Blumer, Carla Brodley, Sridhar Mahadevan, and Eric Miller, for their valuable comments and feedback.

I would like to thank the staff at the Tufts Computer Science Department. George Preble, in particular, has been very helpful.

I would like to thank Saket Joshi for many profitable and interesting discussions. We worked together on First Order Decision Diagrams and value iteration for relational MDPs.

Finally, I would like to thank my parents for their unbounded love and everlasting belief in me, and my husband, Bin Yu, for his love, support, and wisdom.

DEDICATION

To my parents.

Contents

List of Figures

First Order Markov Decision Processes

Chapter 1

Introduction

Many real-world problems can be cast as sequential decision making under uncertainty. Consider the simple example where an agent delivers boxes. The agent can take three types of actions: to load a box on a truck, to unload a box from a truck, and to drive a truck to a city. However the action effects may not be perfectly predictable and may be to some extent stochastic. For example its gripper may be slippery so load actions may not succeed, or its navigation module is not reliable and it may end up in a wrong location. This uncertainty compounds the already complex problem of planning a course of action to achieve some goals or get rewards.

Markov Decision Processes (MDP) have become the de facto standard models for sequential decision making problems (Boutilier et al., 1999). These models also provide a general framework for artificial intelligence (AI) planning, where an agent has to achieve or maintain a well-defined goal. MDP models an agent interacting with the world. The agent can observe the state of the world, and takes actions to change the state of the world; and the agent tries to optimize the rewards it obtains in the world.

We can specify the possible states in the world in terms of a set of propositional variables called state attributes, that together determine the world state. Consider a very simple logistics problem that has only one box, one truck, and one destination, which is Paris, then we can have state attributes such as truck in Paris (TP), box in Paris (BP), box in Boston (BB), etc. If we let the state space be represented by n binary state attributes then the total number of states would be 2^n. Thus any algorithm that works by state enumeration will require time exponential in n. Indeed, it is well known that the standard representation and algorithms for MDPs that enumerates all the states in the world does not scale to solve realistic planning domains.

However we can do better than this. There has been previous work called the propositionally factored approach which directly operates on state attributes instead of enumerating states. By specifying a succinct MDP model and developing algorithms that exploit MDP structure this approach can solve large MDP problems.

But what if we have a more complicated problem? What if now we have four trucks, three boxes, and the goal is that there is a box in Paris, but it does not matter which box is in Paris? With the propositionally factored approach, we need to have one propositional variable for every possible instantiation of the properties and relations in the domain, e.g., box 1 in Paris, box 2 in Paris, box 1 on truck 1, box 2 on truck 1, etc. The goal becomes box 1 in Paris or box 2 in Paris or box 3 in Paris or box 4 in Paris. And there will be a huge number of possible actions, such as to load box 1 on truck 1, to load box 1 on truck 2, etc. The problem becomes huge. We also lose the structure and lose the benefits in terms of computation.

This is why first order or relational MDPs (RMDP) are an attractive approach. With the first order representations of MDPs, we can describe domain objects and relations among them, and use quantification in specifying objectives. In the logistics example, we can introduce three predicates to capture the relations among domain objects, i.e., $Bin(Box, City)$, $Tin(Truck, City)$, and $On(Box, Truck)$ with their obvious meaning. We have three parameterized actions, i.e., $load(Box, Truck)$, $unload(Box, Truck)$, and $drive(Truck, City)$. We can define the goal using existential quantification, i.e., $\exists b, Bin(b, Paris)$.

The relational approach performs general reasoning and never needs to propositionalize a domain. As a result the complexity does not change when the number of domain objects changes. Also the solutions obtained are good for any domain size (even infinite ones) simultaneously. Such an abstraction is not possible within the propositional approach.

Now an obvious question is how to represent and solve relational MDPs efficiently and that is the main issue addressed in this thesis. The thesis develops compact representations and efficient solution methods that utilize such representations for RMDPs. In terms of representation, we develop First Order Decision Diagrams (FODD) to handle the relational structure and define a set of reduction operators to minimize the representation size when working with such diagrams. In terms of solution methods, we show how value iteration can be performed using FODDs. We also develop a new algorithm, Relational Modified Policy Iteration, which incorporates an aspect of policy improvement into policy evaluation. We show that the algorithm converges to the optimal value function and policy. We further extend our work to relational Partially Observable MDPs (RPOMDP), where the process is "partially" observable if the agent's knowledge about the current state of world is not complete.

1.1 Background and Motivation

In the past years there has been an increased interest in developing RMDPs. Some examples include symbolic dynamic programming (SDP) (Boutilier et al., 2001), the relational Bellman algorithm (ReBel) (Kersting et al., 2004), first order value iteration (FOVIA) (Großmann et al., 2002; Hölldobler & Skvortsova, 2004), approximate linear programming for RMDPs (Guestrin et al., 2003a; Sanner & Boutilier, 2005; Sanner & Boutilier, 2006), Envelop-based planning (Gardiol & Kaelbling, 2003), approximate policy iteration (Fern et al., 2003), inductive policy selection using first order regression (Gretton & Thiebaux, 2004), and first order machine learning (Mausam & Weld, 2003). These methods show how classical MDP solution methods can be "upgraded" to RMDPs in some cases introducing novel techniques to approximate RMDP solutions.

Among these, only SDP, ReBel, and FOVIA are exact solution methods. To our knowledge there is no working implementation of SDP because it is hard to keep the state formulas consistent and of manageable size in the context of situation calculus. Compared with SDP, ReBel and FOVIA provide a more practical solution. They both use restricted languages to represent RMDPs, so that reasoning over formulas is easier to perform. We are interested in developing a representation that combines the strong points of these approaches and supports efficiently exact solution methods.

It is interesting that all exact solution methods listed above use the value iteration algorithm (Puterman, 1994). Other algorithms and in particular Policy Iteration (Puterman, 1994) may have obvious advantages, especially in cases where the abstract value function requires an infinite number of state partitions, but there is a simple optimal policy for the domain. Recall that a value function assigns each state a value that captures the "long-term reward" expected when starting in this state. An abstract value function assigns values to sets of states or "abstract states" so actual states do not need to be enumerated. To illustrate the property above, consider the blocks world domain and a simple goal that there is no block on block a, i.e. $Clear(a)$. This requires an infinite number of state partitions in the optimal value function (Kersting et al., 2004) because we do not know how many blocks are on top of the stack of a and this number determines the value expected. However the optimal policy, stating that any block which is above a must be moved, is trivial and can be expressed as follows:

$\exists x, Above(x,a) \wedge Clear(x) \rightarrow MoveToTable(x)$

otherwise, do nothing (no-op).

Therefore in some cases a value function requires infinite size to represent and thus converging to the optimal value function is problematic. However finding a compact policy may be possible, so policy iteration may be better than value iteration in such cases. Therefore we are interested in developing and analyzing relational variants of both value iteration and policy iteration.

Partially Observable Markov Decision Processes (POMDP) provide a more general framework for sequential decision making problems than MDPs. MDPs assume that the underlying state of the process is known with certainty. This assumption is too strong for many practical planning problems, e.g., in medical diagnosis, the exact internal state of the patient is not completely known, yet the doctor needs to decide on a course of action. POMDPs generalizes MDPs by allowing for imperfect or incomplete information about the underlying state of the process. Each POMDP action has two effects: changing the world and gathering information about the world. In recent years, researchers have sought to combine the benefits of logic with the power of POMDPs (e.g., (Geffner & Bonet, 1998; Poole, 1997; Boutilier & Poole, 1996; Bacchus et al., 1999; Hansen & Feng, 2000)). However, there has not been much research on relational or first order POMDPs (RPOMDP). Boutilier and Poole (1996) and Hansen and Feng (2000) only develop solution methods for propositionally factored POMDPs. Poole (1997) and Bacchus et al. (1999) only focus on first order representations and do not build planners. Geffner and Bonet (1998) compile the first order modeling language into the underlying POMDP with flat states, therefore their approach can not provide general solutions. We are interested in extending our work to POMDPs and developing relational RPOMDPs.

1.2 Our Approach and Contributions

Inspired by the successful application of Algebraic Decision Diagrams (ADD) (Bryant, 1986; McMillan, 1993; Bahar et al., 1993) in solving propositionally factored MDPs and POMDPs (Hoey et al., 1999; St-Aubin et al., 2000; Hansen & Feng, 2000; Feng & Hansen, 2002), we develop First Order Decision Diagrams (FODD) by lifting propositional ADDs to handle relational structure. We then use FODDs in the solution of RMDPs. The intuition behind this idea is that the ADD representation allows information sharing, e.g., sharing the value of all states that belong to an "abstract state", so that algorithms can consider many states together and do not need to resort to state enumeration. If there is sufficient regularity in the model, ADDs can be very compact, allowing problems to be represented and solved efficiently.

First order decision trees and even decision diagrams have already been considered in the literature (Blockeel & De Raedt, 1998; Groote & Tveretina, 2003) and several semantics for such diagrams are possible. Blockeel and De Raedt (1998) lift propositional decision trees to handle relational structure in the context of learning from relational datasets. Groote and Tveretina (2003) provide a notation for first order Binary Decision

Diagrams (BDD) that can capture formulas in Skolemized conjunctive normal form and then provide a theorem proving algorithm based on this representation. In this thesis we study the differences between semantics based on each representation and point out that the semantics by Groote and Tveretina (2003) supports value iteration better than the semantics based on first order decision trees (Blockeel & De Raedt, 1998). Therefore we adapt and extend their approach to handle first order MDPs. In particular, we extend the definitions to handle existential quantification and numerical leaves through the use of an aggregation function. This allows us to capture value functions using algebraic diagrams in a natural way. We also provide additional reduction transformations for algebraic diagrams that help keep their size small, and allow the use of background knowledge in reductions. We then develop appropriate representation and algorithms showing how value iteration can be performed using the decision diagrams. In particular, we use truth value diagrams (TVD), a special FODD, to capture domain dynamics, and use FODDs to represent probabilities, rewards, and value functions. We perform regression of the value function, that is we calculate the value of a state before an action from the values of states after an action, using an operation we call "block replacement". In this operation each node in a diagram is replaced with a TVD capturing its regressed value. This offers a modular and efficient form of regression that accounts for all potential effects of an action simultaneously. Figure 1.1 illustrates how to perform block replacement. Each block is a TVD for the corresponding predicate under an action. This diagram is simply shown for illustration. Precise definitions and technical details are given in following chapters. The value function V_1, capturing the optimal value function when there is one step to go, is represented using a FODD as shown on the left hand side of Figure 1.1. This value function partitions the whole state space into the following four abstract states, and assigns a value to each abstract state.

$\exists b, Bin(b, Paris) : 19$

$\neg\exists b, Bin(b, Paris) \wedge \exists b, t, On(b, t) \wedge Tin(t, Paris) \wedge rain : 6.3$

$\neg\exists b, Bin(b, Paris) \wedge \exists b, t, On(b, t) \wedge Tin(t, Paris) \wedge \neg rain : 8.1$

otherwise: 0

Other operations over diagrams require “standardizing apart” (as in logic). Some operations on the graphs of the diagrams are followed by reductions to maintain small size. We show that our version of abstract value iteration is correct and hence it converges to optimal value function and policy.

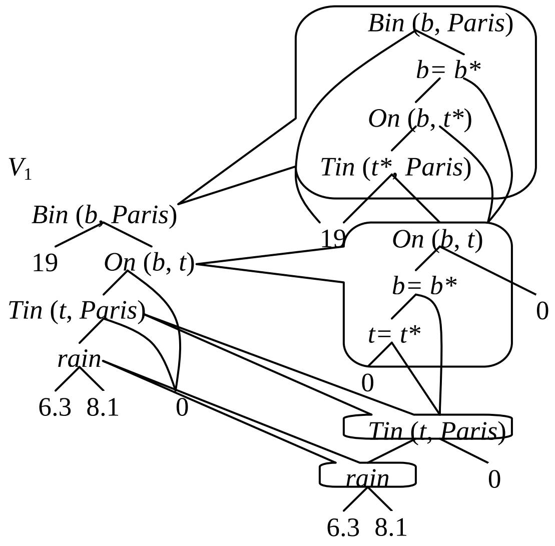


Figure 1.1: An example of block replacement.

We also investigate the potential of developing and analyzing policy iteration for relational domains in the context of FODDs. We introduce a new algorithm, Relational Modified Policy Iteration, that uses special operations with FODDs to mimic the original modified policy iteration algorithm (Puterman, 1994). We point out two anomalies of policy languages in the context of policy evaluation. First, some policy languages in the literature do not have well defined value functions. Second, there is some interaction between the value and policy languages so that, when using a restricted representation scheme, the value function of some natural policies are not expressible in the language. Overcoming this difficulty, our algorithm incorporates an aspect of policy improvement into policy evaluation. We show that the algorithm converges to the optimal value function and policy.

Finally we discuss first steps towards solving RPOMDPs. We use FODDs to model an action’s informational effect, and develop a *sum* statement to represent belief states.

We then show how the value iteration algorithm can be lifted to the relational case. The result is not complete and we discuss several open questions that arise due to subtle issues that do not exist in the propositional case during planning and execution.

It is useful to compare our solutions to the propositional ones. The main difficulty in lifting the ideas from the propositional case (Hoey et al., 1999; St-Aubin et al., 2000) is that in relational domains the transition function specifies a set of schemas for conditional probabilities. The propositional solution uses the concrete conditional probability to calculate the regression function. But this is not possible with schemas. One way around this problem is to first ground the domain and problem at hand and only then perform the reasoning (see for example (Sanghai et al., 2005)). However this does not allow for solutions abstracting over domains and problems. Like SDP, ReBel, and FOVIA, our constructions do perform general reasoning and they do so by using decision diagrams.

In summary, our contributions include the following:

1. We have identified differences between the semantics based on first order decision trees (Blockeel & De Raedt, 1998) and the semantics based on first order BDDs (Groote & Tveretina, 2003) for first order decision diagrams and their relevance for RMDP algorithms.

2. We have developed the technical details of first order ADDs and for the algorithms using them in the solution of relational MDPs. It is non-trivial to lift methods for propositional decision diagrams to the first order case, and our work highlights some of the key semantics and computational issues and proposes solutions. We have also developed novel weak reduction operations for first order decision diagrams and shown their relevance to solving relational MDPs.

3. We have developed relational value iteration for MDPs using FODDs and have shown that it is correct and thus converges to the optimal value function and policy.

4. We have developed and analyzed policy iteration in the relational domain and have shown that the algorithm converges to the optimal value function and policy and that it dominates the iterates from value iteration.

5. We have lifted a value iteration algorithm for propositional POMDPs to handle the relational case. Although the result is not complete, we have made first steps towards solving RPOMDPs, and identified some subtle issues that do not exist in propositional case during planning and execution.

Our contributions are discussed in more detail in Chapter 7.

1.3 Thesis Overview

This introductory chapter has briefly summarized the research background, motivation, and our approach. The remainder of the thesis is organized as follows.

Chapter 2 provides background on standard MDPs, including POMDPs, and their solution methods. It discusses propositionally factored representations and current work on RMDPs focusing on exact solution methods. It also summarizes related work.

Chapter 3 describes our representation framework FODDs. It examines in detail the syntax and semantics of the language. A large part of this chapter is devoted to developing reduction operators that remove redundancies in FODDs. These are not important for correctness of our algorithms but are crucial to keep representation small and make the algorithms feasible.

Chapter 4 describes how to represent RMDPs with FODDs. To facilitate the description it introduces a variant of the "logistics problem" as an example domain. It also discusses the expressiveness of the representation and gives a procedure of translating into FODDs problem descriptions in PPDDL (Younes & Littman, 2004), which has become a standard language used to encode probabilistic planning domains and problems.

Chapter 5 presents our algorithms for RMDPs, relational value iteration and relational modified policy iteration, and investigates properties of policy language that arise in context of policy iteration. It shows that relational value iteration is correct and hence converges. For relational modified policy iteration, the algorithm incorporates an aspect of policy improvement into policy evaluation, but it still converges to the optimal value function and policy.

Chapter 6 describes how the algorithms for RMDPs using FODDs can be extended to handle Relational POMDPs. In particular it shows how the incremental pruning algorithm (Cassandra et al., 1997) that implements value iteration can be lifted to the relational case. It points out some subtle issues that do not exist in the propositional case during planning and execution, and raises some open questions.

Finally, Chapter 7 summarizes the contributions of this work and points out some directions for future research.

Chapter 2

Markov Decision Processes

In this chapter we review standard notions and commonly used solution methods in Markov Decision Processes, including Partially Observable Markov Decision Processes. We also introduce previous work on handling large state spaces. For a comprehensive introduction to fully observable MDPs please refer to (Puterman, 1994).

2.1 Fully Observable Markov Decision Processes

Fully Observable Markov Decision Processes (MDPs) are mathematical models of sequential optimization problems with stochastic actions where the agent has full knowledge of the state of the world. A MDP can be characterized by a state space S, an action space A, a state transition function $Pr(s_j|s_i, a)$ denoting the probability of transition to state s_j given state s_i and action a, and an immediate reward function $r(s)$, specifying the immediate utility of being in state s.

A policy π for a MDP is a mapping from S to A. It is associated with a value function $V^\pi : S \rightarrow \Re$, where $V^\pi(s)$ is the expected discounted total reward that the agent gets if it starts at state s. V^π can be computed exactly by solving the following set of linear equations with one equation for each state:

$$V^\pi(s) = r(s) + \gamma \sum_{s' \in S} Pr(s'|s, \pi(s))V^\pi(s') \tag{2.1}$$

A solution to a MDP is an optimal policy that maximizes expected discounted total reward as defined by the Bellman equation.

$$V^*(s) = max_{a \in A}[r(s) + \gamma \sum_{s' \in S} Pr(s'|s,a)V^*(s')]$$

where V^* represents the optimal state-value function.

A class of algorithms, such as value iteration, policy iteration, linear programming, and heuristic search, can be used to find optimal policies.

2.1.1 Value Iteration

The value iteration algorithm (VI) uses the Bellman equation to iteratively refine an estimate of the optimal value function

$$V_{n+1}(s) = max_{a \in A}[r(s) + \gamma \sum_{s' \in S} Pr(s'|s,a)V_n(s')] \tag{2.2}$$

where $V_n(s)$ represents our current estimate of the value function and can be considered as the optimal value function when an agent has n steps to go, and $V_{n+1}(s)$ is the next estimate and can be considered as the optimal value function when an agent has $n+1$ steps to go. Value iteration starts with the reward function, which is the optimal value function when an agent has 0 step to go, and repeats the update until $\|V_{n+1}(s) - V_n(s)\| \leq \frac{\varepsilon(1-\gamma)}{2\gamma}$, which guarantees that $\|V_{n+1}(s) - V^*(s)\| \leq \varepsilon$, a condition known as ε-optimality. The set of actions $\pi_{n+1}(s) = argmax_{a \in A}[r(s) + \gamma \sum_{s' \in S} Pr(s'|s,a)V_n(s')]$ are called the greedy policy with respect to the value function V_n. Since we can calculate V_{n+1} and π_{n+1} at the same time, we introduce the notation $(V_{n+1}, \pi_{n+1}) = \text{greedy}(V_n)$.

Let Q_V^a denote the value obtained by performing action a for one step and receiving the terminal value V. We can rewrite Equation 2.2 in the following two steps:

1. Regression:

   $Q_{V_n}^a = r(s) + \gamma \sum_{s' \in S} Pr(s'|s,a)V_n(s')$.

2. Maximization:

$V_{n+1}(s) = max_{a \in A} Q^a_{V_n}(s)$.

2.1.2 Policy Iteration

Fixing a policy π in Equation (2.2) (instead of maximizing over actions), and replacing V_n with a generic value function V we obtain

$$Q^\pi_V(s) = r(s) + \gamma \sum_{s' \in S} Pr(s'|s, \pi(s)) V(s')$$

so that Q^π_V is the value obtained by executing policy π for one step and then receiving the terminal value V. Note that $Q^\pi_V(s) = Q^a_V(s)$ where $\pi(s) = a$. When V is V_n and the output is taken as V_{n+1} we get the successive approximation algorithm calculating the value function V^π (Puterman, 1994). We later distinguish the algorithmic part from the Q value calculated in order to check whether they are the same in the relational case. We therefore denote the algorithm calculating Q from V and π by regress-policy(V, π).

Policy iteration (PI) is an alternative algorithm to VI that can be faster in some cases (Puterman, 1994). PI starts with any policy π and repeats the following steps until the policy does not change.

1. Policy evaluation: compute V^π by solving Equation 2.1 or through successive approximation.

2. Policy improvement: $\pi =$ greedy(V^π).

The final policy is the optimal policy and the value function corresponding to this policy is the optimal value function.

For large state spaces it may be computationally prohibitive to evaluate the policy exactly or use successive approximation until convergence. It has been observed that the exact value of the current policy is not needed for policy improvement and an approximation based on a few number of iterations of successive approximation is often sufficient (Puterman, 1994). Therefore, Puterman (1994) introduced Modified Policy Iteration (MPI) where the sequence m_n of non-negative integers controls the number of updates in policy evaluation steps:

Procedure 1 *Modified Policy Iteration*

```
1. $n = 0$, $V_0 = R$.
2. Repeat
    (a) (Policy improvement)
        $(w^0_{n+1}, \pi) = greedy(V_n)$.
    (b) If $\|w^0_{n+1} - V_n\| \leq \epsilon(1-\gamma)/2\gamma$, return $V_n$ and $\pi$, else go to step 2c.
    (c) (Partial policy evaluation)
        k=0.
        while $k < m_{n+1}$ do
            i. $w^{k+1}_{n+1} = regress\text{-}policy(w^k_{n+1}, \pi)$.
            ii. k=k+1.
    (d) $V_{n+1} = w^{m_{n+1}}_{n+1}$, $n = n + 1$.
```

Note that this algorithm uses the same stopping criterion as that of value iteration, and the returned policy is ϵ-optimal. Puterman (1994) has shown that the algorithm converges for any m_n sequence.

A MDP can also be solved by formulating it as a linear program (LP). For details see (Puterman, 1994).

2.1.3 Heuristic Search

If the initial state is known and we have a finite-horizon MDP, i.e., the planning problem proceeds for a finite number of stages, we can solve the MDP using forward search. We can construct a stochastic decision tree rooted at the initial state by expanding on actions up to a certain depth and then computing expectations and maximizations as we "fold back" the tree, where we first determine values at leaves and then proceed to calculate values at successive levels of the tree. Figure 2.1 illustrates this process. A brute-force forward search would require time exponential in the horizon to calculate the expected value for a single state and cannot solve the infinite-horizon problem. However heuristic search algorithms that focus on a partial search space can do much better and can handle infinite horizon problems.

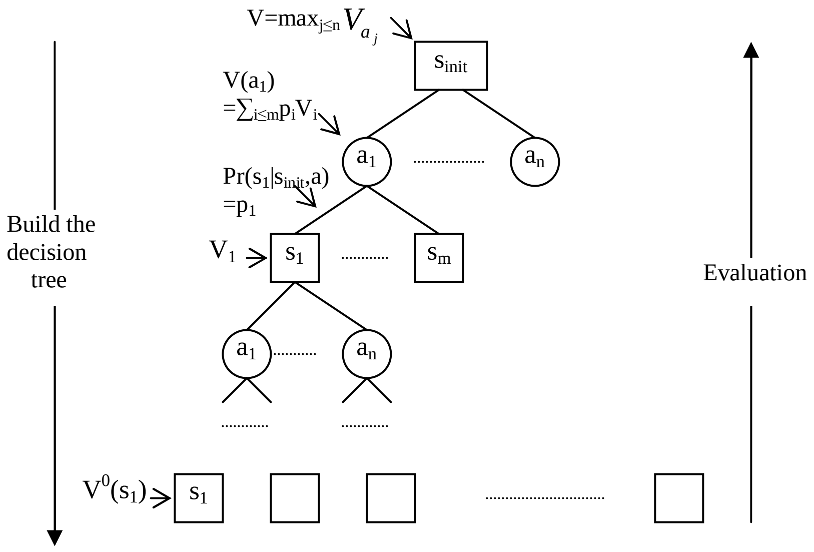


Figure 2.1: Decision tree building and evaluation.

Note that the decision tree is an AND-OR tree — the actions form the OR branches because the optimal action for a state is a choice among a set of actions; and a set of possible next states form the AND branches for an action because the utility of an action is a sum of the utility of each next state multiplied by the probability of transition to that state. The heuristic search algorithm AO* (Nilsson, 1971; Martelli & Montanari, 1973) is often used to find optimal solutions for problems formalized as AND-OR trees or graphs. However, AO* can only take an acyclic graph as a solution and thus unable to handle solution with loops, which are typical for MDPs, e.g., a policy that keeps trying a stochastic action until it succeeds. LAO* (Hansen, 2001) is an extension of AO*, in which a solution is represented as a finite state controller. In contrast to AO*, which updates state values in a single backward pass from the states on the fringe of best partial solution to the start state, LAO* relies on a dynamic programming algorithm such as value iteration or policy iteration to perform such update.

Real time dynamic programming (RTDP) (Barto et al., 1995) is another method to avoid search in the entire state space. RTDP is a probabilistic extension of a heuristic search algorithm Learning Real-Time A* (LRTA*) (Korf, 1990). Trial-based RTDP starts with an admissible heuristic function (i.e., non-overestimating). Each trial begins with the initial state and repeats the following steps until a goal is reached or a stopping criterion is met: first, for the current state an action is greedily selected based on the heuristic function; then the current state is updated based on the outcome of the action. This updates the heuristic function for next iteration. If the heuristic function is non-overestimating and there is a path (with positive probability) from the initial state to the goal, the algorithm converges to an optimal solution (Barto et al., 1995).

2.2 Propositionally Factored MDPs

Value iteration and policy iteration for MDPs require time polynomial in $|S|$ and $|A|$ for each iteration. However $|S|$ can be very large. If we let the state space be represented by n binary state attributes $\mathbf{X} = X_1, \ldots, X_n$, then the total number of states would be 2^n. This is the well-known "curse of dimensionality", implying that algorithms that enumerate states run in exponential time in the number of state attributes. Consequently efficient representations that explicitly model structure in MDP problems, and algorithms that utilize such structure are extremely important in order to solve problems with large state spaces.

Several representations for propositionally factored MDPs, where a state is represented by a set of attributes, have been proposed. These representations can be much more concise than the standard representation. Dynamic Bayesian Networks (DBN) (Dean & Kanazawa, 1989) are most widely used factored representation of MDPs. Let $\mathbf{X} = \{X_1, \cdots, X_n\}$ represent the state attributes at the current time t and $\mathbf{X}' = \{X'_1, \cdots, X'_n\}$ represent the state attributes in the next step $t + 1$. Figure 2.2 shows a DBN involving two stages (also referred to as two-stage temporal Bayesian network (2TBN)). Each edge between two state attributes in the DBN indicates a direct probabilistic dependency between them, therefore the DBN explicitly captures independence among state attributes. Notice that we have one DBN for each action a.

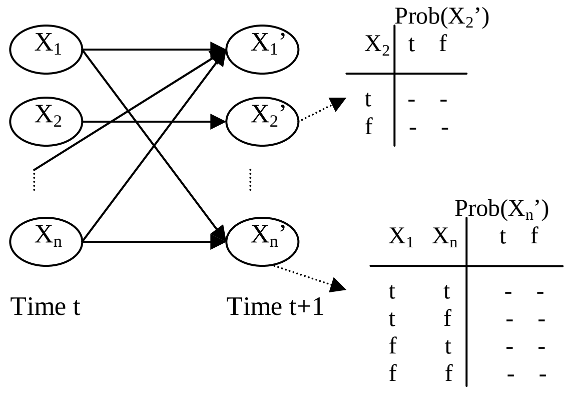


Figure 2.2: An example illustrating the DBN.

With the DBN representation, we can define the state transition function as follows:

$P(X_1', X_2', \cdots, X_n'|X_1, X_2, \cdots, X_n, a) = P(X_1'|Pa(X_1'), a) \times P(X_2'|Pa(X_2'), a) \cdots \times P(X_n'|Pa(X_n'), a)$

where $Pa(X_i')$ refers to the parent nodes of X_i' in the DBN. This formula is true in general, because it is using all the parents of a node. But it is most convenient when all parents are in time slice t since then the action's effect on each variable is independent given the previous state.

Let m be the maximum number of variables a variable can depend on, then we need to specify $n2^m$ parameters with the DBN representation. Since in many domains each state attribute only depends on a few others, this factored representation of state transition functions may require much fewer parameters than a full state-based transition matrix, which requires 2^{2n} parameters. We can specify the conditional probability more compactly for each successor state attribute using decision trees or algebraic decision diagrams (ADD) because of a property called context-specific independence, i.e., certain values of variables in a conditional probability table (CPT) make other values irrelevant (Boutilier et al., 1996). As shown by Boutilier et al. (2000) and Hoey et al. (1999), we can further use decision trees or ADDs to compactly represent rewards, value functions, and policies.

There are alternative representations to the DBN representation such as probabilistic STRIPS-like language (PStrips) (Kushmerick et al., 1995). Littman (1997) proved that the two representations are representationally equivalent if one is allowed to change domain encoding. However each has its own strength that makes each more suitable to represent a certain class of problems. The DBN representation is better at describing the effects of actions on distinct variables, while PStrips is more effective for representing actions with correlated effects (Boutilier et al., 1999).

Now that we have compact representations for MDPs, the next challenge is to develop algorithms that can exploit such factored representations. Boutilier et al. (2000) and Hoey et al. (1999) have shown that if we can represent each of $r(s)$, $Pr(s'|s,a)$, and $V_k(s)$ compactly using a decision tree or an algebraic decision diagram (ADD) then value iteration and policy iteration can be done directly using these representations, avoiding the need to enumerate the state space. In general ADDs provide a more compact representation than decision trees. Empirical studies by Hoey et al. (1999) showed substantial speedup for propositionally factored domains and that ADD representation outperforms the algorithm using the decision tree representation. Feng and Hansen (2002) further generalize the LAO* algorithm to solve factored MDPs represented using ADDs. The ADD representation and first order generalizations for them are discussed in detail in Chapter 4.

2.3 Relational or First Order MDPs

Propositional solutions, obtained by grounding a domain instance, do not provide abstract solutions that are independent of domain size or instantiation and they get slower with increasing problem size. There has been an increasing interest in developing relational or first order representations of MDPs because one can develop abstract solutions for them.

While "DBNs+ADDs" and "DBNs+decision trees" approaches have been successfully used for efficient representations and computations in propositionally factored MDPs, it is not straightforward to use similar techniques in first-order MDPs. One difficulty lies in lifting the propositional DBN to the first order case without having to propositionalize the domain. While the relational or first-order DBN (RDBN) can specify the dependencies between different predicates, it is not clear how it can capture the correlation between different instances of the same predicate. For example in the blocks world, the action $move(x, y, z)$, which means moving block x to y from z, causes $on(x, y)$ and $\neg on(x, z)$ to happen simultaneously. They are either simultaneously true or simultaneously false. But the RDBN will make $+on(x, y)$ and $-on(x, z)$ conditionally independent whatever the preconditions are. Mausam and Weld (2003) and Sanghai et al. (2005) defined a version of RDBNs that handled such cases by having multiple DBN nodes for the same predicate (with different arguments). As a result, they perform inference at the propositional level.

On the other hand, we can easily use PStrips to represent the following action:

$Move(x, y, z)$

Precondition: $on(x, z)$, $clear(x)$, $clear(y)$

Effects:0.1 nothing

0.9 add_list $clear(z)$, $on(x, y)$

del_list $on(x, z)$, $clear(y)$

where for any object x, y, z, if the precondition $on(x, z) \wedge clear(x) \wedge clear(y)$ holds, then with probability 0.9 $on(x, z) \wedge clear(y)$ will be false in next state and $clear(z) \wedge on(x, y)$ will be true. With probability 0.1 the next state will be identical to the current state.

First-order MDPs were first introduced by Boutilier et al. (2001). We call their approach *SDP* for Symbolic Dynamic Programming. Their work was developed in the context of the situation calculus. One of the useful restrictions introduced in their work is that stochastic actions must be specified as a randomized choice among deterministic alternatives. For example, action *load* in the logistics example can succeed or fail. Therefore there are two alternatives for this action: *loadS* (load success) and *loadF* (load failure). The formulation and algorithms support any number of action alternatives. The randomness in the domain is captured by a random choice specifying which action alternative gets executed following some specified probability. In this way one can separate the regression over action effects, which is now deterministic, from the probabilistic choice of action. The same style of specification is used in the PStrips example given above where choice of "effects-version" is probabilistic but each effect is deterministic.

Boutilier et al. (2001) introduce the case notation to represent probabilities and rewards compactly. The expression $t = case[\phi_1, t_1; \cdots; \phi_n, t_n]$, where ϕ_i is a logic formula, is equivalent to $(\phi_1 \wedge t = t_1) \vee \cdots \vee (\phi_n \wedge t = t_n)$. In other words, t equals t_i when ϕ_i is true. The ϕ_i's are disjoint and partition the state space. The t_i's are constants. Each ϕ_i denotes an abstract state whose member states have the same value for that probability or reward.

On each regression step during value iteration, the value of a stochastic action $A(\vec{x})$ parameterized with free variables $\vec{x}$ is determined in the following manner:

$$Q_V^{A(\vec{x})}(s) = rCase(s) \oplus \gamma[\oplus_j pCase(n_j(\vec{x}), s) \otimes Regr(vCase(do(n_j(\vec{x}), s)))]$$

where $rCase(s)$ and $vCase(s)$ denote reward and value functions in case notation, $n_j(\vec{x})$ denotes the possible outcomes of the action $A(\vec{x})$, and $pCase(n_j(\vec{x}), s)$ the choice probabilities for $n_j(\vec{x})$. Operators $\oplus$ and $\otimes$ are defined in the following way:

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

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

Note that we can replace a sum over s' in the standard value iteration with a sum over j, the action alternatives, since different next states arise only through different action alternatives. *Regr* is the same as classical goal regression, which determines what states one must be in before an action in order to reach a particular state after the action. Figure 2.3 illustrates the regression of $\exists b, Bin(b, Paris)$ in the reward function R through the action alternative $unloadS(b^*, t^*)$. $\exists b, Bin(b, Paris)$ will be true after the action $unloadS(b^*, t^*)$ if it was true before or box b^* was on truck t^* and truck t^* was in Paris. Notice how the reward function R partitions the state space into two regions or abstract states, each of which may include an infinite number of complete world states (e.g., when we have infinite number of domain objects). Also notice how we get another set of abstract states after the regression step, which ensures that we can work on abstract states and never need to propositionalize the domain.

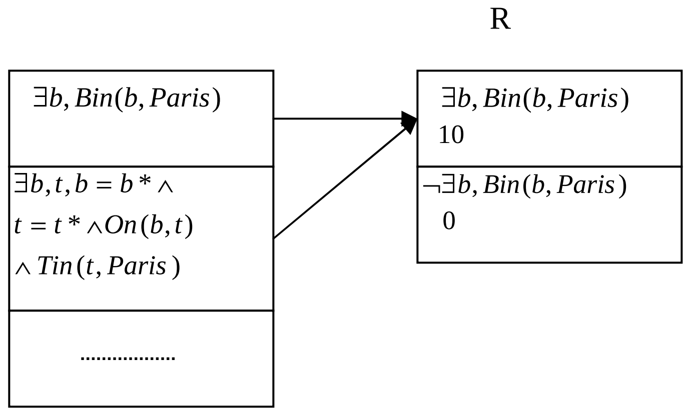


Figure 2.3: An example illustrating regression over an action alternative.

After the regression, we get a parameterized Q-function which accounts for all possible versions of an action. We need to maximize over the action parameters of each Q-function to get the maximum value that could be achieved by using an instance of this action. Consider the logistics example where we have two boxes b_1 and b_2, and b_1 is on truck t_1 which is in Paris ($On(b_1, t_1)$ and $Tin(t_1, Paris)$), while b_2 is in Boston ($Bin(b_2, Boston)$). For the action schema $unload(b^*, t^*)$, we can instantiate b^* and t^* with b_1 and t_1 respectively, which will help us to achieve the goal; or we can instantiate b^* and t^* with b_2 and t_1 respectively, which will have no effect. Therefore we need to perform maximization over action arguments to get the best instance of an action.

In SDP, this is done by sorting each partition in $Q_V^{A(\vec{x})}$ by the value in decreasing order and including the negated conditions for the first n partitions in the partition formula for the $(n+1)^{th}$ partition, ensuring that a partition can be satisfied only when no higher value partition can be satisfied. Notice how this step leads to complex description of the resulting state partitions.

Finally, to get the next value function we maximize over the Q-functions of different actions.

The solution of ReBel (Kersting et al., 2004) follows the same outline but uses a simpler logical language, a probabilistic STRIPS-like language, for representing RMDPs. More importantly the paper uses a decision list (Rivest, 1987) style representation for value functions and policies. The decision list gives us an implicit maximization operator since rules higher on the list are evaluated first. As a result the object maximization step is very simple in ReBel. Each state partition is represented implicitly by the negation of all rules above it, and explicitly by the conjunction in the rule. On the other hand regression in ReBel requires that one enumerate all possible matches between a subset of a conjunctive goal (or state partition) and action effects and reason about each of these separately.

Besides exact solution methods, there are other representation languages and algorithms based on approximation or heuristic methods (Guestrin et al., 2003a; Mausam & Weld, 2003; Fern et al., 2003; Gardiol & Kaelbling, 2003; Gretton & Thiebaux, 2004; Sanner & Boutilier, 2005; Sanner & Boutilier, 2006). For example, (Fern et al., 2003; Gretton & Thiebaux, 2004) use inductive learning methods, which first solve instantiations of RMDPs in small domains and then generalize these policies to large domains. Sanner and Boutilier (2005; 2006) give a method that does not need to propositionalize the domain. They represent value functions as a linear combination of first order basis functions and obtain the weights by lifting the propositional approximate linear programming techniques (Schuurmans & Patrascu, 2001; Guestrin et al., 2003b) to handle the first order case.

2.4 Partially Observable MDPs

The MDP model assumes that the underlying state of the process will be known with certainty during plan execution. The POMDP generalizes the MDP model by allowing for another form of uncertainty — the states are not directly observable. We add an observation space O and an observation function $Pr(o|s,a)$, denoting the probability of observing o when action a is executed and the resulting state is s.

Since the agent does not know the state, a belief state — a probability distribution over all states — is commonly used. Let b^a represent the updated belief state that results from doing action a in belief state b and b^a_o from observing o after doing a in b, we can update a belief state as follows (Kaelbling et al., 1998):

$$b^a(s) = \sum_{s' \in S} Pr(s|s', a) b(s') \tag{2.3}$$

$$b^a_o(s) = Pr(o|s, a) b^a(s) / \sum_{s' \in S} Pr(o|s', a) b^a(s') \tag{2.4}$$

We use $Pr(o|b, a)$ to denote the denominator $\sum_{s' \in S} Pr(o|s', a) b^a(s')$ in equation 2.4, which is the probability of observing o given the belief state b and action a.

Given a belief state, its successor belief state is determined by the action and the observation. Therefore a POMDP can be converted to a MDP over belief space. The Bellman update now becomes:

$$V_{n+1}(b) = max_{a \in A}[\sum_{s \in S} b(s) r(s) + \gamma \sum_{o \in O} Pr(o|b, a) V_n(b^a_o)] \tag{2.5}$$

Since the state space for this MDP is a $|S|$-dimensional continuous space, it is much more complex than MDPs with discrete state space. A significant amount of work has been devoted to finding efficient algorithms for POMDPs, e.g., (Hauskrecht, 1997; Cassandra, 1998; Hansen, 1998; Zhang et al., 1999). In the following sections we review some of them.

2.4.1 Exact Algorithms

Like fully observable MDPs, value iteration and policy iteration are two commonly used algorithms for solving POMDPs. Value iteration algorithms conduct search in the value function space whereas policy iteration search in the policy space.

Most exact algorithms are value iteration algorithms. They start with an initial value function and iteratively perform Bellman updates to generate the value function for the next stage. Although the states of the belief state MDP are continuous, Sondik (1971) and Smallwood and Sondik (1973) prove that value functions for POMDPs are piecewise linear and convex (PWLC) and can be represented by a finite set of $|S|$-dimensional vectors $\{v^1, \cdots, v^n\}$. That is, the value function can be represented as $V(b) = max_i \sum_s b(s)v^i(s)$.

There are several value iteration algorithms, e.g., one-pass (Sondik, 1971), linear support (Cheng, 1988), enumeration algorithm (Monahan, 1982), incremental pruning (Zhang & Liu, 1997; Cassandra et al., 1997), and the witness algorithm (Kaelbling et al., 1998). The first two algorithms generate a new set of vectors directly by backing up on a finite number of systematically generated belief points. The next two algorithms first generate a set of vectors that are not parsimonious and then prune redundant vectors. The witness algorithm (Kaelbling et al., 1998) first constructs for each action a parsimonious set of vectors based on a finite number of belief points called "witness points" and then puts together vectors for different actions for further pruning. Among these algorithms, the incremental pruning (IP) algorithm is easy to understand and implement, yet very efficient, and thus becomes the basis for many other algorithms for POMDPs, such as incremental pruning with point-based improvement (Zhang & Zhang, 2001) and propositionally factored POMDPs (Hansen & Feng, 2000). We will explain this algorithm in more detail in Chapter 6 when we discuss relational POMDPs.

Policy iteration for POMDPs was first suggested by Sondik (1978), who proposed to represent policies as finite state controllers (FSC). Hansen (1998) proposed a more practical and implementable version. A policy is represented with a FSC, with each node associated with an action and each arc associated with an observation. The algorithm starts with an arbitrary FSC, and each iteration, just like that of MDPs, consists of two steps: policy evaluation and policy improvement. First, the current policy is evaluated by solving a set of linear equations, yielding a set of $|S|$-dimensional vectors, one for each FSC node. Next a dynamic programming update, e.g., incremental pruning, is used to generate a new set of vectors from the set of vectors obtained from policy evaluation. For each new vector that is added in the update, a new node is added to FSC and useless nodes are pruned away. Hansen (1998) proved that the algorithm converges to the optimal policy and also showed empirically that policy iteration converges faster than value iteration.

2.4.2 Approximations

Solving POMDPs exactly is hard (Papadimitrios & Tsitsiklis, 1987; Madani et al., 2003). Therefore there has been a substantial amount of research directed to calculating good approximations of the optimal solution. Hauskrecht (2000) provides a comprehensive survey of computing approximate value functions of POMDPs. For example, grid-based method (Lovejoy, 1991; Brafman, 1997; Hauskrecht, 1997; Zhou & Hansen, 2001) is the first approximation approach for POMDPs and is still widely used (Zhou & Hansen, 2001). It approximates value functions by discretizing the belief space using a fixed or variable grid and maintaining values only for the grid points. Values at non-grid points are estimated by interpolation/extrapolation. Another example of value function approximation is to approximate each vector that composes the piecewise linear convex value functions by a linear combination of basis functions (Guestrin et al., 2001).

There has also been work on approximation of belief states. Belief states can be represented compactly using Dynamic Bayesian Networks (DBNs), e.g., Forbes et al. (1995) determine the current belief state by the network for the current time slice along with current percepts. Unfortunately, even though variables start out being independent (thus admitting a compact representation of distribution) and there are only a few connections between one variable and another, over time, all the variables may become fully correlated. Therefore, compact representation of belief states is usually impossible (Poupart & Boutilier, 2000; Boyen & Koller, 1998). Based on this observation, several approximation schemes have been proposed. Boyen and Koller (1998) approximate belief states of DBNs by dividing the state variables into a set of subset of variables and using a product of marginals of each subset as a simplification, assuming that the subsets are independent. Koller and Fratkina (1998) propose another approximation scheme based on density trees, which are similar to classification decision trees. A density tree splits on the domains of variables, and the probabilities labeling the leaves denote the summations of probabilities of every state consistent with the corresponding branch. Moreover, the distribution at each leaf is uniform over these states, making density trees similar in meaning to our belief state representation which is covered in Chapter 6.

2.4.3 Heuristic Search

Since the POMDP is a belief state MDP, we can build a stochastic decision tree and perform heuristic search just as in MDPs. The differences is that each tree node is now a belief state, and given an action the probability of transitioning to the next belief state is determined by $Pr(o|b, a)$ (as defined in Section 2.4).

Washington (1996; 1997) perform AO* search in the belief state space, which can be used to solve finite-horizon problems. We can use LAO* for infinite-horizon POMDPs; however testing whether two belief states are equal (so as to know whether they refer to the same state in the search space) is a costly operation. Hansen (1998) uses AO* in a variant of policy iteration when the initial state is known. Although AO* is used in this case, the solution can be cyclic because it uses value function corresponding to a finite state controller as the lower bound function and improves the controller during search. The algorithm converges to a ε-optimal policy for the initial belief state after a finite number of steps.

Bonet and Geffner (2000) use RTDP-BEL, an adaptation of RTDP for solving belief state MDPs, to search for an approximation of the optimal plan. Bonet and Geffner (2001) further provide a unified view of solving planning problems, where the action dynamics can be deterministic, non-deterministic, and probabilistic, and sensor feedback can be null, partial, and complete, as a heuristic search in the form of real-time dynamic programming in the (belief) state space.

2.4.4 Heuristic Solutions

To solve a POMDP we can solve the underlying MDP, and use that as the basis of various heuristics. Cassandra (1998) provides a set of heuristic solution methods. The simplest one is the most likely state (MLS) method — find the most likely state according to the current belief state and choose the action according to the policy for the underlying MDP. The most widely used is the Q-MDP method. Instead of using policies as in the MLS, it uses the Q-functions of the optimal policy for the underlying MDP, and choose the action that maximize $\sum_s b(s)Q^a(s)$. The problem with this approach is that it assumes that all the uncertainty will vanish in the next step, because the underlying MDP is fully observed. Hence it will never perform information gathering actions (Cassandra, 1998).

Cassandra (1998) tries to alleviate this problem by a dual model control method — if the entropy of the belief state is below a pre-specified threshold, use one of the above heuristics; otherwise choose an action to reduce the entropy. This problem can also be partially overcome by doing deeper lookahead, as in (Washington, 1996; Washington, 1997). Zubek and Dietterich (2000) propose another approach to solve this problem. Instead of using the underlying MDP as an approximation of the POMDP value function, they define an "even-odd" POMDP that is identical to the original POMDP except that the state is fully observable at even time steps. This even-odd POMDP can be converted into an equivalent MDP (the 2MDP) with different actions and rewards. Let V^*_{2MDP} be the optimal value function for the 2MDP (which captures some of the sensing costs of the original POMDP). They get an improved approximation to the POMDP value function by performing a shallow lookahead search and evaluating the leaf states using V^*_{2MDP}. Zubek and Dietterich (2001) further define a chain-MDP approximation when the actions can be partitioned into those that change the world and those that are pure sensing actions. For such problems, the chain-MDP algorithm is able to capture more of the sensing costs than the even-odd POMDP approximation in many cases (Zubek & Dietterich, 2001).

2.4.5 Propositionally Factored POMDPs

Several representations for propositionally factored POMDPs have been proposed. Boutilier and Poole (1996) describe an algorithm based on Monahan's enumeration algorithm that exploits the factored representation in a POMDP in the form of Dynamic Bayesian Network to compute decision tree structured value functions and policies. But the belief state is not represented in a compact way. When it comes to policy execution, one must maintain a distribution over flat states online. Hansen and Feng (2000) extend this algorithm by using Algebraic Decision Diagrams instead of decision trees and an efficient incremental pruning algorithm instead of Monahan's enumeration algorithm to compute the dynamic programming update. Their test results indicate that a factored state representation can significantly speed up dynamic programming in the best case and incurs little overhead in the worst case.

2.4.6 First Order POMDPs

First order representations have also been proposed for POMDPs. Poole (1997) uses choice space to capture uncertainty, and uses logic programs to capture deterministic certainty. Bacchus et al. (1999) extend the situation calculus to include probability. But these two works only focus on knowledge representation and do not build a planner. Geffner and Bonet (1998) use a goal satisfaction model and compile the high-level modeling language into the underlying POMDP with flat states; therefore their approach does not scale up well to large problems.

2.5 Other Related Work

In a classical planning framework like STRIPS (Fikes & Nilsson, 1972), actions have deterministic effects and the initial state is known. Under these assumptions, all states that are reachable during plan execution can be computed trivially from the initial state and the actions executed. There is no need to obtain feedback from the environment. Plans generated are in the form of sequence of actions, totally ordered or partially ordered. Some examples include UCPOP (Penberthy & Weld, 1992), Graphplan (Blum & Furst, 1995), and SatPlan (Kautz & Selman, 1996). Weld (1999) surveys AI planning techniques.

However, classical planners are rarely useful in the real world domains that involve uncertainty, because the plan may fail at an early stage. In the past decade, there has been growing interest in generalizing classical planners that ignored stochastic uncertainty to planning problems that include uncertainty about both the effects of actions and the problem state. These are called conditional and/or probabilistic planners, e.g., C-BURIDAN (Drape et al., 1994), Weaver (Blythe, 1998), and Mahinur (Onder, 1997). These systems extend STRIPS representations to allow for stochastic actions. They first develop a base plan using a classical planning method, then improve plans by either increasing the chance of the occurrence of the desired outcome of an action or dealing with the situation when the desired outcome of an action does not occur by forming new branches that indicate alternative response to different observational result or simply removes one action and replace it with an alternative. These systems use a goal satisfaction model and ignore plan cost. C-BURIDAN and Weaver give plans that reach any goal state with probability exceeding some threshold and Mahinur tries to improve the probability until time allotted has expired. In addition, Weaver introduces an explicit mechanism to efficiently deal with exogenous events, i.e., events beyond the control of an agent. Mahinur selects flaws for refinement by estimating how much utility could be gained based on a simplifying assumption of additive, independent rewards associated with each top-level subgoal.

Hyafil and Bacchus (2003) provide an interesting comparison between solving conformant probabilistic planning (CPP) via constraint satisfaction problems (CSP) and via POMDPs (with complete unobservability), and their findings are that the CSP method is faster for shorter plans but the POMDP method eventually catches up — the complexity of the CSP method is always exponential in the plan length, while the increase in the number of vectors needed to represent the value function in POMDPs tends to slow down as plan length increases.

There has also been research on planning under partial observability in non-deterministic domains, where uncertainties are treated as disjunctions and there is no information about the relative likelihood of possible outcomes or states. As a result, one must plan for all contingencies, which makes scaling a concern. However, this may be desirable in domains where all eventualities need to be considered. For example, Bertoli et al. (2001) propose conditional planning under partial uncertainty as and-or heuristic search of the possibly cyclic graph induced by the domain, and used Binary Decision Diagram (BDD) based techniques, which provide data structures and algorithms for making the operations on belief sets more efficient. The conditional acyclic plans generated are guaranteed to achieve the goal despite of the uncertainty in the initial condition and the effects of actions. However, the planner cannot solve problems that require a cyclic plan, such as the Omelette problem (Geffner & Bonet, 1998).

Boutilier et al. (1999) provide a comprehensive survey on MDPs and explain how MDPs provide a unifying framework for sequential decision making problems including AI planning. The survey includes compact representations of MDPs (e.g., DBN and PStrips) and efficient solution methods utilizing abstraction, aggregation, and decomposition methods.

Chapter 3

First Order Decision Diagrams

In this chapter we describe our representation framework FODDs. We examine the syntax and semantics of the language, and develop reduction operators that remove redundancies in FODDs. These are crucial to keep the representation small and make algorithms feasible.

3.1 Propositional Decision Diagrams

A decision diagram is a labeled directed acyclic graph where non-leaf nodes are labeled with propositional variables, each non-leaf node has exactly two children corresponding to `true` and `false` branches, and leaves are labeled with numerical values. A decision diagram is a compact representation of a mapping from a set of boolean variables to a set of values. A Binary Decision Diagram (BDD) represents a mapping to a boolean set $\{0, 1\}$ (Bryant, 1986). An Algebraic Decision Diagram (ADD) represents a more general mapping to any discrete set of values (Bahar et al., 1993).

An ADD defines a function f for each node using the following rules (Bahar et al., 1993):

1. For a leaf node labeled c, $f = c$.
2. For non-leaf node labeled x, $f = x \cdot f_{true} + (\neg x) \cdot f_{false}$, where f_{true} and f_{false} are functions corresponding to its two children.

The functions corresponding to a BDD is defined in the same way except that "·" is replaced by $\wedge$ and "+" replaced by $\vee$. Figure 3.1 shows a BDD representing the boolean function $x_1 \vee x_2 \vee x_3$. In this diagram as well as in the rest of the thesis, left going edges represent `true` branches.

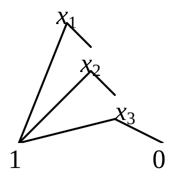


Figure 3.1: A BDD representing the boolean function $x_1 \vee x_2 \vee x_3$.

Ordered decision diagrams (ODD) specify a fixed order on propositions and require that node labels respect this order on every path in the diagram. It is well known (Bryant, 1986) that starting with any ODD, we can reduce its size by repeatedly applying the following rules until neither is applicable:

1. Merge nodes with same label and same children.

2. Remove a node with both children leading to the same node.

For a given variable ordering, every function has a unique canonical representation. That is, we have the following theorem:

Theorem 1 (Bryant, 1986)
If two ODDs D_1 and D_2 are reduced, and they represent the same function, then D_1 and D_2 are isomorphic.

This theorem shows that reducing an ODD gives a normal form. This property means that propositional theorem proving is easy for ODD representations. For example, if a formula is contradictory then this fact is evident when we represent it as a BDD, since the normal form for a contradiction is a single leaf valued 0. This together with efficient manipulation algorithms for ODD representations lead to successful applications, e.g., in VLSI design and verification (Bryant, 1992; McMillan, 1993; Bahar et al., 1993).

3.2 First Order Decision Diagrams

3.2.1 Syntax of First Order Decision Diagrams

There are various ways to generalize ADDs to capture relational structure. One could use closed or open formulas in the nodes, and in the latter case we must interpret the quantification over the variables. In the process of developing the ideas in this thesis we have considered several possibilities including explicit quantifiers but these did not lead to useful solutions. We therefore focus on the following syntactic definition which does not have any explicit quantifiers.

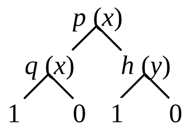


Figure 3.2: A simple FODD.

Definition 1 *1. We assume a fixed set of predicates and constant symbols, and an enumerable set of variables. We also allow using an equality between any pair of terms (constants or variables).*

2. A First Order Decision Diagram (FODD) is a labeled directed acyclic graph, where each non-leaf node has exactly two children. The outgoing edges are marked with values `true` and `false`.

3. Each non-leaf node is labeled with: an atom $P(t_1, \ldots, t_n)$ or an equality $t_1 = t_2$ where each t_i is a variable or a constant.

4. Leaves are labeled with numerical values.

We can see that the definition of a FODD is quite similar to that of an ADD except that it allows for more expressivity in a non-leaf node — a node can be labeled with a predicate or an equality in addition to a propositional literal.

Figure 3.2 shows a FODD with binary leaves. To simplify diagrams in the thesis we draw multiple copies of the leaves 0 and 1 (and occasionally other values or small sub-diagrams) but they represent the same node in the FODD.

We use the following notation: for a node n, $n_{\downarrow t}$ denotes the `true` branch of n, and $n_{\downarrow f}$ the `false` branch of n; $n_{\downarrow a}$ is an outgoing edge from n, where a can be `true` or `false` . For an edge e, $source(e)$ is the node that edge e issues from, and $target(e)$ is the node that edge e points to. Let e_1 and e_2 be two edges, we have $e_1 = sibling(e_2)$ iff $source(e_1) = source(e_2)$. It is clear that the two outgoing edges of any node are siblings to each other.

In the following we will slightly abuse the notation and let $n_{\downarrow a}$ mean either an edge or the sub-FODD this edge points to. We will also use $n_{\downarrow a}$ and $target(e_1)$ interchangeably where $n = source(e_1)$ and a can be `true` or `false` depending on whether e_1 lies in the `true` or `false` branch of n.

3.2.2 Semantics of First Order Decision Diagrams

We use a FODD to represent a function that assigns values to states. For example, in the logistics domain, we would like to assign values to different states in such a way that if there is a box in Paris, then the state is assigned a value of 19; if there is no box in Paris but there is a box on a truck that is in Paris and it is raining, this state is assigned a value of 6.3, etc, as shown in Figure 3.3. So the question is how we define semantics of FODDs in order to have the intended meaning?

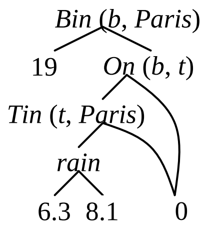


Figure 3.3: An example of the value function.

The semantics of first order formulas are given relative to interpretations. An interpretation has a domain of elements, a mapping of constants to domain elements, and for each predicate a relation over the domain elements which specifies when the predicate is true. In the MDP context, an interpretation is just a state. For example in the logistics domain, a state includes objects such as boxes, trucks, and cities, and relations among them, such as box 1 on truck 1 ($On(b_1, t_1)$), box 2 in Paris ($Bin(b_2, Paris)$), etc. There is more than one way to define the meaning of FODD B on interpretation I. In the following we discuss two possibilities.

Semantics Based on a Single Path

A semantics for decision trees is given by Blockeel and De Raedt (1998) that can be adapted to FODDs. The semantics define a unique path that is followed when traversing B relative to I. All variables are existential and a node is evaluated relative to the path leading to it.

In particular, when we reach a node some of its variables have been seen before on the path and some are new. Consider a node n with label $l(n)$ and the path leading to it from the root, and let C be the conjunction of all labels of nodes that are exited on the `true` branch on the path. Then in the node n we evaluate $\exists\vec{x}, C \wedge l(n)$, where $\vec{x}$ includes all the variables in C and $l(n)$. If this formula is satisfied in I then we follow the `true` branch. Otherwise we follow the `false` branch. This process defines a unique path from the root to a leaf and its value.

For example, if we evaluate the diagram in Figure 3.2 on the interpretation with domain $\{1, 2, 3\}$ and relations $\{p(1), q(2), h(3)\}$ then we follow the `true` branch at the root since $\exists x, p(x)$ is satisfied, but we follow the `false` branch at $q(x)$ since $\exists x, p(x) \wedge q(x)$ is not satisfied. Since the leaf is labeled with 0 we say that B does not satisfy I. This is an attractive approach, because it builds mutually exclusive partitions over states, and various FODD operations can be developed for it. However, for reasons we discuss later this semantics is not so well suited to value iteration, and it is therefore not used in the thesis.

Semantics Based on a Multiple Paths

Following Groote and Tveretina (2003) we define the semantics first relative to a variable valuation ζ. Given a FODD B over variables $\vec{x}$ and an interpretation I, a valuation ζ maps each variable in $\vec{x}$ to a domain element in I. Once this is done, each node predicate evaluates either to `true` or `false` and we can traverse a single path to a leaf. The value of this leaf is denoted by $\mathrm{MAP}_B(I,\zeta)$.

Different valuations may give different values; but recall that we use FODDs to represent a function over states, and each state must be assigned a single value. Therefore, we next define

$$\mathrm{MAP}_B(I) = \mathrm{aggregate}_\zeta\{\mathrm{MAP}_B(I,\zeta)\}$$

for some aggregation function. That is, we consider all possible valuations ζ, and for each we calculate $\mathrm{MAP}_B(I,\zeta)$. We then aggregate over all these values. In the special case of Groote and Tveretina (2003) leaf labels are in $\{0,1\}$ and variables are universally quantified; this is easily captured in our formulation by using minimum as the aggregation function. In this thesis we use maximum as the aggregation function. This corresponds to existential quantification in the binary case (if there is a valuation leading to value 1, then the value assigned will be 1) and gives useful maximization for value functions in the general case. We therefore define:

$$\mathrm{MAP}_B(I) = \max_\zeta\{\mathrm{MAP}_B(I,\zeta)\}$$

Consider evaluating the diagram in Figure 3.2 on the interpretation with domain $\{1,2,3\}$ and relations $\{p(1), q(2), h(3)\}$. The valuation where x is mapped to 2 and y is mapped to 3 denoted $\{x/2, y/3\}$ leads to a leaf with value 1 so the maximum is 1. When leaf labels are in {0,1}, we can interpret the diagram as a logical formula. When $\mathrm{MAP}_B(I) = 1$, as in our example, we say that I satisfies B and when $\mathrm{MAP}_B(I) = 0$ we say that I falsifies B.

We define node formulas (NF) and edge formulas (EF) recursively as follows. For a node n labeled $l(n)$ with incoming edges $e_1, \ldots, e_k$, the node formula $\text{NF}(n) = (\vee_i \text{EF}(e_i))$. The edge formula for the `true` outgoing edge of n is $\text{EF}(n_{\downarrow t}) = \text{NF}(n) \wedge l(n)$. The edge formula for the `false` outgoing edge of n is $\text{EF}(n_{\downarrow f}) = \text{NF}(n) \wedge \neg l(n)$. These formulas, where all variables are existentially quantified, capture the conditions under which a node or edge are reached.

3.2.3 Basic Reduction of FODDs

Groote and Tveretina (2003) define several operators that reduce a diagram into "normal form". A total order over open predicates (node labels) is assumed. We describe these operators briefly and give their main properties.

(R1) Neglect operator: if both children of a node p in the FODD lead to the same node q then we remove p and link all parents of p to q directly.

(R2) Join operator: if two nodes p, q have the same label and point to the same 2 children then we can join p and q (remove q and link q's parents to p).

(R3) Merge operator: if a node and its child have the same label then the parent can point directly to the grandchild.

(R4) Sort operator: If a node p is a parent of q but the label ordering is violated ($l(p) > l(q)$) then we can reorder the nodes locally using 2 copies of p and q such that labels of the nodes do not violate the ordering.

Define a FODD to be reduced if none of the 4 operators can be applied. We have the following:

Theorem 2 (Groote & Tveretina, 2003)
(1) Let $O \in \{Neglect, Join, Merge, Sort\}$ be an operator and $O(B)$ the result of applying O to FODD B, then for any ζ, $MAP_B(I, \zeta) = MAP_{O(B)}(I, \zeta)$
(2) if B_1, B_2 are reduced and satisfy $\forall \zeta$, $MAP_{B_1}(I, \zeta) = MAP_{B_2}(I, \zeta)$ then they are identical.

Property (1) gives soundness, and property (2) shows that reducing a FODD gives a normal form. However, this only holds if the maps are identical for every ζ and this condition is stronger than normal equivalence. This normal form suffices for Groote and Tveretina (2003) who use it to provide a theorem prover for first order logic, but is not strong enough for our purposes. Figure 3.4 shows two pairs of reduced FODDs (with respect to R1-R4) such that $\text{MAP}_{B_1}(I) = \text{MAP}_{B_2}(I)$ but $\exists\zeta, \text{MAP}_{B_1}(I,\zeta) \neq \text{MAP}_{B_2}(I,\zeta)$. In this case although the maps are the same the FODDs are not reduced to the same form. In Section 3.3.2 we show that with additional reduction operators we have developed, B1 in the first pair is reduced to one. Thus the first pair have the same form after reduction. However, our reductions do not resolve the second pair. Even though B1 and B2 are logically equivalent (notice that we just change the order of two nodes and rename variables), they cannot be reduced to the same form using R1-R4 or our new operators. Notice that both these functions capture a path of two edges labeled p in a graph. To identify a unique minimal syntactic form one may have to consider all possible renamings of variables and the sorted diagrams they produce, but this is an expensive operation. See (Garriga et al., 2007) for a discussion of normal form for conjunctions that uses such an operation.

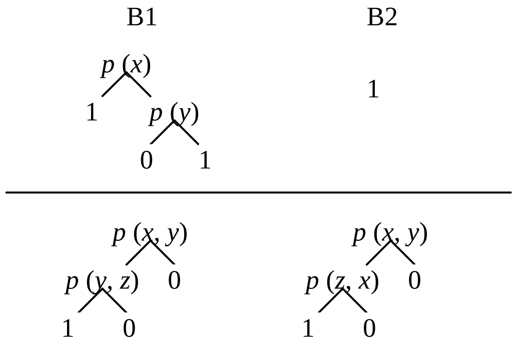


Figure 3.4: Examples illustrating weakness of normal form.

3.2.4 Combining FODDs

Given two algebraic diagrams we may need to add the corresponding functions, take the maximum or use any other binary operation, `op`, over the values represented by the functions. Here we adopt the solution from the propositional case (Bryant, 1986) in the form of the procedure **Apply**(p,q,`op`) where p and q are the roots of two diagrams. This procedure chooses a new root label (the lower among labels of p, q) and recursively combines the corresponding sub-diagrams, according to the relation between the two labels ($<$, $=$, or $>$). In order to make sure the result is reduced in the propositional sense one can use dynamic programming to avoid generating nodes for which either neglect or join operators ((R1) and (R2) above) would be applicable.

Figure 3.5 illustrates this process. In this example, we assume predicate ordering as $p_1 < p_2$, and parameter ordering $x_1 < x_2$.

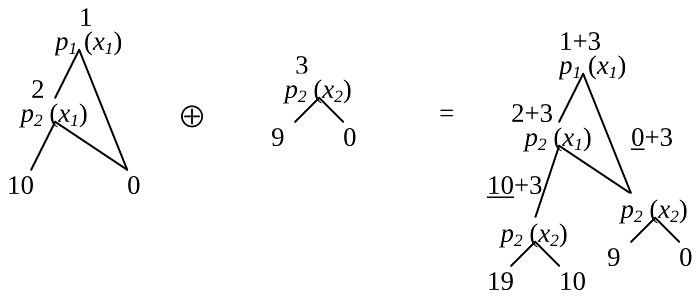


Figure 3.5: A simple example of adding two FODDs.

3.2.5 Order of Labels

The syntax of FODDs allows for two "types" of objects: constants and variables. In the process of our MDP algorithm we use a third type which we call action parameters. This is required by the structure of the value iteration algorithm given in Chapter 5. Action parameters behave like constants in some places and like variables in others.

We assume a complete ordering on predicates, constants, action parameters, and variables. Any argument of a predicate can be a constant or an action parameter or a variable. The ordering $<$ between two labels is given by the following rules:

1. $P(x_1, ..., x_n) < P'(x'_1, ..., x'_m)$ if $P < P'$
2. $P(x_1, ..., x_n) < P(x'_1, ..., x'_n)$ if there exists i such that $type(x_i) < type(x'_i)$ or $type(x_i) = type(x'_i)$ and $x_i < x'_i$ (where type can be constant, action parameter, or variable) and $x_j = x'_j$ for all $j < i$.

It may be helpful if the equality predicate is the first in the predicate ordering so that equalities are at the top of the diagrams. During reduction we often encounter situations where one side of the equality can be completely removed. It may also be helpful to order the parameter types in the following order: constant < action parameters < variables. This ordering may be helpful for reductions. Intuitively, a variable appearing later can be bound to the value of a constant so it is better to place the variable lower in the diagram. The action parameters are constants before object maximization, but become variables after object maximization, so we put them in the middle.

The rest of this chapter focuses on reductions. Readers can understand the thesis without reading it, but notice that reductions are performed to minimize the FODD size whenever possible.

3.3 Additional Reduction Operators

In our context, especially for algebraic FODDs we may want to reduce the diagrams further. We distinguish *strong reduction* that preserves $\text{MAP}_B(I, \zeta)$ for all ζ and *weak reduction* that only preserves $\text{MAP}_B(I)$. In the following let $\mathcal{B}$ represent any background knowledge we have about the domain. For example in the Blocks World we may know that $\forall x, y, [on(x, y) \rightarrow \neg clear(y)]$.

When we define conditions for reduction operators, there are two types of conditions: the reachability condition and the value condition. We name reachability conditions by starting with P (for Path Condition) and the reduction operator number. We name conditions on values by starting with V and the reduction operator number.

3.3.1 (R5) Strong Reduction for Implied Branches

Consider any node n such that whenever n is reached then the `true` branch is followed. In this case we can remove n and connect its parent directly to the `true` branch. We first present the condition, followed by the lemma regarding this operator.

(P5) : $\mathcal{B} \models \forall \vec{x}, [\mathrm{NF}(n) \rightarrow l(n)]$ where $\vec{x}$ are the variables in $\mathrm{EF}(n_{\downarrow t})$.

Let R5-remove(n) denote the operator that removes node n and connects its parent directly to the `true` branch. Notice that this is a generalization of R3. It is easy to see that the following lemma is true:

Lemma 1 *Let B be a FODD, n a node for which condition P5 hold, and B' the result of R5-remove(n), then for any interpretation I and any valuation ζ we have $MAP_B(I,\zeta) = MAP_{B'}(I,\zeta)$.*

A similar reduction can be formulated for the `false` branch, i.e., if $\mathcal{B} \models \forall \vec{x}, [\mathrm{NF}(n) \rightarrow \neg l(n)]$ then whenever node n is reached then the `false` branch is followed. In this case we can remove n and connect its parent directly to the `false` branch.

Implied branches may simply be a result of equalities along a path. For example $(x = y) \wedge p(x) \rightarrow p(y)$ so we may prune $p(y)$ if $(x = y)$ and $p(x)$ are known to be true.

Implied branches may also be a result of background knowledge. For example in the Blocks World if $on(x, y)$ is guaranteed to be true when we reach a node labeled $clear(y)$ then we can remove $clear(y)$ and connect its parent to $clear(y)_{\downarrow f}$.

3.3.2 (R6) Weak Reduction Removing Dominated Siblings

Consider any node n such that if we can reach node n using some valuation then we can reach $n_{\downarrow t}$ using a possibly different valuation. Intuitively, if $n_{\downarrow t}$ always gives better values than $n_{\downarrow f}$ then we should be able to remove $n_{\downarrow f}$ from the diagram. We first present all the conditions needed for the operator and show relations between them, and then follow with the definition of the operator.

(P6.1) : $\mathcal{B} \models \forall \vec{x}, [\mathrm{NF}(n) \rightarrow \exists \vec{y}, l(n)]$ where $\vec{x}$ are the variables that appear in $\mathrm{NF}(n)$, and $\vec{y}$ the variables in $l(n)$ and not in $\mathrm{NF}(n)$. This condition requires that every valuation reaching n can be extended into a valuation reaching $n_{\downarrow t}$.

(P6.2) : $\mathcal{B} \models [\exists \vec{x}, \text{NF}(n)] \rightarrow [\exists \vec{x}, \vec{y}, \text{EF}(n_{\downarrow t})]$. This condition requires that if n is reachable then $n_{\downarrow t}$ is reachable but does not put any restriction on the valuations (in contrast with the requirement in P6.1 to extend the valuation reaching n).

(P6.3) : $\mathcal{B} \models \forall \vec{u}, [\exists \vec{v}, \text{NF}(n)] \rightarrow [\exists \vec{v}, \vec{w}, \text{EF}(n_{\downarrow t})]$ where $\vec{u}$ are the variables that appear in $n_{\downarrow t}$, $\vec{v}$ the variables that appear in NF(n) but not in $n_{\downarrow t}$, and $\vec{w}$ the variables in $l(n)$ and not in $\vec{u}$ or $\vec{v}$. This condition requires that for every valuation ζ_1 that reaches $n_{\downarrow f}$ there is a valuation ζ_2 that reaches $n_{\downarrow t}$ and such that ζ_1 and ζ_2 agree on all variables in (the sub-diagram of) $n_{\downarrow t}$.

(P6.4) : no variable in $\vec{y}$ appears in the sub-diagram of $n_{\downarrow t}$, where $\vec{y}$ is defined as in P6.1.

(P6.5) : $\mathcal{B} \models \forall \vec{r}, [\exists \vec{v}, \text{NF}(n)] \rightarrow [\exists \vec{v}, \vec{w}, \text{EF}(n_{\downarrow t})]$ where $\vec{r}$ are the variables that appear in both $n_{\downarrow t}$ and $n_{\downarrow f}$, $\vec{v}$ the variables that appear in NF(n) but not in $\vec{r}$ and $\vec{w}$ the variables in $l(n)$ and not in $\vec{r}$ or $\vec{v}$. This condition requires that for every valuation ζ_1 that reaches $n_{\downarrow f}$ there is a valuation ζ_2 that reaches $n_{\downarrow t}$ and such that ζ_1 and ζ_2 agree on the intersection of variables in (the sub-diagrams of) $n_{\downarrow t}$ and $n_{\downarrow f}$.

(V6.1) : $\min(n_{\downarrow t}) \geq \max(n_{\downarrow f})$ where $\min(n_{\downarrow t})$ is the minimum leaf value in $n_{\downarrow t}$, and $\max(n_{\downarrow f})$ the maximum leaf value in $n_{\downarrow f}$. In this case regardless of the valuation we know that it is better to follow $n_{\downarrow t}$ and not $n_{\downarrow f}$.

(V6.2) : all leaves in the diagram $D = n_{\downarrow t} - n_{\downarrow f}$ have non-negative values (denoted as $D \geq 0$). In this case for any fixed valuation it is better to follow $n_{\downarrow t}$ instead of $n_{\downarrow f}$.

We have the following lemma regarding relations among the conditions above:

Lemma 2

(a) P6.1 $\rightarrow$ P6.2

(b) P6.1 $\wedge$ P6.4 $\rightarrow$ P6.3

(c) P6.3 $\rightarrow$ P6.5 $\rightarrow$ P6.2

(d) V6.1 $\rightarrow$ V6.2

Proof: It is clear that (a), (c), and (d) hold.

To prove (b), let $\vec{u}$ denote variables that appear in $n_{\downarrow t}$. From the conditions that

$\mathcal{B} \models \forall\vec{x}, [\text{NF}(n) \rightarrow \exists\vec{y}, l(n)]$ and no variables in $\vec{y}$ appear in $n_{\downarrow t}$, we have $\mathcal{B} \models \forall(\vec{x} \cup \vec{u}), [\text{NF}(n) \rightarrow \exists\vec{y}, l(n)]$. Let $\vec{v}$ denote all variables that appear in $\text{NF}(n)$ but not in $n_{\downarrow t}$, this is the same as $\mathcal{B} \models \forall(\vec{v} \cup \vec{u}), [\text{NF}(n) \rightarrow \exists\vec{y}, l(n)]$, which is indeed stronger than P6.3. ■

We define the operator R6-replace$(b, n_{\downarrow f})$ as replacing $n_{\downarrow f}$ with any constant value b between 0 and $\min(n_{\downarrow t})$. We have the following lemmas about this operator.

Lemma 3 *Let B be a FODD, n a node for which condition P6.2 and V6.1 hold, and B' the result of R6-replace$(b, n_{\downarrow f})$, then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: By P6.2, for any valuation ζ_1 that reaches $n_{\downarrow f}$ there is another valuation ζ_2 reaching $n_{\downarrow t}$, and by V6.1 it gives better value. Therefore, the map will never be determined by the `false` branch and we can replace it with any constant value between 0 and $\min(n_{\downarrow t})$ without changing the map. ■

Lemma 4 *Let B be a FODD, n a node for which condition P6.5 and V6.2 hold, and B' the result of R6-replace$(b, n_{\downarrow f})$, then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation ζ_1 that reaches $n_{\downarrow f}$. By the condition P6.5, there is also a valuation ζ_2 that reaches $n_{\downarrow t}$. Because they agree on the variables in the intersection of $n_{\downarrow t}$ and $n_{\downarrow f}$, ζ_2 must lead to a better value (otherwise, there would be a branch in $D = n_{\downarrow t} - n_{\downarrow f}$ with negative value). Therefore according to maximum aggregation the value of $\text{MAP}_B(I)$ will never be determined by the `false` branch. Therefore we can replace $n_{\downarrow f}$ with a constant. ■

Note that the conditions in two lemmas are not comparable since $P6.5 \rightarrow P6.2$ and $V6.1 \rightarrow V6.2$.

In some cases we can also drop the node n completely and connect its parents directly to $n_{\downarrow t}$. We define this action as R6-drop(n). We have the following lemma:

Lemma 5 *Let B be a FODD, n a node for which condition P6.3 and V6.2 hold, and B' the result of R6-drop(n), then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Note first that since P6.3 $\rightarrow$ P6.5 we can replace $n_{\downarrow f}$ with a constant. Consider any valuation ζ_1 for the FODD. If ζ_1 leads to $n_{\downarrow t}$ it will continue reaching $n_{\downarrow t}$ and the value will not change. If ζ_1 leads to $n_{\downarrow f}$ it will now lead to $n_{\downarrow t}$ reaching some leaf of the diagram (and giving its value instead of the constant value assigned above). By the condition P6.3, ζ_2 will reach the same leaf and assign the same value. So under maximum aggregation the map is not changed. ■

Symmetric operators for R6-replace and R6-drop can be defined and used. The transformation is straightforward so we omit the details.

An important special case of R6 occurs when $l(n)$ is an equality $t_1 = y$ where y is a variable that does not occur in the FODD above node n. In this case, the condition P6.1 holds since we can choose the value of y. We can also enforce the equality in the sub-diagram of $n_{\downarrow t}$. Therefore if V6.1 holds we can remove the node n connecting its parents to $n_{\downarrow t}$ and substituting t_1 for y in the diagram $n_{\downarrow t}$. (Note that we may need to make copies of nodes when doing this.) As we see below this case is typical when using FODDs for MDPs.

However, we can not handle inequality (i.e., when it is better to go to the `false` branch child of the equality) in the same way. Even if we assume that the domain has more than two objects, thus the inequality can always hold if it is at the root, we cannot drop the node. The best we can do is to replace the `true` branch of the equality with a constant if all relevant conditions hold. Consider the example shown in Figure 3.6(a). If we drop the node $b = b1$, we get the FODD as shown in Figure 3.6(b), from which we can further apply R6 to remove $Bin(b_1, Paris)$. Suppose we have an interpretation I with domain $\{box_1, box_2, truck_1\}$ and relations $\{Bin(box_1, Paris), Tin(truck1, Paris)\}$. MAP$_{B1}(I) = 0.1$. But if the equality is removed, MAP$_{B2}(I) = 10$ via valuation $\{b/box_1, b_1/box_1, t/truck_1\}$. Since our initial development of these ideas Joshi (2007) has developed a more elaborate reduction to handle equalities by taking a maximum over the left and right children.

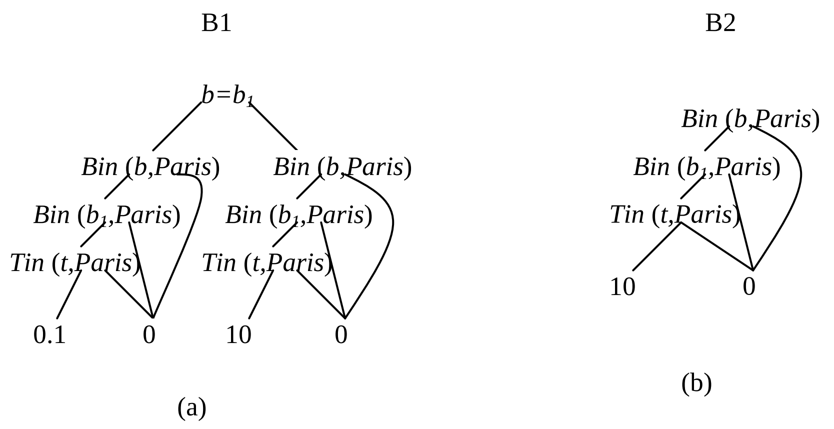


Figure 3.6: An example illustrating the need to keep the inequality.

Some care is needed when applying weak reductions, e.g. when replacing a subdiagram with 0 as above. While this preserves the map it does not preserve the map for every valuation. If we apply non-monotonic operations that depend on all valuations (e.g. subtract two diagrams that share variables) the result may be incorrect. However, the reductions are always safe for monotonic operations and when we only operate in this way if different diagrams do not share variables.

Examples and Discussion

In practice it may be worth checking P6.1 and V6.1 instead of other conditions just to save time, but sometimes these conditions are not strong enough. Consider the example shown in Figure 3.7(a). Here we can see that the FODD is not reducible using P6.1 and V6.1. First, the valuation reaching $Tin(t_2, Paris)$ may not be extended to reach $Tin(t_2, Paris)_{\downarrow t}$. Second, $min(Tin(t_2, Paris)_{\downarrow t}) \ngeq max(Tin(t_2, Paris)_{\downarrow f}$. But it is indeed reducible if we use relaxed conditions on reachability (P6.2) and values (V6.2). First we can remove the node $Tin(t_2, Paris)$ because both P6.3 and V6.2 hold and we get the FODD shown in Figure 3.7(b). Then we further remove the node $On(b_2, t_2)$ and get the result as shown in Figure 3.7(c).

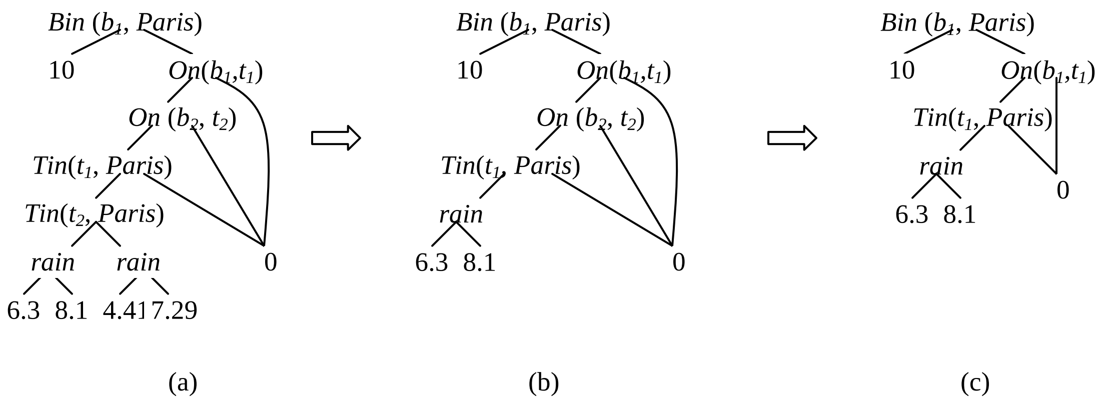


Figure 3.7: An example illustrating the need to relax R6 conditions.

Figure 3.8 illustrates two cases in R6. In part (a) conditions P6.1, P6.4 and V6.1 hold and we can drop $p(y)$ completely. We cannot drop $p(y)$ in part (b). Intuitively removing $p(y)$ removes a constraint on y. Consider an interpretation I with domain $\{1,2\}$ and relations $\{p(1), q(1), f(2)\}$. Before reduction, $\text{MAP}_{B1}(I) = 1$. But if $p(y)$ is removed $\text{MAP}_{B2}(I) = 2$.

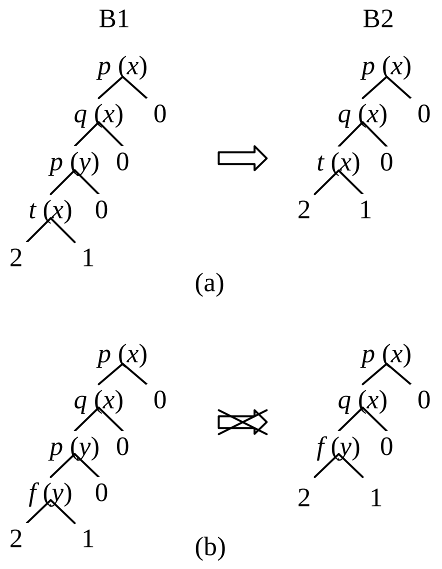


Figure 3.8: Examples illustrating Remove Reduction R6.

Note that the only difference between lemma 3 and lemma 4 is that the former requires that the two valuations agree on only a subset of variables in $n_{\downarrow t}$ that also appear in $n_{\downarrow f}$ for replacing a branch with a constant, while the latter requires two valuations agree on all the variables in $n_{\downarrow t}$ for removing a node. We can use the same example to illustrate why we need stronger condition for removing a node. Again look at the node $p(y)$; $p(y)_{\downarrow t}$ has variable y, while $p(y)_{\downarrow f}$ is a constant. Therefore, the intersection is empty, and conditions P6.5 and V6.2 hold. But we have seen that we cannot remove the node. Intuitively, when we remove a node, we must guarantee that we do not gain extra value. If only condition P6.5 and V6.2 hold, we can only guarantee that for any valuation reaching $n_{\downarrow f}$, there is another valuation reaching $n_{\downarrow t}$ with a better value. But if we remove the node n, a valuation that was supposed to reach $n_{\downarrow f}$ may reach a better value that it would never have reached otherwise. This would change the map. Condition P6.3 is sufficient, but not necessary. A weaker condition can be stated as: for any valuation ζ_1 that reaches $n_{\downarrow f}$ and thus will be redirected to reach a value v_1 in $n_{\downarrow t}$ when n is removed, there is a valuation ζ_2 that reaches $n_{\downarrow t}$ with a value v_2 and $v_2 \geq v_1$. However, this condition may be too complex to test in practice.

Note that condition P6.5 is also sufficient, but not necessary. Sometimes valuations just need to agree on a smaller set of variables than the intersection of variables. To see this, consider the example as shown in Figure 3.9, where $A - B > 0$ and the intersection is $\{x_3, x_4\}$. But we just need to agree on $\{x_4\}$ for if we rename x_3 in left diagram as x_3' then we still get $A - B > 0$. Intuitively, if the variable in the first FODD does not cause loss in value (compared with the situation if the variable did not exist), and this same variable in the second FODD does not cause increase in value, then we can rename the variable and still get the same result.

We can develop a recursive procedure to find a smaller set of variables than the intersection that still guarantees that $A - B > 0$ (Joshi, 2007). But the smallest size set is not unique. Consider the following example as shown in Figure 3.10.

In this example, to get $A - B > 0$ we must preserve either $\{x, z\}$ or $\{x, y\}$. Intuitively we have to agree on the variable x to avoid the situation when two paths $p(x, y) \wedge \neg q(x)$ and $p(x, y) \wedge q(x) \wedge h(z)$ can co-exist. In order to prevent the co-existence of two paths

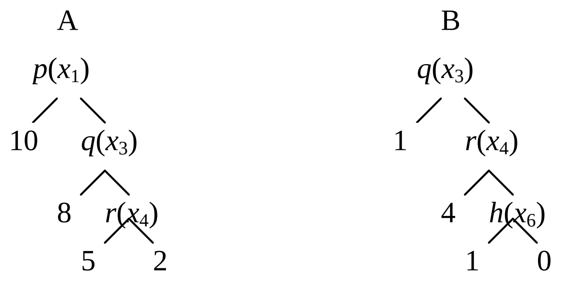


Figure 3.9: An Example illustrating that intersection of variables is more than we need sometimes in P6.5.

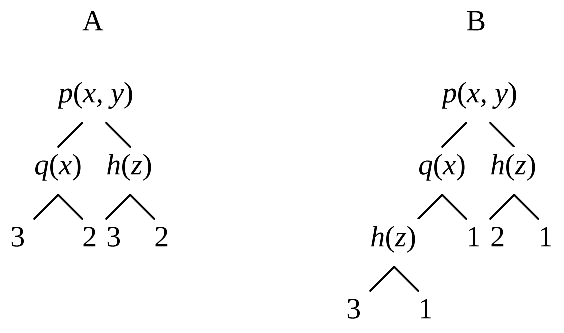


Figure 3.10: An example illustrating a minimum set of variables.

$\neg p(x, y) \wedge \neg h(z)$ and $p(x, y) \wedge q(x) \wedge h(z)$, either y or z have to be the same as well. Now if we change this example a little bit and replace each $h(z)$ with $h(z, v)$, then we have two minimum sets of variables of different size, one is $\{x, y\}$, and the other is $\{x, z, v\}$.

Note that the reachability condition P6.2 alone together with V6.2, i.e., combining the weaker portions of conditions from Lemma 3 and Lemma 4, cannot guarantee that we can replace a branch with a constant. Figure 3.11 illustrates this. Note that both conditions hold for $p(y)$. Consider an interpretation I with domain $\{1, 2\}$ and relations $\{p(1), q(2)\}$. Before reduction, $\text{MAP}_{B1}(I) = 6$ via valuation $\{x/1, y/2\}$. But if $p(y)_{\downarrow f}$ is replaced with 0, $\text{MAP}_{B2}(I) = 0$. Intuitively the subtraction operation $D = n_{\downarrow t} - n_{\downarrow f}$ is propositional, so the test in V6.2 implicitly assumes that the common variables in the operants should be the same and P6.2 does not check this.

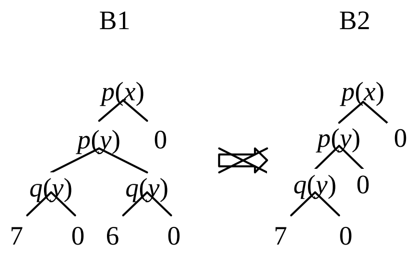


Figure 3.11: An example illustrating the condition for replacing a branch with 0 in R6.

Also note that, although $P6.1 \wedge P6.4$ is a sufficient reachability condition for dropping a node, we cannot drop a node if the condition $P6.2 \wedge P6.4$ hold for reachability. Figure 3.12 illustrates this. It is easy to see that conditions P6.2 and P6.4 hold for $q(x_2)$. Consider an interpretation I with domain $\{1, 2\}$ and relations $\{p(1, 1), p(2, 1), q(1), r(2)\}$. Before reduction, $\text{MAP}_{B1}(I) = 1$ via valuation $\{x_1/x_2/1, y_1/y_2/1\}$. But if $q(x_2)$ is removed, $\text{MAP}_{B2}(I) = 2$ via valuation $\{x_1/1, x_2/2, y_1/y_2/1\}$.

Finally, we illustrate the symmetric operator when $n_{\downarrow t}$ is dominated by $n_{\downarrow f}$. Consider the first example in Figure 3.4. Here the condition P6.1 hold for $p(y)$: $\forall x, [\neg p(x) \rightarrow \exists y, \neg p(y)]$ as well as V6.1: $\min(p(y)_{\downarrow f}) = 1 \geq \max(p(y)_{\downarrow t}) = 0$ and P6.4. Therefore we can remove $p(y)$ and connect its parent $p(x)$ directly to 1. Note that the application of some reductions may trigger other reductions. In this case, both children of $p(x)$ now

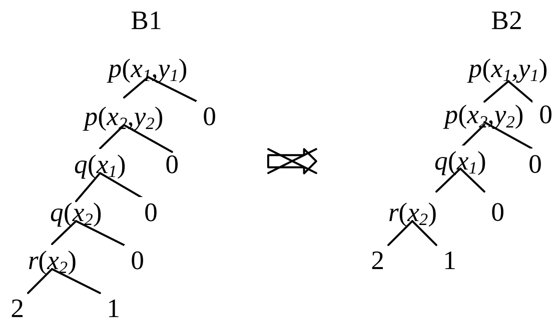


Figure 3.12: An example illustrating conditions P6.2 and P6.4 cannot guarantee that we can drop a node.

point to 1. So we can use neglect operator and the final result is 1.

Application Order, Node Reordering, and Normal Form

In some cases the order of application of reductions is important. Consider Figure 3.13. R6 is applicable to two nodes: $p(x_2, y_2)$ and $q(x_2)$. If we do it in a top down manner, we find that we cannot remove node $p(x_2, y_2)$, because x_2 is used below. But if we start from $q(x_2)$, we first remove this node, which makes removing $p(x_2, y_2)$ possible. So it may be useful to apply R6 in a bottom up manner.

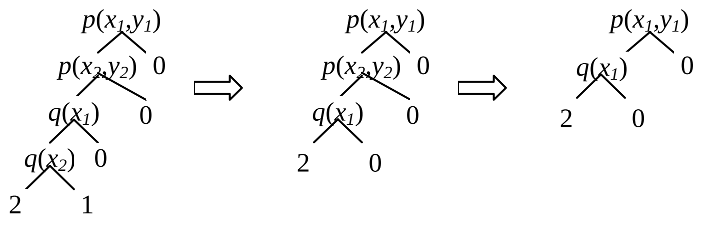


Figure 3.13: An example illustrating the order of how R6 is applied.

Sometimes for R6 to be applicable, we need to reorder the nodes. Consider Figure 3.14(a). Here although the condition P6.1 holds for $On(b_2, t_2)$, we cannot remove this node because t_2 is used below. But if we reorder the nodes so that $On(b_1, t^*)$ is placed below $On(b_2, t_2)$, we can remove the node $On(b_1, t^*)$.

One might hope that repeated application of R6 will lead to a unique reduced result

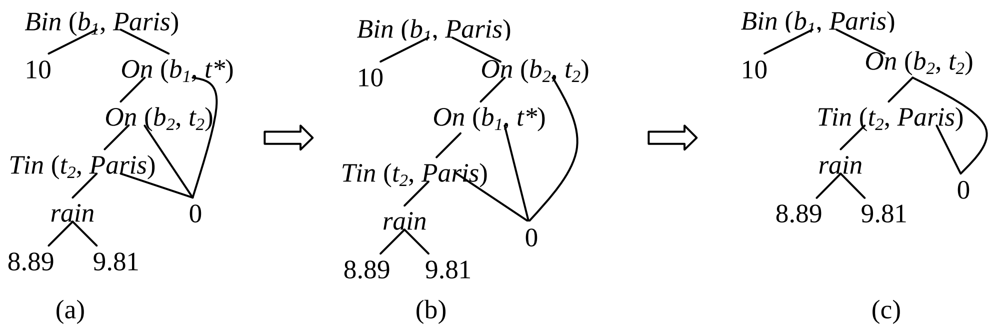


Figure 3.14: An example illustrating that we may need to reorder the nodes in order to apply R6.

but this is not true. In fact, the final result depends on the choice of operators and order of application. Consider Figure 3.15(a). We can apply R6 to the node $Tin(t^*, Paris)$ and we get the FODD shown in Figure 3.14(a). To see why the reachability condition P6.2 holds, consider the following formula:

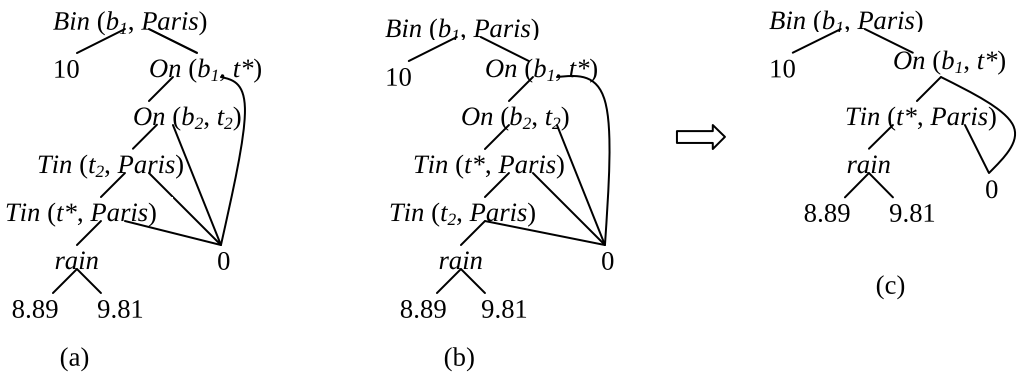


Figure 3.15: An example illustrating that the reductions are order dependent.

$$\mathcal{B} \models \exists b_1, b_2, t_2, t^*, [\neg Bin(b_1, Paris) \wedge On(b_1, t^*) \wedge On(b_2, t_2) \wedge Tin(t_2, Paris)]$$

$$\rightarrow \exists \underline{b_1}, \underline{b_2}, \underline{t_2}, \underline{t^*}, [\neg Bin(\underline{b_1}, Paris) \wedge On(\underline{b_1}, \underline{t^*}) \wedge On(\underline{b_2}, \underline{t_2}) \wedge Tin(\underline{t_2}, Paris) \wedge Tin(\underline{t^*}, Paris)]$$

This condition holds because we can let both $\underline{b_1}$ and $\underline{b_2}$ take the value of b_2, and both $\underline{t_2}$ and $\underline{t^*}$ take the value of t_2, and use the domain knowledge $\forall b, c, [\exists t, On(b,t) \rightarrow \neg Bin(b,c)]$ ($On(b_2, t_2)$ means that $\neg Bin(b_2, Paris)$ is true). Also P6.3 holds too because there is no variable below the node $Tin(t^*, Paris)$. Since P6.3 and V6.2 hold, we can remove this node and get the FODD shown in Figure 3.14(a). After reordering and application of R6 we get the final result as shown in Figure 3.14(c).

However, if we do reordering first and put $Tin(t^*, Paris)$ above $Tin(t_2, Paris)$, we have a FODD as shown in Figure 3.15(b). We can apply R6 first on the node $Tin(t_2, Paris)$ then on node $On(b_2, t_2)$ and get the final result as shown in Figure 3.15(c). Comparing the final results in Figure 3.15(c) and Figure 3.14(c), we can see that although their maps are the same for any interpretation, they are not isomorphic. From this example we can see that the reductions are order dependent, at least when we allow for node reordering.

3.3.3 (R7) Weak Reduction Removing Dominated Edges

Consider any two edges e_1 and e_2 in a FODD whose formulas satisfy that if we can follow e_2 then we can also follow e_1. As in R6 if e_1 gives better value than e_2 then intuitively we can replace e_2 with a constant.

Let $p = source(e_1), q = source(e_2)$, $e_1 = p_{\downarrow a}$, and $e_2 = q_{\downarrow b}$, where a and b can be `true` or `false` , depending on whether e_1 or e_2 lies in the `true` or `false` branch of p or q.

Again we first present all the conditions and then give the reductions.

(P7.1) : $\mathcal{B} \models [\exists \vec{x}, \mathrm{EF}(e_2)] \rightarrow [\exists \vec{y}, \mathrm{EF}(e_1)]$ where $\vec{x}$ are the variables in $\mathrm{EF}(e_2)$ and $\vec{y}$ the variables in $\mathrm{EF}(e_1)$.

(P7.2) : $\mathcal{B} \models \forall \vec{u}, [[\exists \vec{w}, \mathrm{EF}(e_2)] \rightarrow [\exists \vec{v}, \mathrm{EF}(e_1)]]$ where $\vec{u}$ are the variables that appear in both $target(e_1)$ and $target(e_2)$, $\vec{v}$ the variables that appear in $\mathrm{EF}(e_1)$ but are not in $\vec{u}$, and $\vec{w}$ the variables that appear in $\mathrm{EF}(e_2)$ but are not in $\vec{u}$. This condition requires that for every valuation ζ_1 that reaches e_2 there is a valuation ζ_2 that reaches e_1 and such that ζ_1 and ζ_2 agree on all variables in the intersection of $target(e_1)$ and $target(e_2)$.

(P7.3) : $\mathcal{B} \models \forall \vec{r}, [[\exists \vec{s}, \mathrm{EF}(e_2)] \rightarrow [\exists \vec{t}, \mathrm{EF}(e_1)]]$ where $\vec{r}$ are the variables that appear in both $target(e_1)$ and $target(sibling(e_2))$, $\vec{s}$ the variables that appear in $\mathrm{EF}(e_1)$ but are not in $\vec{r}$, and $\vec{t}$ the variables that appear in $\mathrm{EF}(e_2)$ but are not in $\vec{r}$. This condition requires that for every valuation ζ_1 that reaches e_2, there is a valuation ζ_2 that reaches e_1 and such that ζ_1 and ζ_2 agree on all variables in the intersection of $target(e_1)$ and $target(sibling(e_2))$.

(V7.1) : $\min(target(e_1)) \geq \max(target(e_2))$

(V7.2) : $\min(target(e_1)) \geq \max(target(sibling(e_2)))$

(V7.3) : all leaves in $D = target(e_1) - target(e_2)$ have non-negative values, i.e., $D \geq 0$.

(V7.4) : all leaves in $G = target(e_1) - target(sibling(e_2))$ have non-negative values.

We define the operators R7-replace(b, e_1, e_2) as replacing $target(e_2)$ with a constant b that is between 0 and $\min(target(e_1))$ (we may write it as R7-replace(e_1, e_2) if $b = 0$), and R7-drop(e_1, e_2) as dropping the node $q = source(e_2)$ and connecting its parents to $target(sibling(e_2))$. We further present an additional condition, followed by lemmas regarding the operators.

(S1) : NF$(source(e_1))$ and sub-FODD of $target(e_1)$ remain the same before and after R7-replace and R7-drop.

Lemma 6 *Let B be a FODD, e_1 and e_2 edges for which condition P7.1, V7.1, and S1 hold, and B' the result of R7-replace(b, e_1, e_2), then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation ζ_1 that reaches $target(e_2)$. Then according to P7.1, there is another valuation reaching $target(e_1)$ and by V7.1 it gives a higher value. Therefore, MAP$_B(I)$ will never be determined by $target(e_2)$ so we can replace $target(e_2)$ with a constant between 0 and $\min(target(e_1))$ without changing the map. ■

Lemma 7 *Let B be a FODD, e_1 and e_2 edges for which condition P7.2, V7.3, and S1 hold, and B' the result of R7-replace(b, e_1, e_2), then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation ζ_1 that reaches $target(e_2)$. By P7.2 there is another valuation ζ_2 reaching $target(e_1)$ and by V7.3 it achieves a higher value (otherwise, there must be a branch in $D = target(e_1) - target(e_2)$ with negative value). Therefore according to maximum aggregation the value of MAP$_B(I)$ will never be determined by $target(e_2)$, and we can replace it with a constant as described above. ■

Lemma 8 *Let B be a FODD, e_1 and e_2 edges for which condition V7.2 hold in addition to the conditions for replacing target(e_2) with a constant, and B' the result of R7-drop(e_1, e_2), then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation reaching $target(e_2)$. As above its true value is dominated by another valuation reaching $target(e_1)$. When we remove $q = source(e_2)$ the valuation will reach $target(sibling(e_2))$ and by V7.2 the value produced is smaller than the value from $target(e_1)$. So again the map is preserved. ■

Lemma 9 *Let B be a FODD, e_1 and e_2 edges for which P7.3 and V7.4 hold in addition to conditions for replacing target(e_2) with a constant, and B' the result of R7-drop(e_1, e_2), then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation ζ_1 reaching $target(e_2)$. As above its value is dominated by another valuation reaching $target(e_1)$. When we remove $q = source(e_2)$ the valuation will reach $target(sibling(e_2))$ and by the condition P7.3, ζ_2 will reach leaf of greater value in $target(e_1)$(otherwise there will be a branch in G leading to a negative value). So under maximum aggregation the map is not changed. ■

To summarize if both P7.1 and V7.1 hold or both P7.2 and V7.3 hold then we can replace $target(e_2)$ with a constant. If we can replace and V7.2 or both P7.3 and V7.4 hold then we can drop $q = source(e_2)$ completely.

Examples and Discussion

Figure 3.16 illustrates why we cannot remove a node if only P7.1 and V7.1 hold. Notice that the conditions hold for $e_1 = [p(x)]_{\downarrow t}$ and $e_2 = [p(y)]_{\downarrow t}$ so we can replace $[p(y)]_{\downarrow t}$ with a constant. Consider an interpretation I with domain $\{1, 2\}$ and relations $\{q(1), p(2), h(2)\}$. Before reduction, $MAP_{B1}(I) = 10$ via valuation $\{x/2\}$. But if $p(y)$ is removed, $MAP_{B2}(I) = 20$ via valuation $\{x/1, y/2\}$. Therefore we need the additional condition V7.2 to guarantee that we will not gain extra value with node dropping.

We also use an example to illustrate why we need the condition S1, which says that we must not harm the value promised by $target(e_1)$. In other words, we must guarantee

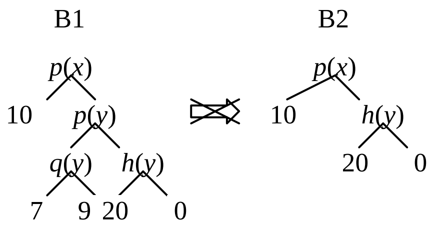


Figure 3.16: An example illustrating the condition for removing a node in R7.

that $p = source(e_1)$ is reachable just as before and the sub-FODD of $target(e_1)$ is not modified after replacing a branch with 0. Recall that $p = source(e_1)$ and $q = source(e_2)$. The condition is violated if q is in the sub-FODD of $p_{\downarrow t}$, or if p is in the sub-FODD of $q_{\downarrow t}$. But it holds in all other cases, that is when p and q are unrelated (one is not the descendent of the other), or q is in the sub-FODD of $p_{\downarrow f}$, or p is in the sub-FODD of $q_{\downarrow f}$.

Consider Figure 3.17(a). Here conditions P7.2 and V7.3 hold for $e_1 = [r(y)]_{\downarrow t}$ and $e_2 = [r(x)]_{\downarrow t}$. So if we ignore S1, we could replace $[r(x)]_{\downarrow t}$ with 0, which of course gives a wrong result. R6 can be used here on node $r(y)$, and this gives the result as shown in Figure 3.17(c).

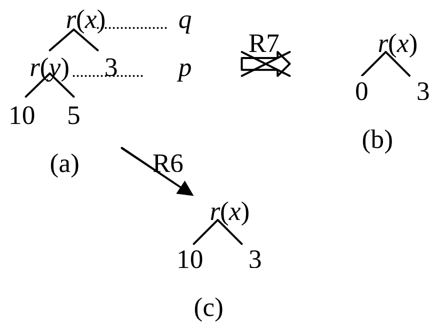


Figure 3.17: An example illustrating the condition for removing a node in R7.

Note that conditions of Lemma 6 and Lemma 7 are not comparable since P7.2→P7.1 and V7.1→V7.3. The reachability condition P7.1 together with V7.3 cannot guarantee that we can replace a branch with a constant. Figure 3.18 illustrates this. Consider an interpretation I with domain $\{1, 2, 3, 4\}$ and relations $\{h(1, 2), q(3, 4), p(2)\}$. The domain knowledge is that $\exists x, y, h(x, y) \rightarrow \exists z, w, q(z, w)$. So P7.1 and V7.3 hold for $e_1 = [q(x, y)]_{\downarrow t}$

and $e_2 = [h(z,y)_{\downarrow t}]$. Before reduction, $\text{MAP}_{B1}(I) = 3$. But if $h(z,y)_{\downarrow t}$ is replaced with 0, $\text{MAP}_{B2}(I) = 0$. As we saw in R6 the substitutions mentioned in P7.2 must preserve variables in $\vec{u}$ because apply has an implicit assumption that the common variables in the operants are the same.

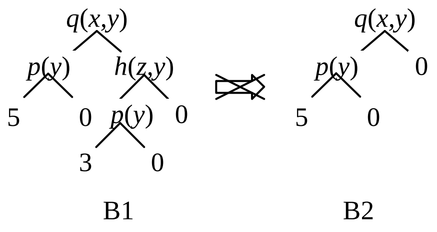


Figure 3.18: An example illustrating the subtraction condition in R7.

We can further relax the conditions in R7 to get wider applicability. Let T be a diagram with $q = source(e_2)$ as root but with $target(sibling(e_2))$ replaced with 0. Consider the diagram $D^* = target(e_1) - T$. We modify the condition V7.3 as follows:

(V7.3*) : all leaves in D^* have non-negative values, i.e., $D^* \geq 0$. Then it is clear that for any fixed valuation it is better to follow e_1 instead of going through q to e_2.

If we use condition $V7.3^*$, we need to modify the condition P7.2 to the following:

(P7.2*) : $\mathcal{B} \models \forall\vec{u}, [[\exists\vec{w}, \text{EF}(e_2)] \rightarrow [\exists\vec{v}, \text{EF}(e_1)]]$. The only difference from P7.2 is that now $\vec{u}$ denotes the variables that appear in the intersection of $target(e_1)$ and T, i.e., the union of $target(e_2)$ and $l(q)$ where $q = source(e_2)$. I.e., $\vec{u} = var(target(e_1)) \cap (var(target(e_2)) \cup var(l(q)))$.

To see why this relaxation is useful, consider the FODD in Figure 3.19(a). If we look at $e_1 = [p(x_1)]_{\downarrow t}$ and $e_2 = [p(x_2)]_{\downarrow t}$ that is in the sub-FODD of $p(x_1)_{\downarrow f}$, we can see that V7.3 does not hold because $[p(x_1)]_{\downarrow t} - [p(x_2)]_{\downarrow t} \not\geq 0$. But now with relaxation of V7.3, we can apply R7 and Figure 3.19(b) shows reduced result. Note however that we can obtain the same final result using different operators. We first apply R7 on a pair of edges $[q(y)]_{\downarrow t}$ and $[q(y)]_{\downarrow t}$ in left and right sub-diagrams, and get the FODD in Figure 3.19(d). Then we can apply R7 on a pair of edges $[q(y)]_{\downarrow f}$ and $[q(y)]_{\downarrow f}$ in left and right sub-diagrams, and this time we can remove the node $p(y)$ (on the right side). We get

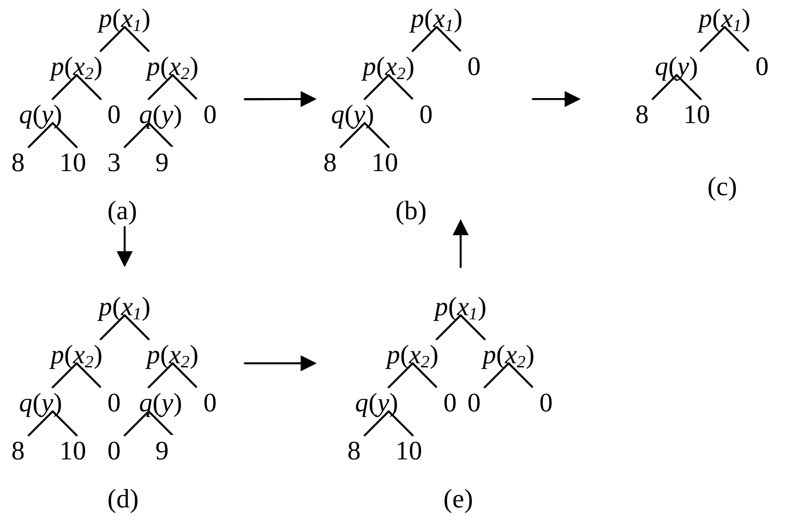


Figure 3.19: An example illustrating the need to relax condition V7.3 in R7.

the FODD in Figure 3.19(e). By applying neglect operator, we obtain the FODD shown in Figure 3.19(b). In this case we need to go through more steps to get the same result.

As in R6 order of application of R7 may be important. Consider the FODD in Figure 3.20(a). R7 is applicable to edges $e_1 = [p(x_1, y_1)]_{\downarrow t}$ and $e_2 = [p(x_2, y_2)]_{\downarrow t}$, and $e_1' = [q(x_3)]_{\downarrow t}$ and $e_2' = [q(x_2)]_{\downarrow t}$. If we do it in a top down manner, i.e., first apply R7 on the pair $[p(x_1, y_1)]_{\downarrow t}$ and $[p(x_2, y_2)]_{\downarrow t}$, we will get the FODD in Figure 3.20(b), and then we apply R7 again on $[q(x_3)]_{\downarrow t}$ and $[q(x_2)]_{\downarrow t}$, and we will get the FODD in Figure 3.20(c). However, if we apply R7 first on $[q(x_3)]_{\downarrow t}$ and $[q(x_2)]_{\downarrow t}$ (thus get Figure 3.20(d)), it will make it impossible for R7 to apply to $[p(x_1, y_1)]_{\downarrow t}$ and $[p(x_2, y_2)]_{\downarrow t}$ because $[p(x_1, y_1)]_{\downarrow t} - [p(x_2, y_2)]_{\downarrow t}$ will have negative leaves now. However, this does not mean that we cannot reduce the FODD using this order. We can do reduction by comparing $[q(x_3)]_{\downarrow t}$ and $[q(x_2)]_{\downarrow t}$ that is in the right part of FODD. We can first remove $q(x_2)$ and get a FODD shown in Figure 3.20(e), and then use neglect operator to remove $p(x_2, y_2)$. It is interesting that above two examples both show that we need to take more steps if we choose to work on unrelated nodes first using R7.

R7 can be applied to the FODD in Figure 3.21(a), which cannot be reduced by other reduction operators. Note that R6 is not immediately applicable because $p(y)_{\downarrow t} - p(y)_{\downarrow f} \not\geq$

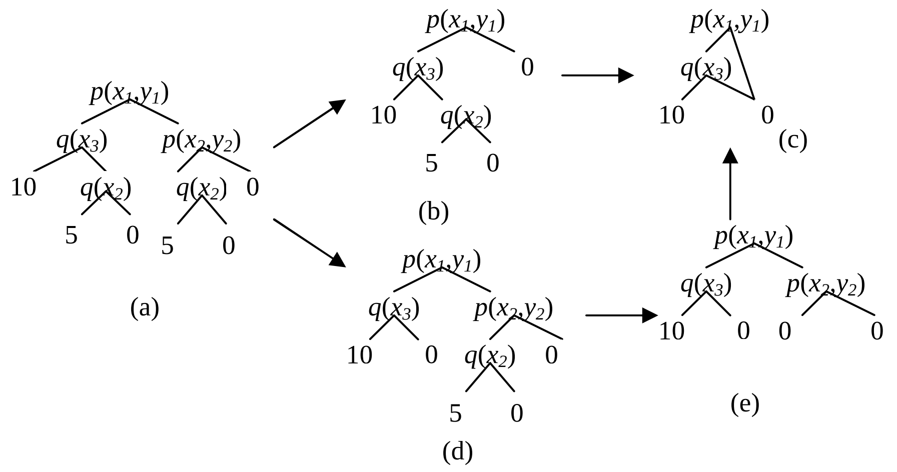


Figure 3.20: An example illustrating the order of applying R7.

0. Since P7.1, V7.1, V7.2, and S7.1 hold for $e_1 = [h(A)]_{\downarrow f}$ and $e_2 = [h(B)]_{\downarrow f}$, we can remove the node $h(B)$ and get the FODD as shown in Figure 3.21(b). Now P6.1, P6.4, and V6.1 hold for the node $p(y)$, so we can apply R6 and remove $p(y)$. Figure 3.21(b) shows the final result.

The last few examples illustrate that R7 can be applied in a flexible way that often avoid the pitfalls we have seen with R6. It remains an open question whether we can use it to get a notion of normal form.

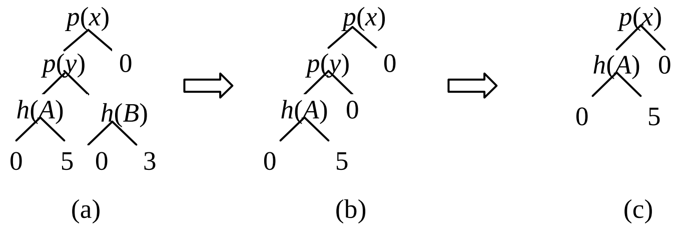


Figure 3.21: An examples illustrating the use of R7.

Simultaneous Application of R7

We can save computation time if we can apply several instances of R7 simultaneously so that we only need to compute edge formulas and conditions once. For this to be possible, we need a theorem that says if conditions for R7 hold for a set of pairs of edges, then we can always find an ordering of applying all these reductions sequentially and therefore we can do reductions simultaneously. As we show next this holds in some cases but not always.

We start with R7-replace(e_A, e_B) under simple conditions, which do not involve subtraction and complex reachability condition. Note that we have the following simple conditions for R7-replace(e_A, e_B) to be applicable:

1. If we can reach e_B, then we can reach e_A (P7.1).

2. $min(e_A) \geq max(e_B)$(V7.1).

3. Reachability of e_A does not change after the reduction (S1).

We have the following theorem regarding simultaneous R7-replace under simple conditions. Note that the theorem gives the required ordering since we can pick the first reduction to apply and then inductively apply the rest.

Theorem 3 *Suppose we have a FODD where all basic reduction operators (i.e., neglect operator, join operator, and merge operator) have been applied. If there are $m \geq 2$ pairs $(e_A^1, e_B^1) \cdots (e_A^m, e_B^m)$ of edges such that P7.1, V7.1, S1 hold for each pair, and none of the edges point to a constant, then there is a k such that after applying R7-replace(e_A^k, e_B^k), P7.1, V7.1, S1 hold for all other pairs.*

Note that these pairs of edges are not necessarily disjoint. We may have cases where one edge dominates more than one edge, e.g., (e_1, e_2) and (e_1, e_3), or one edge dominates and is also dominated by other edges, e.g., (e_1, e_2) and (e_3, e_1). We will refer an edge e as an e_A edge if there is a pair (e_A^i, e_B^i) such that $e_A^i = e$; and an edge e as an e_B edge if there is a pair (e_A^i, e_B^i) such that $e_B^i = e$.

Proof: The proof is by induction. In both the base case and the inductive step, we will look at two situations, i.e.,

1. $\exists i, e_B^i$ is not related to any e_A^j.

2. $\forall i, e_B^i$ is related to some e_A^j.

By "related", we mean that either e_B^i is above e_A^j, or it is below e_A^j, or it is e_A^j. Note that e_B^i cannot be related to e_A^i, otherwise condition S1 will be violated.

In the base case, m=2. So we have two pairs (e_A^1, e_B^1) and (e_A^2, e_B^2).

In situation 1, consider the case where e_B^1 is unrelated to e_A^2 (the other case is symmetric). In this case, we can replace e_B^1 with 0. Consider P7.1 for (e_A^2, e_B^2). e_B^2 may become less reachable (e.g., when e_B^2 is below e_B^1; in the extreme, e_B^2 becomes 0 if it only has one parent, then we need to do nothing on the second pair of edges), but since e_A^2 is not affected, condition P7.1 either still holds or R7-replace(e_A^2, e_B^2) has already been done (in the extreme case). Consider V7.1 for (e_A^2, e_B^2). The value of e_B^2 may be reduced (e.g., when e_B^2 is above e_B^1), but since the value of e_B^2 can only go down and the value of e_A^2 is not affected, condition V7.1 still holds.

In situation 2, every e_B is related to some e_A. We have the following cases, which are illustrated in Figure 3.22.

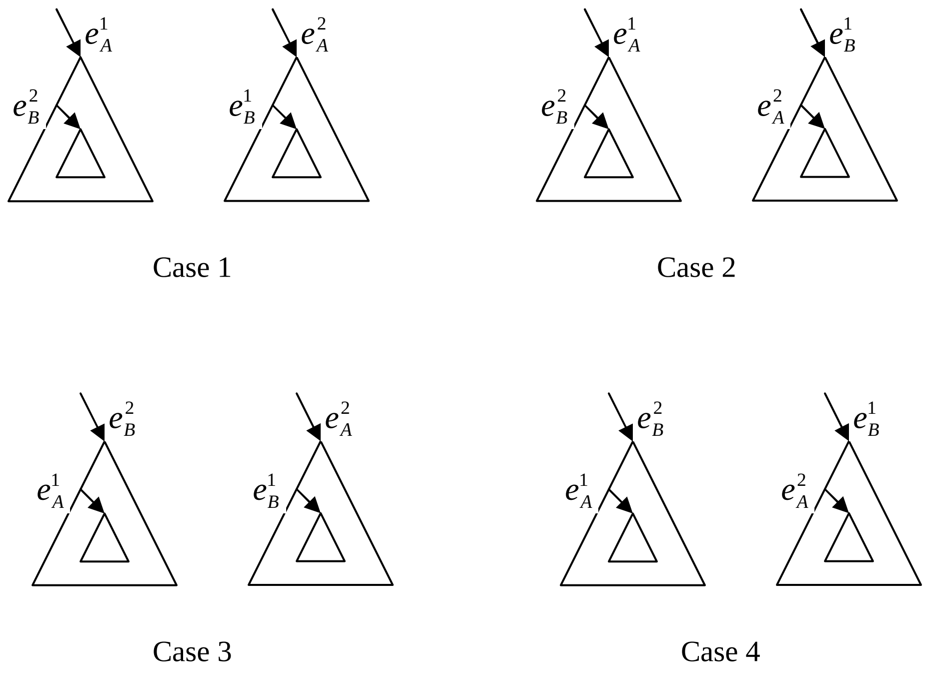


Figure 3.22: Base cases when every B is related to some A.

1. e^1_A above e^2_B, e^2_A above e^1_B.

2. e^1_A above e^2_B, e^2_A below e^1_B.

3. e^1_A below e^2_B, e^2_A above e^1_B.

4. e^1_A below e^2_B, e^2_A below e^1_B.

Note that for each case, we can replace "above" or "below" with "same as". The following argument hold in these cases as well.

Now we prove that situation 2 is impossible, i.e., it is impossible for us to have any of the above cases. Note that if C is a sub-FODD of D, then we have $min(C) \geq min(D)$ and $max(D) \geq max(C)$.

1. $min(e^2_B) \geq^{subtree} min(e^1_A) \geq^{V7.1} max(e^1_B)$
   $\geq min(e^1_B) \geq^{subtree} min(e^2_A) \geq^{V7.1} max(e^2_B)$
   Therefore e^2_B points to a constant. A contradiction to the condition of the theorem.

2. $min(e^2_B) \geq^{subtree} min(e^1_A) \geq^{V7.1} max(e^1_B)$
   $\geq^{subtree} max(e^2_A) \geq min(e^2_A) \geq^{V7.1} max(e^2_B)$
   Therefore e^2_B points to a constant. A contradiction to the condition of the theorem.

3. $max(e^1_A) \leq^{subtree} max(e^2_B) \leq^{V7.1} min(e^2_A)$
   $\leq^{subtree} min(e^1_B) \leq max(e^1_B) \leq^{V7.1} min(e^1_A)$
   Therefore e^1_A points to a constant. A contradiction to the condition of the theorem.

4. $max(e^1_A) \leq^{subtree} max(e^2_B) \leq^{V7.1} min(e^2_A)$
   $\leq max(e^2_A) \leq^{subtree} max(e^1_B) \leq^{V7.1} min(e^1_A)$
   Therefore e^1_A points to a constant. A contradiction to the condition of the theorem.

Now for the inductive step. Assume that the theorem holds for $m = k$ pairs of edges. We show that the theorem holds for $m = k + 1$ pairs. Again first look at situation 1. There is an e^i_B that is not related to any e^j_A, then we can apply R7-replace(e^i_A, e^i_B), and we can use a similar argument as in the base case to prove that P7.1, V7.1, S1 hold for the remaining k pairs.

Now look at situation 2, where each e_B^i is related to some e_A^j. We can seek a loop condition such that

e_B^1 above/below e_A^2,

e_B^2 above/below e_A^3,

$\cdots$

e_B^{n-1} above/below e_A^n,

e_B^n above/below e_A^1.

The loop condition is illustrated in Figure 3.23. Note that the vertical pairs of edges

$$(e_B^1, e_A^2), \cdots, (e_B^i, e_A^{i+1}), \cdots, (e_B^n, e_A^1)$$

are about the position (above/below), so we call such a pair "position" pair of edges. The diagonal pairs (e_A^i, e_B^i) are about dominance, and we call such a pair "dominance" pair of edges.

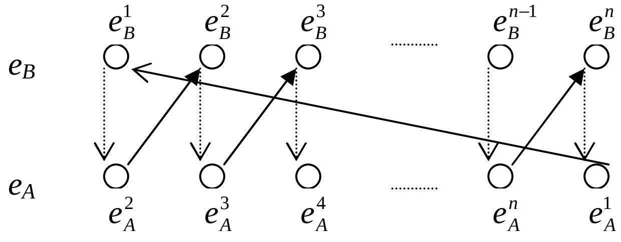


Figure 3.23: A figure to illustrate a loop condition.

We can find a loop by starting with any e_B, and we name it as e_B^1. Since it is related to some e_A, choose one and name it as e_A^2. Look at all (e_A, e_B) pairs where $e_A = e_A^2$, and choose one e_B and name it as e_B^2. Since it is also related to some e_A, choose one and name it as e_A^3. We continue with the chaining procedure until we find a loop. Whenever we have an edge e_B^i, we will look at all edges from e_A's that are related to it and choose one. There are two possibilities: one is that this e_A has appeared in the chain, then we know we have a loop (starting from the e_B that is dominated by this e_A before, and ending with e_A). The other possibility is that this e_A has never appeared in the chain, so we name it as e_A^{i+1} and continue. Whenever we have an edge e_A^j, we will look at all (e_A, e_B) pairs where $e_A = e_A^j$. There are two possibilities: one is that one of e_B's has appeared in the chain, which means we have found a loop (beginning at where this e_B first appears, ending in e_A^j). The other possibility is that none of e_B's have appeared before, then we just choose one and name it as e_B^j and go on. We know that at some point we must find an e_B that has appeared before because we only have $k+1$ pairs of edges (thus $k+1$ e_B edges). The maximum length of a loop is $k+1$.

Suppose there is a loop of length n where $n \geq 3$ (we have proved for $n = 2$ in the base case). As we can see in the above chaining process, a loop does not necessarily start with e_B^1, but we use the loop as shown in Figure 3.23 in the following proof. We prove it will always lead to a contradiction. we consider the following two cases:

1. e_B^1 is below e_A^2.

   $min(e_B^1) \geq^{subtree} min(e_A^2) \geq^{V7.1} max(e_B^2)$

   If e_B^2 is below e_A^3, we have

   $\geq min(e_B^2) \geq^{subtree} min(e_A^3) \geq^{V7.1} max(e_B^3)$

   If e_B^2 is above e_A^3, we have

   $\geq^{subtree} max(e_A^3) \geq min(e_A^3) \geq^{V7.1} max(e_B^3)$

   $\ldots$

   $\cdots \geq max(e_B^n)$

   If e_B^n is below e_A^1, we have

   $\geq min(e_B^n) \geq^{subtree} min(e_A^1) \geq^{V7.1} max(e_B^1)$

   If e_B^n is above e_A^1, we have

   $\geq^{subtree} max(e_A^1) \geq min(e_A^1) \geq^{V7.1} max(e_B^1)$

   Therefore we have $min(e_B^1) \geq max(e_B^1)$, i.e., e_B^1 points to a constant. A contradiction to the condition of the theorem.

2. e_B^1 is above e_A^2.

   $max(e_A^2) \leq^{subtree} max(e_B^1) \leq^{V7.1} min(e_A^1)$

If e_B^n is above e_A^1, we have

$\leq max(e_A^1) \leq^{subtree} max(e_B^n) \leq^{V7.1} min(e_A^n)$

If e_B^n is below e_A^1, we have

$\leq^{subtree} min(e_B^n) \leq max(e_B^n) \leq^{V7.1} min(e_A^n)$

$\cdots \leq min(e_A^{n-1})$

$\cdots$

$\cdots \leq min(e_A^3)$

If e_B^2 is above e_A^3, we have

$\leq max(e_A^3) \leq^{subtree} max(e_B^2) \leq^{V7.1} min(e_A^2)$

If e_B^2 is below e_A^3, we have

$\leq^{subtree} min(e_B^2) \leq max(e_B^2) \leq^{V7.1} min(e_A^2)$

Therefore we have $max(e_A^2) \leq min(e_A^2)$, i.e., e_A^2 points to a constant. A contradiction to the condition of the theorem.

■

Note that the condition in the theorem is sufficient, but not necessary. Since a loop can start anywhere, it is easy to see that for each "position" pair of edges in the loop, every edge that is below the other in a pair points to a constant. Therefore, a tighter condition could be "for any loop, not all edges that are below some edges point to constants" instead of "none of the edges point to a constant". It is also easy to see from the above proof that if both edges in a "dominance" pair (e_A^i, e_B^i) are below some other edges in one loop, then e_A^i and e_B^i must point to the same constant.

Figure 3.24(a) shows that a loop can exist if we allow all edges that are below some edges to point to a constant. If we replace e_B^1 with 0, we reduce the value of e_A^2, thus condition V7.1 is violated for (e_A^2, e_B^2). If we replace e_B^2 with 0, we reduce the value of e_A^1, thus condition V7.1 is violated for (e_A^1, e_B^1). This example does not raise a real need for reduction because we can either reverse (e_A^2, e_B^2) then all the conditions still hold but we will not get a loop; or we can simply list one pair (e_A^1, e_B^1) and totally ignore (e_A^2, e_B^2). In both cases, we get Figure 3.24(b) as a result. But if we do replace e_B^2 with 0 in the first place, then we cannot reduce the FODD any more. Thus the example illustrates the complications that can arise. Note that this is another example where the reduced result is not unique. We can see from this example that it is important to develop some heuristics for applying R7.

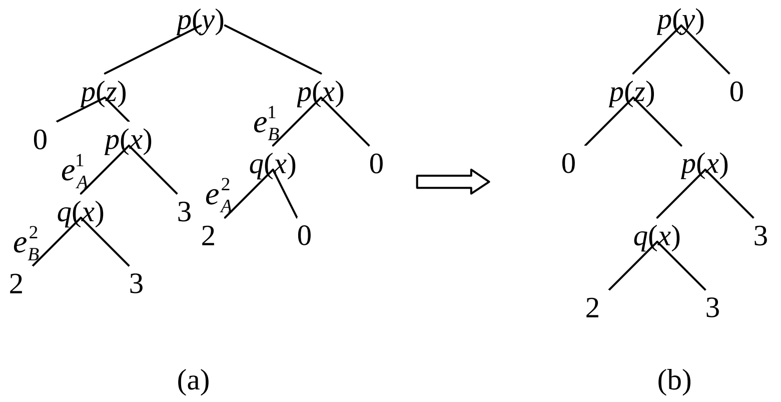


Figure 3.24: An example to illustrate that we can have a loop if we allow some edges to point to a constant.

One might hope that we can also do R7-replace(e_A, e_B) simultaneously under more sophisticated value conditions. But it is difficult, if possible, to have a similar theorem. Figure 3.25(a) shows that a loop can exist in such a situation, which will prevent us from finding an ordering of applying all these reductions sequentially. Here conditions P7.2, V7.3, and S1 hold for a set of pairs $(e_A^1, e_B^1), (e_A^2, e_B^2), (e_A^3, e_B^3)$ and we have a loop $(e_B^2, e_A^1), (e_B^1, e_A^2)$. If we replace e_B^1 with 0, V7.3 will be violated for (e_A^2, e_B^2). If we replace e_B^2 with 0, V7.3 will be violated for (e_A^1, e_B^1). However we can apply R7-drop(e_A^3, e_B^3) and get Figure 3.25(b). We can do one more reduction and get Figure 3.25(c). Note that the new pair of edges (e_A^1, e_B^1) as shown in Figure 3.25(b) was not there before.

3.3.4 Comparing R6 and R7

In this section we show that R6 is in fact a special case of R7. Let $p = source(e_1)$ and $q = source(e_2)$. R6 is a special case of R7 when $p = q$, i.e., e_1 and e_2 are siblings. Let $e_1 = p_{\downarrow t}$ and $e_2 = p_{\downarrow f}$, then all the conditions in R7 become the following:

P7.1 requires that $\mathcal{B} \models [\exists \vec{x}, \mathrm{EF}(p_{\downarrow f})] \rightarrow [\exists \vec{y}, \mathrm{EF}(p_{\downarrow t})]$, which is P6.2 in R6.

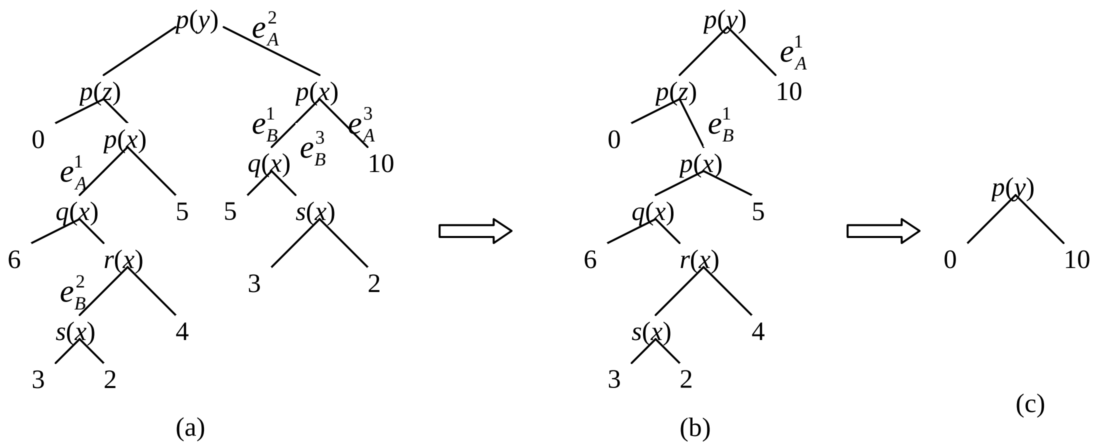


Figure 3.25: An example to illustrate that we can have a loop under more complex value conditions.

V7.1 requires that $\min(p_{\downarrow t}) \geq \max(p_{\downarrow f})$, which is V6.1 in R6.

P7.2 requires that $\mathcal{B} \models \forall \vec{u}, [[\exists \vec{w}, \mathrm{EF}(p_{\downarrow f})] \rightarrow [\exists \vec{v}, \mathrm{EF}(p_{\downarrow t})]]$. Note that $\vec{u}$ now becomes the intersection of variables in $p_{\downarrow t}$ and $p_{\downarrow f}$, therefore this condition is the same as P6.5 in R6.

P7.3 requires that $\mathcal{B} \models \forall \vec{r}, [[\exists \vec{s}, \mathrm{EF}(p_{\downarrow f})] \rightarrow [\exists \vec{t}, \mathrm{EF}(p_{\downarrow t})]]$. Note that $\vec{r}$ are the variables that appear in both $target(e_1)$ and $target(sibling(e_2))$, and in this case, $sibling(e_2) = e_1$, therefore $\vec{r} = var(target(e_1)) = var(p_{\downarrow t})$. This condition is the same as P6.3 in R6.

S1 requires that $\mathrm{NF}(p_{\downarrow t})$ and sub-FODD of $p_{\downarrow t}$ remain the same before and after the replacement with 0, which is always true in this case.

V7.2 requires that $\min(p_{\downarrow t}) \geq \max(p_{\downarrow t})$, which means that $p_{\downarrow t}$ is a constant.

V7.3 requires that all leaves in $D = p_{\downarrow t} - p_{\downarrow f}$ have non-negative values, which is the same as condition V6.2 in R6.

V7.4 requires that all leaves in $G = p_{\downarrow t} - p_{\downarrow t}$ have non-negative values, which is always true.

In R7, we can replace a branch with 0 if (P7.1 $\wedge$ V7.1) $\vee$ (P7.2 $\wedge$ V7.3) holds. For the special applications as R6 this is translated to (P6.2 $\wedge$ V6.1) $\vee$ (P6.5 $\wedge$ V6.2), which is exactly the same as in the conditions for replacing in R6. In R7, we can remove a node if we can replace and (V7.2) $\vee$ (P7.3 $\wedge$ V7.4). For R6, this is translated to $p_{\downarrow t}$ is a constant (if this holds, then P6.3 holds) or P6.3 (in addition to (P6.2 $\wedge$ V6.1) $\vee$ (P6.5 $\wedge$ V6.2)), this is exactly what R6 says about dropping the node.

3.3.5 (R8) Weak Reduction by Unification

Consider a FODD B. Let $\vec{v}$ denote its variables, and let $\vec{x}$ and $\vec{y}$ disjoint subsets of $\vec{v}$, which are of the same cardinality. We define the operator R8-unify(B, x, y) as replacing variables in $\vec{x}$ by the corresponding variables in $\vec{y}$. We denote the resulting FODD by $B\{\vec{x}/\vec{y}\}$ and clearly it only has variables in $\vec{v} \setminus \vec{x}$.

We first present the condition, followed by lemmas regarding this operator.

(V8) : all leaves in $B\{\vec{x}/\vec{y}\} - B$ are non negative.

Lemma 10 *Let B be a FODD, B' the result of R8-unify$(B, \vec{x}, \vec{y})$ for which V8 holds, then for any interpretation I we have $MAP_B(I) = MAP_{B'}(I)$.*

Proof: Consider any valuation ζ_1 to $\vec{v}$ in B. By V8, $B\{\vec{x}/\vec{y}\}$ gives a better value on the same valuation. Therefore we do not lose any value by this operator. We also do not gain any extra value. Consider any valuation ζ_2 to variables in B' reaching a leaf node with value v, we can construct a valuation ζ_3 to $\vec{v}$ in B with all variables in $\vec{x}$ taking the corresponding value in $\vec{y}$, and it will reach a leaf node in B with the same value. Therefore the map will not be changed by unification. ■

In some cases, R6 is not directly applicable because variables are used below the node n. Figure 3.26 gives an example. Here R6 does not apply because x_2 is used below. But with $\{x_1/x_2\}$, we can get a FODD as shown in Figure 3.26(b). Since $(b) - (a) \geq 0$, (b) becomes the result after reduction. If we unify in the other way, i.e.,$\{x_2/x_1\}$, we will get Figure 3.26(c), it is isomorphic to Figure 3.26(b), but we cannot reduce the original FODD to this result, because $(c) - (a) \not\geq 0$. This phenomenon happens since all the reductions use propositional apply and are therefore sensitive to variable names.

We can also see that if we allow to use reordering in R6, we can first reorder the nodes in Figure 3.26(a) so that $p(x_1)$ is placed below $p(x_2)$. Then we can use R6 to remove node $p(x_1)$.

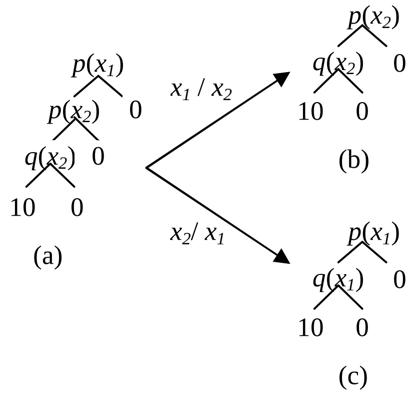


Figure 3.26: An example illustrating R8.

3.4 Summary

In this chapter we developed first order decision diagrams, their semantics, and reduction operators for them. As mentioned above one of the important properties of propositional ADDs is that given a variable ordering a formula has a unique representation that serves as a normal form for the formula, and this is useful in the context of theorem proving. While we have developed reduction operators, we have not established such a normal form for FODDs, so this remains an important open question. We cannot guarantee in advance that our reduction operators will be applicable and useful in a particular domain, but as the following chapters illustrate they can substantially reduce the size of diagrams used in value iteration and policy iteration.

Chapter 4

Decision Diagrams for MDPs

In this chapter we describe how to represent relational MDPs with FODDs. We follow Boutilier et al. (2001) and specify stochastic actions as a randomized choice among deterministic alternatives. We therefore use FODDs to represent the domain dynamics of deterministic action alternatives, probabilities, rewards, and value functions. We also discuss the expressiveness of the representation and give a procedure of translating into FODDs problem descriptions in PPDDL (Younes & Littman, 2004), which has become a standard language used to encode probabilistic planning domains and problems.

4.1 Example Domain

We first give a concrete formulation of the logistics problem discussed in the introduction. This example follows exactly the details given by Boutilier et al. (2001), and is used to illustrate our constructions for MDPs. The domain includes boxes, trucks and cities, and predicates are $Bin(Box, City)$, $Tin(Truck, City)$, and $On(Box, Truck)$. Following Boutilier et al. (2001), we assume that $On(b, t)$ and $Bin(b, c)$ are mutually exclusive, so a box on a truck is not in a city. That is, our background knowledge includes statements $\forall b, c, t, On(b, t) \rightarrow \neg Bin(b, c)$ and $\forall b, c, t, Bin(b, c) \rightarrow \neg On(b, t)$. The reward function, capturing a planning goal, awards a reward of 10 if the formula $\exists b, Bin(b, Paris)$ is true, that is if there is any box in Paris. Thus the reward is allowed to include constants but need not be completely ground.

The domain includes 3 actions *load*, *unload*, and *drive*. Actions have no effect if their preconditions are not met. Actions can also fail with some probability. When attempting *load*, a successful version *loadS* is executed with probability 0.99, and an unsuccessful version *loadF* (effectively a no-operation) with probability 0.01. The drive action is executed deterministically. When attempting *unload*, the probabilities depend on whether it is raining or not. If it is not raining then a successful version *unloadS* is executed with probability 0.9, and *unloadF* with probability 0.1. If it is raining *unloadS* is executed with probability 0.7, and *unloadF* with probability 0.3.

4.2 The Domain Dynamics

The domain dynamics are defined by *truth value diagrams* (TVDs). For every action schema $A(\vec{a})$ and each predicate schema $p(\vec{x})$ the TVD $T(A(\vec{a}), p(\vec{x}))$ is a FODD with $\{0, 1\}$ leaves. The TVD gives the truth value of $p(\vec{x})$ in the next state when $A(\vec{a})$ has been performed in the current state. We call $\vec{a}$ action parameters, and $\vec{x}$ predicate parameters. No other variables are allowed in the TVD. We will explain why we need this restriction in Section 5.1.1. The truth value is valid when we fix a valuation of the parameters.

Notice that the TVD simultaneously captures the truth values of all instances of $p(\vec{x})$ in the next state. Notice also that TVDs for different predicates are separate and independent. This can be safely done even if an action has coordinated effects (not conditionally independent) since the actions are deterministic.

For any domain, a TVD for predicate $p(\vec{x})$ can be defined generically as in Figure 4.1. The idea is that the predicate is true if it was true before and is not "undone" by the action or was false before and is "brought about" by the action. TVDs for the logistics domain in our running example are given in Figure 4.2. All the TVDs omitted in the figure are trivial in the sense that the predicate is not affected by the action. In order to simplify the presentation we give the TVDs in their generic form and did not sort the diagrams. Notice in the examples that the TVDs capture the implicit assumption usually taken in such planning-based domains that if the preconditions of the action are not satisfied then the action has no effect.

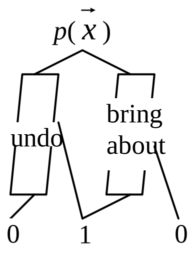


Figure 4.1: A template for the TVD

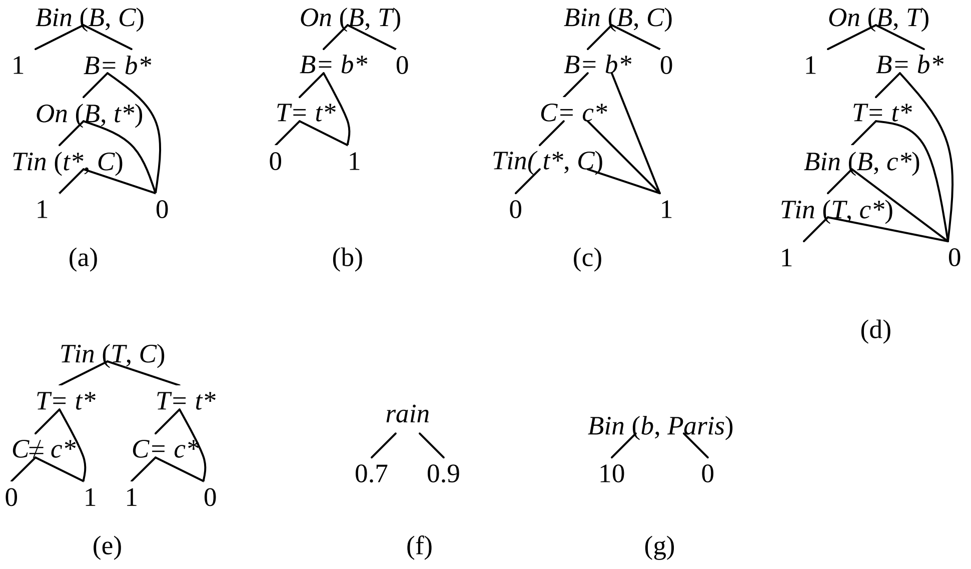


Figure 4.2: FODDs for logistics domain: TVDs, action choice, and reward function. (a)(b) The TVDs for $Bin(B, C)$ and $On(B, T)$ under action choice $unloadS(b^*, t^*)$. (c)(d) The TVDs for $Bin(B, C)$ and $On(B, T)$ under action choice $loadS(b^*, t^*, c^*)$. Note that c^* must be an action parameter so that (d) is a valid TVD. (e) The TVD for $Tin(T, C)$ under action choice $driveS(t^*, c^*)$. (f) The probability FODD for the action choice $unloadS(b^*, t^*)$. (g) The reward function.

Notice how we utilize the multiple path semantics with maximum aggregation. A predicate is true if it is true according to one of the paths specified so we get a disjunction over the conditions for free. If we use the single path semantics the corresponding notion of TVD is significantly more complicated since a single path must capture all possibilities for a predicate to become true. To capture that we must test sequentially for different conditions and then take a union of the substitutions from different tests and in turn this requires additional annotation on FODDs with appropriate semantics. Similarly an OR operation would require union of substitutions, thus complicating the representation. We explain these issues in more detail in Section 5.1.1 after we introduce the first order value iteration algorithm.

4.3 Probabilistic Action Choice

One can consider modeling arbitrary conditions described by formulas over the state to control nature's probabilistic choice of action. Here the multiple path semantics makes it hard to specify mutually exclusive conditions using existentially quantified variables and in this way specify a distribution. We therefore restrict the conditions to be either propositional or depend directly on the action parameters. Under this condition any interpretation follows exactly one path (since there are no variables and thus only the empty valuation) thus the aggregation function does not interact with the probabilities assigned. A diagram showing action choice for *unloadS* in our logistics example is given in Figure 4.2. In this example, the condition is propositional. The condition can also depend on action parameters, for example, if we assume that the result is also affected by whether the box is big or not, we can have something as in Figure 4.3 regarding the action choice probability of *unloadS*.

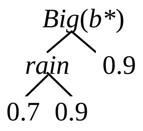


Figure 4.3: An example showing that the choice probability can depend on action parameters.

Note that a probability usually depends on the current state. It can depend on arbitrary properties of the state (with the restriction stated as above), e.g., *rain* and $big(b^*)$, as shown in Figure 4.3. We allow arbitrary conditions that depend on predicates with arguments restricted to action parameters so the dependence can be complex. However,

we do not allow any free variables in the probability choice diagram. For example, we cannot model a probabilistic choice of $unload(b^*, t^*)$ that depends on other boxes on the truck t^*, e.g.,
$\exists b, On(b, t^*) \wedge b \neq b^* : 0.2$
otherwise, 0.7.
While we can write a FODD to capture this condition, the semantics of FODD means that a path to 0.7 will be selected by max aggregation so the distribution cannot be modeled in this way. While this is clearly a restriction, the conditions based on action arguments still give a substantial modeling power.

4.4 Reward and Value Functions

Reward and value functions can be represented directly using algebraic FODDs. The reward function for our logistics domain example is given in Figure 4.2.

4.5 PPDDL and FODDs

In terms of expressiveness, our approach can easily capture probabilistic STRIPS style formulations as in ReBel, allowing for more flexibility since we can use FODDs to capture rewards and transitions. However, it is more limited than SDP since we cannot use arbitrary formulas for rewards, transitions, and probabilistic choice. For example we cannot express universal quantification using maximum aggregation. Our approach can also easily capture traditional maze-like RL domains with state based reward (which are propositional) in factored form since the reward can be described as a function of location. In general, it seems that one may be able to get around representation restrictions in some cases by modifying the set of base predicates.

Probabilistic Planning Domain Definition Language (PPDDL) (Younes & Littman, 2004) has become a standard language for describing probabilistic and decision theoretic planning problems. It has been used to describe probabilistic planning domains and problems since the fourth International Planning Competition (IPC-4) held as part of the

International Conference on Planning and Scheduling (ICAPS'04) in which probabilistic planning track was introduced (Younes et al., 2005). PPDDL is an extension of the Planning Domain Definition Language (PDDL) (McDermott, 2000) with additional constructs to support the modeling of probabilistic effects and decision-theoretic concepts, e.g. rewards. PPDDL can be considered a probabilistic STRIPS-like language but with richer constructs. In this section we sketch a procedure of generating TVDs from PPDDL action descriptions. Note that our representation is more restricted than general PPDDL, e.g., we do not support functions or universal quantification.

4.5.1 PPDDL Action Effects

An action in PPDDL, $a \in A$, consists of a precondition ϕ_a and effect e_a. We support the following action effects (which are also rules to recursively construct action effects).

1. Simple effects: null-effect $\top$, and atomic formula or its negation, i.e., $p(\vec{x})$ and $\neg p(\vec{x})$ where p is a predicate and $\vec{x}$ are predicate parameters.
2. Conjunction of effects: $e_1 \wedge \cdots \wedge e_n$ where e_i is an effect.
3. Conditional effect: $c \rhd e$ where c is a formula over predicates, dictating the condition for the effect e to take place.
4. Probabilistic effects: $p_1 e_1 \mid \ldots \mid p_n e_n$ meaning that the effect e_i occurs with probability p_i. If $\sum_{i=1}^{n} p_i < 1$, we have a default effect *null effect* with probability $p_{i+1} = 1 - \sum_{i=1}^{n} p_i$.

For example the following is an action description for *move-left* in the elevator domain in the probabilistic track of the Fifth International Planning Competition (IPC-5) held in ICAPS'06 (Gerevini et al., 2006). This domain is about an agent moving around to collect coins scattered in different positions in different floors. The agent can take the elevator to move up and down. It can also move horizontally to the left or the right but risk ending up in some place (i.e., position $p1$ in floor $f1$ as shown below) that is not the destination. Consider for example the following PPDDL description (where predicates are written in prefix notation):

```
(:action move-left
 :parameters (?f - floor ?p ?np - pos)
 :precondition (and (at ?f ?p) (dec_p ?p ?np)
 :effect (probabilistic 0.5 (and (not at ?f ?p))(at ?f ?np))
                        0.5 (and (not (at ?f ?p))
                                 (when (gate ?f ?p)(at f1 p1))
                                 (when (not (gate ?f ?p)) (at ?f ?np))))
)
```

In this description $f1$ is a constant of type floor and $p1$ is a constant of type position. The symbols starting with question marks denote variables. The precondition dictates that the agent must initially be in the starting location and the destination position must be immediately to the left of its current position. The effect equation says that with probability 0.5 the agent will successfully land at the destination position and thus no longer be in its previous position. But with probability 0.5, although the agent will not be in previous position, where it actually lands on depends on whether the condition (gate ?f ?p) is true. If it is true, the agent will be at an unexpected place (i.e., position $p1$ in floor $f1$); otherwise, it will be in the destination.

Note that PPDDL allows for more complicated action effects, e.g., universal effects or associating a reward with a state transition. Rewards associated with state transitions can be easily incorporated into our framework by using a more general reward model $R(s, a)$. On the other hand we cannot handle universal effects without grounding the domain.

4.5.2 Transformations of Action Effects

PPDDL allows arbitrary nesting of conditional and probabilistic effects thus can be very complex. Note that TVDs are defined for deterministic action alternatives and should not have any probabilistic effects inside. Rintanen (2003) has proved that any action effect e constructed by Rules in Section 4.5.1 can be translated into the form $p_1e_1 \mid \ldots \mid p_ne_n$ where e_i is a deterministic effect by using the following equivalences. Rintanen (2003) called this form Unary Nondeterminism Normal Form(1ND)

$$c \rhd (e_1 \wedge \cdots \wedge e_n) \equiv (c \rhd e_1) \wedge \cdots \wedge (c \rhd e_n)$$

$$c \rhd (c' \rhd e) \equiv (c \wedge c') \rhd e$$

$$c \rhd (p_1 e_1 \mid \ldots \mid p_n e_n) \equiv p_1(c \rhd e_1) \mid \ldots \mid p_n(c \rhd e_n)$$

$$e \wedge (p_1 e_1 \mid \ldots \mid p_n e_n) \equiv p_1(e \wedge e_1) \mid \ldots \mid p_n(e \wedge e_n)$$

$$p_1(p'_1 e'_1 \mid \ldots \mid p'_k e'_k) \mid p_2 e_2 \mid \ldots \mid p_n e_n \equiv (p_1 p'_1) e'_1 \mid \ldots \mid (p_1 p'_k) e'_k \mid p_2 e_2 \mid \ldots \mid p_n e_n$$

Rintanen (2003) showed that the first three equivalences are used to move the conditionals inside so that their consequents are atomic effects, and the last two equivalences are applied to the intermediate results to get the effect of the 1ND form. With this transformation we decompose a stochastic action a into a set of deterministic action alternatives a_i, one for each action effect e_i, and the probability this action alternative gets chosen is p_i.

Note that now each effect e_i has the form $(c_{i1} \rhd e_{i1}) \wedge \ldots \wedge (c_{in_i} \rhd e_{in_i})$ and each e_{ij} is a conjunction of simple effects. In order to facilitate the translation of action effects into TVDs, we would like to have conditions $c_{i1}, \cdots, c_{in_i}$ mutually exclusive and collectively exhaustive. We use the following equivalences (Younes & Littman, 2004) to accomplish this.

$$e \equiv \top \rhd e$$

$$c \rhd e \equiv (c \rhd e) \wedge (\neg c \rhd \top)$$

$$(c_1 \rhd e_1) \wedge (c_2 \rhd e_2) \equiv ((c_1 \wedge c_2) \rhd (e_1 \wedge e_2)) \wedge ((c_1 \wedge \neg c_2) \rhd e_1) \wedge ((\neg c_1 \wedge c_2) \rhd e_2) \wedge ((\neg c_1 \wedge \neg c_2) \rhd \top)$$

Both transformations can result in exponential increases in the size of the effect formula (Rintanen, 2003; Younes & Littman, 2004). Later we will give a special case where we may have exponential savings compared to the result obtained by performing these transformations directly.

We can use the equivalences described above to translate the action effect of *move-left* into the desired form. For the sake of uniformity with the syntactic format used in the rest of the thesis, we do not include question marks before a variable, and we use parenthesis around predicate parameters.

$0.5(\top \rhd (\neg at(f,p) \wedge at(f,np)))\ |$

$0.5(gate(f,p) \rhd (\neg at(f,p) \wedge at(f1,p1)) \wedge (\neg gate(f,p) \rhd (\neg at(f,p) \wedge at(f,np))$

Therefore we will have two deterministic alternatives for this action, denoted as move-left-outcome1 (*move-left* succeeds) and move-left-outcome2 (*move-left* has conditional effects). For move-left-outcome1, it would be more straightforward to have just $\neg at(f,p) \wedge at(f,np)$. I.e., if an action alternative has only a conjunction of simple effects then we keep it as is.

4.5.3 Translating Action Effects into FODDs

The following is a sketch of the procedure to translate an effect (of a deterministic action alternative) of the form $(c_{i1} \rhd e_{i1}) \wedge \ldots \wedge (c_{in_i} \rhd e_{in_i})$ into a set of TVDs, one TVD for each predicate $p(\vec{x})$. Note that e_{ij} is a conjunction of simple effects and conditions are mutually exclusive and collectively exhaustive.

Procedure 2 *1. Initialize the TVD for each predicate as shown in Figure 4.4(a). k=1.*

2. For the conditional effect $c_{ik} \rhd e_{ik}$, for each predicate $p(\vec{x})$, do the following:

(a) If the predicate does not appear in the effect, we leave the TVD as is.

(b) If the predicate appears as the positive effect $p()$, let t be the number of times $p()$ (with different set of predicate parameters) appears in e_{ik}. We use $\vec{v}_j^{ik}$ where $j \leq t$ to denote predicate parameters in one $p()$. We construct a sub-FODD as follows, which is illustrated in Figure 4.4(b):

i. The root node of this sub-FODD is c_{ik}.

ii. $(c_{ik})_{\downarrow f}$ is left "open".

iii. $(c_{ik})_{\downarrow t}$ *is linked to a sequence of predicate parameters equality tests starting from* $\vec{x} = \vec{v}_1^{ik}$. *For each* $j \in \{1 \cdots m-1\}$, *create a test for predicate parameters equality* $\vec{x} = \vec{v}_j^{ik}$, *and set* $(\vec{x} = \vec{v}_j^{ik})_{\downarrow t} = 1$ *and* $(\vec{x} = \vec{v}_j^{ik})_{\downarrow f}$ *linked to the test for* $\vec{x} = \vec{v}_{j+1}^{ik}$. *For* $j = m$ *the test is* $\vec{x} = \vec{v}_m^{ik}$ *and we set* $(\vec{x} = \vec{v}_m^{ik})_{\downarrow t} = 1$ *and* $(\vec{x} = \vec{v}_m^{ik})_{\downarrow f} = 0$.

In this case we need to change the "bring about" branch. We link such a sub-FODD directly to $p(\vec{x})_{\downarrow f}$ *if* $p(\vec{x})_{\downarrow f} = 0$, *or to an "open" exit otherwise. We illustrate these two cases in Figure 4.4(c)and (d) respectively.*

(c) *If the predicate appears as the negative effect* $\neg p()$, *we need to change the "undo" branch. The construction is symmetric to the above and is illustrated in Figure 4.4(e), (f), and (g), which correspond to constructing a sub-FODD to test predicate parameters equality, the case when* $p(\vec{x})_{\downarrow f} = 1$, *and the case when there is an "open" exit respectively.*

3. *Set* $k = k + 1$ *(go to the next conditional effect). If* $k < n_i$ *go to 2.*

4. *Note that here we handle the last condition-effect pair.*

(a) *If the predicate does not appear in the effect and if the TVD has "open" exit, we close it by linking the exit to* 0 *in the sub-FODD of* $p(\vec{x})_{\downarrow f}$, *and* 1 *in* $p(\vec{x})_{\downarrow t}$.

(b) *If the predicate appears as the positive effect* $p()$, *and if all the other conditions have appeared in the sub-FODD of* $p(\vec{x})_{\downarrow f}$, *we link the "open" exit with the sub-FODD for testing a sequence of predicate parameters equality (all the conditions are mutually exclusive and exhaustive, therefore the last condition is implied by the conjunction of the negations of all the other conditions). This is illustrated in Figure 4.4(h). If not all conditions have appeared in the* $p(\vec{x})_{\downarrow f}$, *we still need to test this condition, followed by test for parameters equality. This is illustrated in Figure 4.4(i).*

(c) *If the predicate appears as the negative effect* $p()$, *the construction is symmetric to the above.*

5. *As the last step, we need to add the precondition ϕ_a to the TVDs of those predicates with "bring about" or "undo" branches, as illustrate by Figure 4.4 (j). In "bring about" branch $(\phi_a)_{\downarrow f} = 0$, and in "undo" branch $(\phi_a)_{\downarrow f} = 1$.*

Figure 4.5 shows the TVD of the predicate *at* for move-left-outcome2. First note that we deal with the precondition differently than (P)PDDL. In (P)PDDL it is considered an error to apply an action in a state where the precondition does not hold . However we consider an action as *no-op* if the precondition does not hold. Also note that strong reductions and sorting have to be applied to get the final TVD. In this example The final diagram is obtained after a neglect reduction to remove the unnecessary condition $gate(f, p)$ in $at(F, P)_{\downarrow t}$.

Note that currently we only handle conditions (e.g., preconditions or conditions in conditional effect) as a conjunction of atomic formulas and equalities. Figure 4.6 illustrates how to translate a condition $C = c_1 \wedge c_2 \cdots \wedge c_n$ into FODD components.

4.5.4 A Special Case in Translation

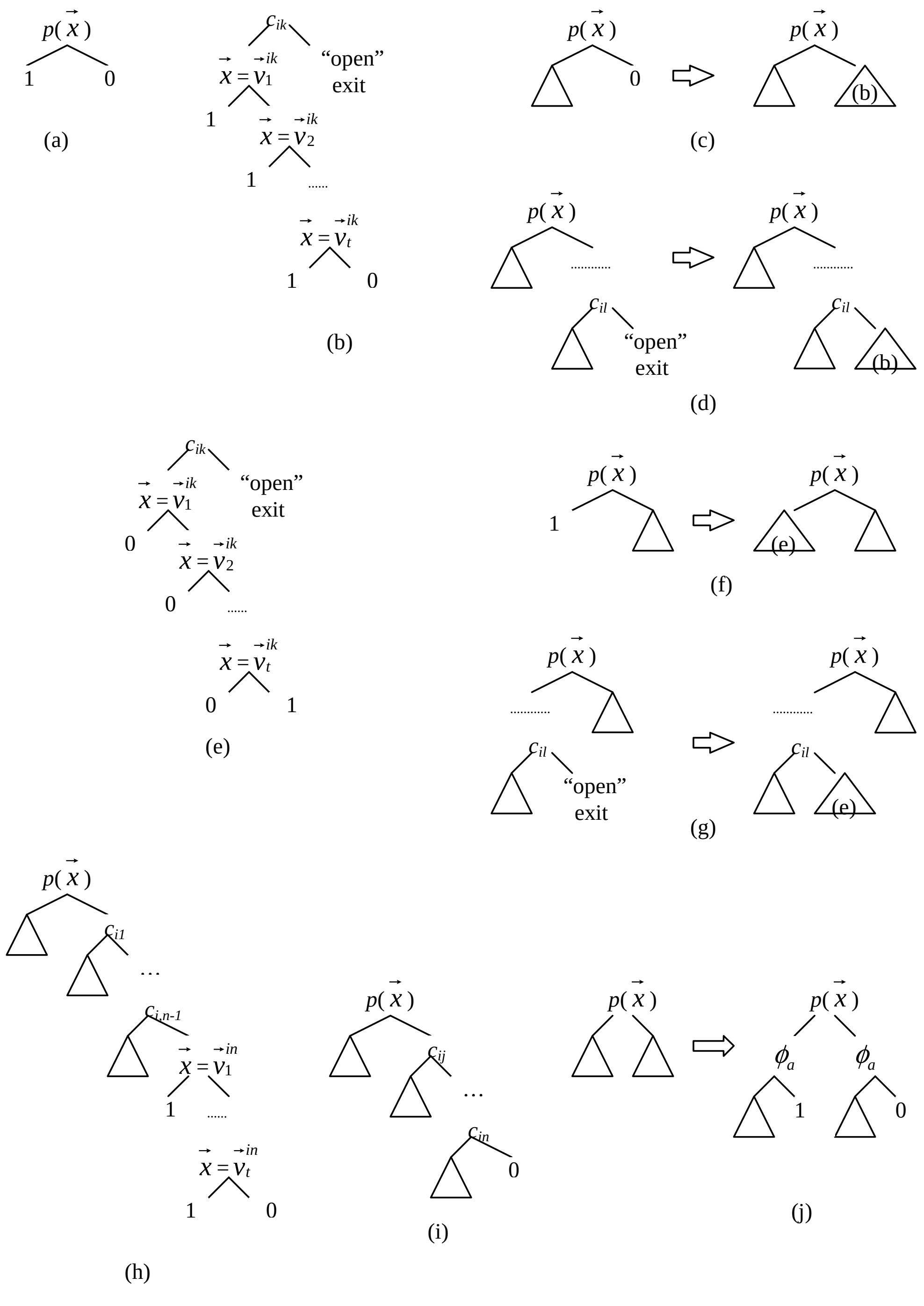
Recall that the action effect transformations may cause exponential size increases. Here we give a special case where our representation can provide more compact encoding even compared with the PPDDL description. If we have a probabilistic effect in the following form:

$$(p_{11}(c_1 \rhd e_1) \mid \ldots \mid p_{1n}(c_1 \rhd e_n)) \wedge$$

$$\cdots \wedge$$

$$p_{m1}(c_m \rhd e_1) \mid \ldots \mid p_{mn}(c_m \rhd e_n)$$

where e_i is a conjunction of simple effects, and $c_1 \ldots c_m$ are mutually exclusive and exhaustive conditions. Notice repetitive structure where c_i repeats in row and e_i in column. If we use the transformation rules, we may end up with n^m deterministic action alternatives. In fact, we can just have n deterministic alternatives, one for each e_i, and use the choice



$p(\vec{x})$
1
0
(a)
c_{ik}
$\vec{x} = \vec{v}_1^{ik}$
"open" exit
$\vec{x} = \vec{v}_2^{ik}$
$\vec{x} = \vec{v}_t^{ik}$
(b)
(c)
c_{il}
(d)
(e)
(f)
(g)
c_{i1}
$c_{i,n-1}$
$\vec{x} = \vec{v}_1^{in}$
$\vec{x} = \vec{v}_t^{in}$
(h)
c_{ij}
c_{in}
(i)
ϕ_a
(j)

Figure 4.4: Examples illustrating translating action effects in PPDDL into TVDs.

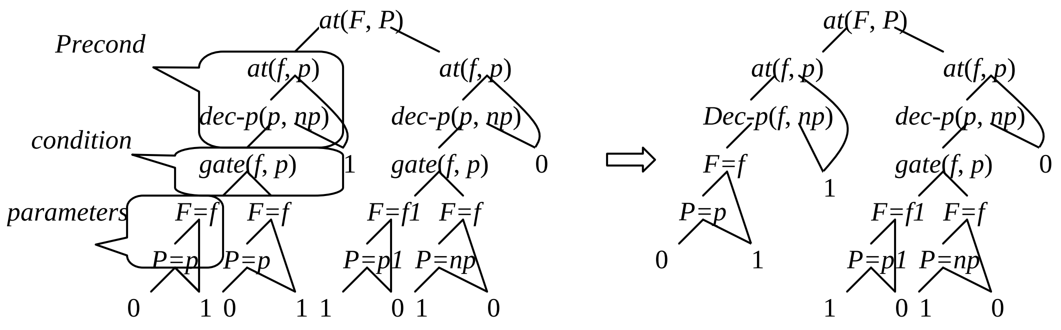


Figure 4.5: The TVD for *at* under action choice *move-left-outcome*2 in the elevator domain.

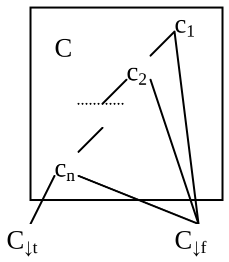


Figure 4.6: Translating a condition into FODD components.

probabilities to capture all the conditions. For example, for the first action alternative corresponding to e_1, we have the choice probability defined as shown in Figure 4.7. Note that we do not need to include the last condition c_m because $c_m = \neg c_1 \wedge \cdots \wedge \neg c_{m-1}$.

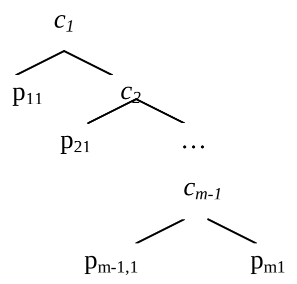


Figure 4.7: The choice probability for the action alternative corresponding to e_1.

The following is an action description for *look-at-light* in the drive domain in the probabilistic track of the Fifth International Planning Competition (Gerevini et al., 2006).

```
(:action look-at-light
    :parameters (?x - coord ?y - coord)
    :precondition (and
             (light_color unknown)
             (at ?x ?y)
                 )
     :effect (and
         (probabilistic
            9/10
            (when (and (heading north) (light_preference ?x ?y north_south))
                   (and (not (light_color unknown))(light_color green)))
            1/10
            (when (and (heading north) (light_preference ?x ?y north_south))
                   (and (not (light_color unknown))(light_color red))))
         (probabilistic
            9/10
            (when (and (heading south) (light_preference ?x ?y north_south))
                   (and (not (light_color unknown))(light_color green)))
            1/10
            (when (and (heading south) (light_preference ?x ?y north_south))
                   (and (not (light_color unknown))(light_color red))))
         (probabilistic
            1/10
            (when (and (heading east) (light_preference ?x ?y north_south))
                   (and (not (light_color unknown))(light_color green)))
```

```
   9/10
   (when (and (heading east) (light_preference ?x ?y north_south))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   1/10
   (when (and (heading west) (light_preference ?x ?y north_south))
          (and (not (light_color unknown))(light_color green)))
   9/10
   (when (and (heading west) (light_preference ?x ?y north_south))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   1/10
   (when (and (heading north) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color green)))
   9/10
   (when (and (heading north) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   1/10
   (when (and (heading south) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color green)))
   9/10
   (when (and (heading south) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   9/10
   (when (and (heading east) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color green)))
   1/10
   (when (and (heading east) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   9/10
   (when (and (heading west) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color green)))
   1/10
   (when (and (heading west) (light_preference ?x ?y east_west))
          (and (not (light_color unknown))(light_color red))))
(probabilistic
   1/2
   (when (light_preference ?x ?y none)
          (and (not (light_color unknown))(light_color green)))
   1/2
   (when (light_preference ?x ?y none)
          (and (not (light_color unknown))(light_color red)))))
```

)

Intuitively *look-at-light* is a sensing action to sense the light color. Therefore the precondition is that the light color is unknown and the agent is at the right spot. It only has two outcomes. One is that the agent senses light color to be green and we denote the outcome as *Lc-green*, and other is that the agent senses light color to be red and we denote this outcome as *Lc-red*. Each outcome is associated with different probabilities under different conditions. Figure 4.8 gives the choice probability for the action alternative *Lc-green*. Note that, as a heuristic, we start with the condition of shortest length, i.e., we start with the condition *light-preference* being none.

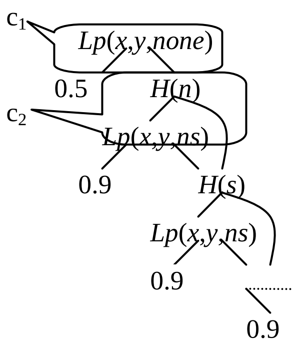
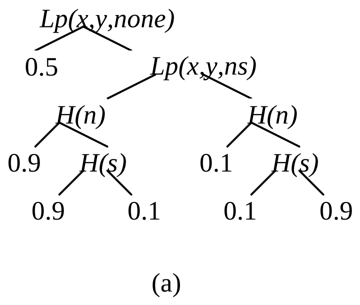


Figure 4.8: The choice probability for the action alternative *Lc-green* before reduction and sorting. *Lp* denotes light-preference, *H* heading, n, s, ns and e north, south, north-south, and east respectively.

.

Figure 4.9 shows the choice probability after reduction and sorting, and the TVD for *light-color*(abbreviated as LC) under the action alternative *Lc-green*.

We translated action descriptions in several domains in the probabilistic track planning problems in IPC-5 and we have found that TVDs and choice probabilities represented as FODDs provide a concise and natural encoding of action effects.



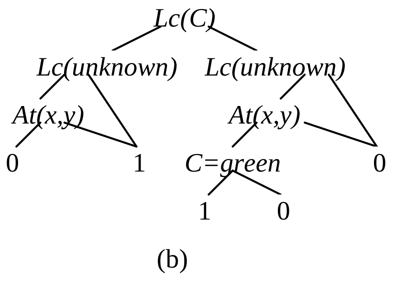


Figure 4.9: (a) The choice probability for the action alternative *Lc-green*. (b)The TVD for *light-color* under the action alternative *Lc-green*

.

Chapter 5

Value Iteration and Policy Iteration with FODDs

In this chapter, we present our algorithms for RMDPs. We show how value iteration can be performed using FODDs. We also introduce a new algorithm, Relational Modified Policy Iteration. We point out two anomalies of policy languages in the context of policy evaluation and show that our algorithm incorporates an aspect of policy improvement into policy evaluation. We further provide the proof that the algorithm converges to the optimal value function and policy.

5.1 Value Iteration with FODDs

The general first order value iteration algorithm (Boutilier et al., 2001) works as follows: given the reward function R and the action model as input, we set $V_0 = R, n = 0$ and repeat the procedure *Rel-greedy* until termination:

Procedure 3 *Rel-greedy*

1. *For each action type $A(\vec{x})$, compute:*

$$Q_{V_n}^{A(\vec{x})} = R \oplus [\gamma \otimes \oplus_j (prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x})))] \tag{5.1}$$

2. $Q^A_{V_n} = obj\text{-}max(Q^{A(\vec{x})}_{V_n})$.

3. $V_{n+1} = \max_A Q^A_{V_n}$.

The notation and steps of this procedure were discussed in Section 2.3 except that now $\otimes$ and $\oplus$ work on FODDs instead of case statements. Note that since the reward function does not depend on actions, we can rewrite the procedure as follows:

Procedure 4 *Rel-greedy*

1. For each action type $A(\vec{x})$, compute:

$$T^{A(\vec{x})}_{V_n} = \oplus_j (prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x}))) \tag{5.2}$$

2. $Q^A_{V_n} = R \oplus \gamma \otimes obj\text{-}max(T^{A(\vec{x})}_{V_n})$.

3. $V_{n+1} = \max_A Q^A_{V_n}$.

Later we will see that the object maximization step makes more reductions possible; therefore by moving this step forward before adding the reward function we may get some savings in computation. We compute the updated value function in this way in the comprehensive example of value iteration given later in Section 5.1.5.

Value iteration terminates when $\|V_{i+1} - V_i\| \leq \frac{\varepsilon(1-\gamma)}{2\gamma}$, i.e., when each leaf node of the resulting FODD from subtraction operation has a value less than the threshold (Puterman, 1994).

Some formulations of goal based planning problems use an absorbing state with zero additional reward once the goal is reached. We can handle this formulation when there is only one non-zero leaf in R. In this case, we can replace Equation 5.1 with $Q^{A(\vec{x})}_{V_n} = max(R, \gamma \otimes \oplus_j (prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x}))))$. Note that, due to discounting, the max value is always $\leq R$. If R is satisfied in a state we do not care about the action (max would be R) and if R is 0 in a state we get the value of the discounted future reward.

Note that we can only do this in goal based domains, i.e., there is only one non-zero leaf. This does not mean that we cannot have disjunctive goals, but it means that we must value each goal condition equally. For example, this method does not apply when we have complex reward functions, e.g., the one as shown in Figure 5.1.

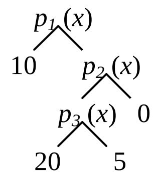


Figure 5.1: An example showing a complex reward function.

5.1.1 Regression by Block Replacement

We first describe the calculation of $Regr(V_n, A_j(\vec{x}))$ using a simple idea we call block replacement. We then proceed to discuss how to obtain the result efficiently.

Consider V_n and the nodes in its FODD. For each such node take a copy of the corresponding TVD, where predicate parameters are renamed so that they correspond to the node's arguments and action parameters are unmodified. BR-regress$(V_n, A(\vec{x}))$ is the FODD resulting from replacing each node in V_n with the corresponding TVD, with outgoing edges connected to the 0, 1 leaves of the TVD.

Let s denote a state resulting from executing an action $A(\vec{x})$ in $\hat{s}$. Notice that V_n and BR-regress$(V_n, A(\vec{x}))$ have exactly the same variables. We have the following lemma:

Lemma 11 *Let ζ be any valuation to variables of V_n (and thus also variables of BR-regress$(V_n, A(\vec{x}))$), $MAP_{V_n}(s,\zeta) = MAP_{BR-regress(V_n,A(\vec{x}))}(\hat{s},\zeta)$*

Proof: Consider the paths $P, \hat{P}$ followed under the valuation ζ in the two diagrams. By the definition of TVDs, the sub-paths of $\hat{P}$ applied to $\hat{s}$ guarantee that the corresponding nodes in P take the same truth values in s. So $P, \hat{P}$ reach the same leaf and the same value is obtained. ■

Now we can explain why we cannot have variables in TVDs through an example illustrated in Figure 5.2. Suppose we have a value function as defined in Figure 5.2(a), saying that if there is a blue block that is not on a big truck then value 1 is assigned. Figure 5.2(b) gives the TVD for $On(B,T)$ under action $loadS$, in which c is

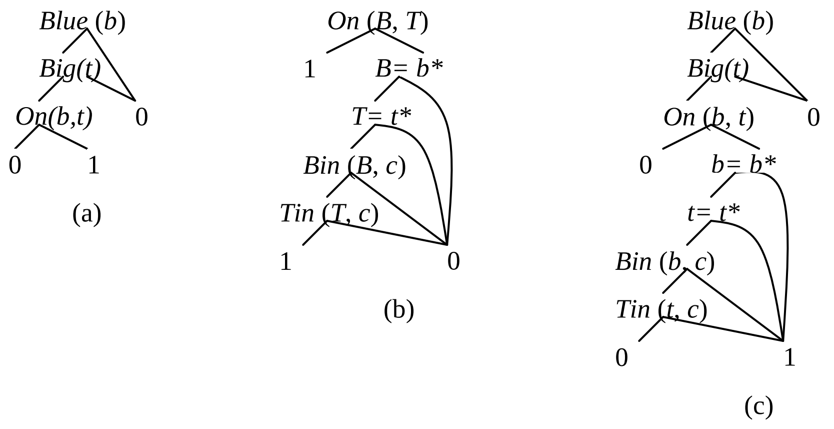


Figure 5.2: An example illustrating why variables are not allowed in TVDs.

a variable instead of an action parameter. Figure 5.2(c) gives the result after block replacement. Consider an interpretation $\hat{s}$ with domain $\{b_1, t_1, c_1, c_2\}$ and relations $\{Blue(b_1), Big(t_1), Bin(b_1, c_1), Tin(t_1, c_1)\}$. After the action $loadS(b_1, t_1)$ we will reach the state $s = \{Blue(b_1), Big(t_1), On(b_1, t_1), Tin(t_1, c_1)\}$, which gives us a value of 0. But Figure 5.2(c) with $b^* = b_1, t^* = t_1$ evaluated in $\hat{s}$ gives value of 1 by valuation $\{b/b_1, c/c_2, t/t_1\}$. Here the choice c/c_2 makes sure the precondition is violated. By making c an action parameter, applying action must explicitly choose valuation and this leads to correct value function. Object maximization turns action parameters into variables and allows us to choose the argument so as to maximize the value.

The naive implementation of block replacement may not be efficient. If we use block replacement for regression then the resulting FODD is not necessarily reduced and moreover, since the different blocks are sorted to start with the result is not even sorted. Reducing and sorting the results may be an expensive operation. Instead we calculate the result as follows. For any FODD V_n we traverse BR-regress($V_n, A(\vec{x})$) using postorder traversal in term of blocks and combine the blocks. At any step we have to combine up to 3 FODDs such that the parent block has not yet been processed (so it is a TVD with binary leaves) and the two children have been processed (so they are general FODDs). If we call the parent B_n, the `true` branch child B_t and the `false` branch child B_f then we can represent their combination as $[B_n \times B_t] + [(1 - B_n) \times B_f]$.

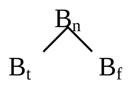


Figure 5.3: A FODD illustrating the idea of block combination

Lemma 12 *Let B be a FODD (as shown in Figure 5.3) where B_t and B_f are FODDs, and B_n is a FODD with $\{0,1\}$ leaves. Let $\hat{B}$ be the result of using Apply to calculate the diagram $[B_n \times B_t] + [(1 - B_n) \times B_f]$. Then for any interpretation I and valuation ζ we have $MAP_B(I,\zeta) = MAP_{\hat{B}}(I,\zeta)$.*

Proof: This is true since by fixing the valuation we effectively ground the FODD and all paths are mutually exclusive. In other words the FODD becomes propositional and clearly the combination using propositional Apply is correct. ■

A high-level description of the algorithm to calculate BR-regress$(V_n, A(\vec{x}))$ by block combination is as follows:

Procedure 5 *1. Perform a topological sort on V_n nodes.*

2. In reverse order, for each non-leaf node n (its children B_t and B_f have already been processed), let B_n be a copy of the corresponding TVD, calculate $[B_n \times B_t] + [(1 - B_n) \times B_f]$.

3. Return the FODD corresponding to the first node (i.e., the root).

Notice that different blocks share variables so we cannot perform weak reductions during this process. However, we can perform strong reductions in intermediate steps since they do not change the map for any valuation. After the process is completed we can perform any combination of weak and strong reductions since this does not change the map of the regressed value function.

From the discussion so far we have the following lemma:

Lemma 13 *Given a value function V_n and an action type $A(\vec{x})$, Equation 5.1 correctly calculates the value of executing $A(\vec{x})$ and receiving the terminal value V_n.*

In Section 4.2 we briefly mentioned why the single path semantics does not support value iteration as well as the multiple path semantics, where we get disjunction for free. Now with the explanation of regression, we can use an example to illustrate this. Suppose we have a value function as defined in Figure 5.4(a), saying that if we have a red block in a big city then value 1 is assigned. Figure 5.4(b) gives the result after block replacement under action $unloadS(b^*, t^*)$. However this is not correct. Consider an interpretation $\hat{s}$ with domain $\{b_1, b_2, t_1, c_1\}$ and relations $\{Red(b_2), Blue(b_1), Big(c_1), Bin(b_1, c_1), Tin(t_1, c_1), On(b_2, t_1)\}$. Note that we use the single path semantics. We follow the `true` branch at the root since $\exists b, c, Bin(b, c)$ is true with $\{b/b_1, c/c_1\}$. But we follow the `false` branch at $Red(b)$ since $\exists b, c, Bin(b, c) \wedge Red(b)$ is not satisfied. Therefore we get a value of 0. As we know, we should get a value of 1 instead with $\{b/b_2, c/c_1\}$, but it is impossible to achieve this value in Figure 5.4(b) with the single path semantics. The reason block replacement fails is that the top node decides on the true branch based on one instance of the predicate but we really need all true instances of the predicate to filter into the true leaf of the TVD.

To correct the problem, we want to capture all instances that were true before and not undone and all instances that are made true on one path. Figure 5.4(c) gives one possible way to do it. Here $\cup$ stands for union operator, which takes a union of all substitutions, and we treat union as an edge operation. Note that it is a coordinated operation, i.e., instead of taking the union of the substitutions for b' and b'', c' and c'' separately we need to take the union of the substitutions for (b', c') and (b'', c''). This approach may be possible but it clearly leads to complicated diagrams. Similar complications arise in the context of object maximization. Finally if we are to use this representation then all our procedures will need to handle edge marking and unions of substitutions so this approach does not look promising.

5.1.2 Object Maximization

Notice that since we are handling different probabilistic alternatives of the same action separately we must keep action parameters fixed during the regression process and until

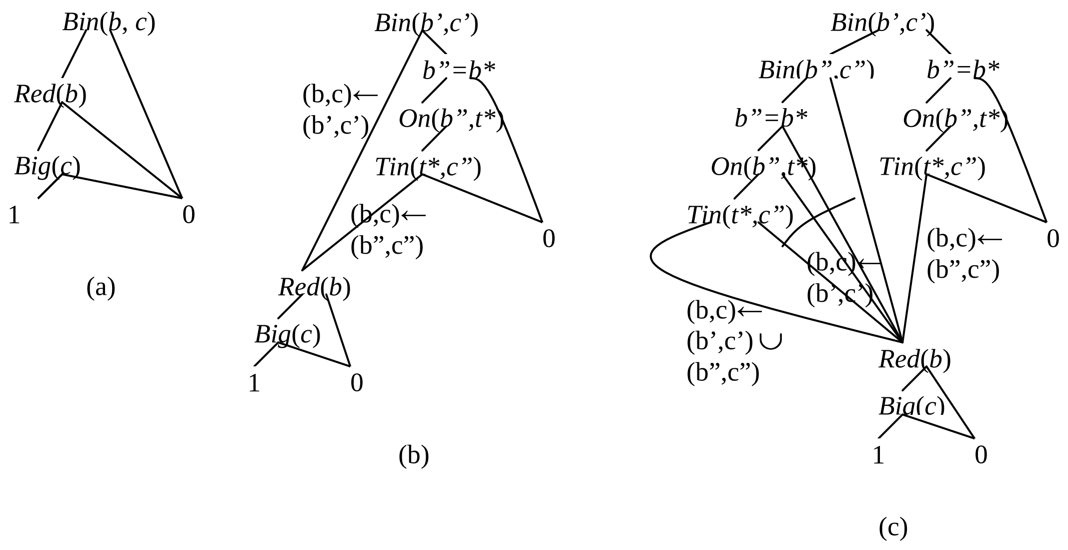


Figure 5.4: An example illustrating union or.

they are added in step 1 of the algorithm. In step 2 we maximize over the choice of action parameters. As mentioned above we get this maximization for free. We simply rename the action parameters using new variable names (to avoid repetition between iterations) and consider them as variables. The aggregation semantics provides the maximization. Since constants are turned into variables additional reduction is typically possible at this stage. Any combination of weak and strong reductions can be used.

From the discussion we have the following lemma:

Lemma 14 *Object maximization correctly computes the maximum value achievable by an action instance.*

5.1.3 Adding and Maximizing Over Actions

Adding and maximizing over actions can be done directly using the Apply procedure. Recall that $prob(A_j(\vec{x}))$ is restricted to include only action parameters and cannot include variables. We can therefore calculate $prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x}))$ in step (1) directly using Apply. However, the different regression results are independent functions so that in the sum $\oplus_j(prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x})))$ we must standardize apart the different regression results before adding the functions (note that action parameters are still considered constants at this stage). Similarly the maximization $V_{n+1} = \max_A Q^A_{n+1}$ in step (3) must first standardize apart the different diagrams. The need to standardize apart complicates the diagrams and often introduces structure that can be reduced. In each of these cases we first use the propositional Apply procedure and then follow with weak and strong reductions.

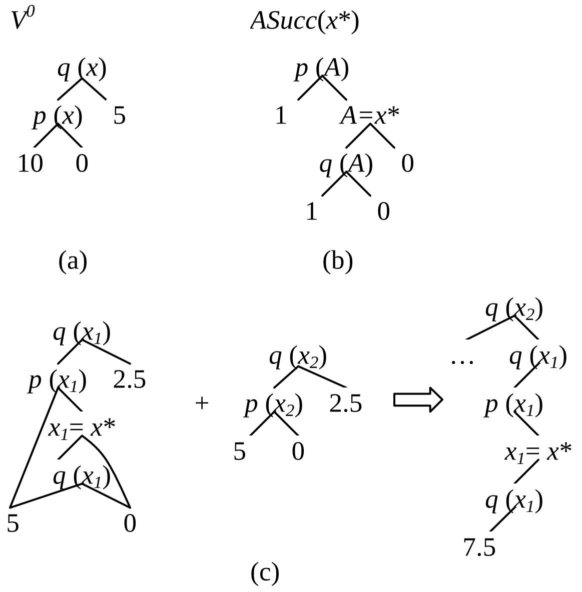


Figure 5.5: An example illustrating the need to standardize apart.

Figure 5.5 illustrates why we need to standardize apart different action outcomes. Action A can succeed (denoted as $ASucc$) or fail (denoted as $AFail$, effectively a no-operation), and each is chosen with probability 0.5. Part (a) gives the value function V^0. Part (b) gives the TVD for $P(A)$ under the action choice $ASucc(x^*)$. All other TVDs are trivial. Part (c) shows part of the result of adding the two outcomes for A after standardizing apart (to simplify the presentation the diagrams are not sorted). Consider an interpretation with domain $\{1, 2\}$ and relations $\{q(1), p(2)\}$. As can be seen from (c), by choosing $x^* = 1$, i.e. action $A(1)$, the valuation $x_1 = 1, x_2 = 2$ gives a value of 7.5 after the action (without considering the discount factor). Obviously if we do not standardize apart (i.e $x_1 = x_2$), there is no leaf with value 7.5 and we get a wrong value. Intuitively the contribution of $ASucc$ to the value comes from the "bring about" portion of the diagram and $AFail$'s contribution uses bindings from the "not undo" portion. Standardizing apart allows us to capture both simultaneously.

5.1.4 Convergence and Complexity

Since each step of Procedure 3 is correct we have the following theorem:

Theorem 4 *If the input to Procedure 3, V_n, correctly captures the value function when there are n steps to go, then the output V_{n+1} correctly captures the value function when there are $n+1$ steps to go.*

Note that for first order MDPs some problems require an infinite number of state partitions. Thus we cannot converge to V^* in a finite number of steps. However, since our algorithm implements VI exactly standard results about approximating optimal value functions and policies still hold. In particular the following standard result (Puterman, 1994) holds for our algorithm, and our stopping criterion guarantees approximating optimal value functions and policies.

Theorem 5 *Let V^* be the optimal value function and let V_k be the value function calculated by the relational VI algorithm.*
(1) If $r(s) \leq M$ for all s then $\|V_n - V^*\| \leq \varepsilon$ for $n \geq \frac{log(\frac{2M}{\varepsilon(1-\gamma)})}{log\frac{1}{\gamma}}$.
(2) If $\|V_{n+1} - V_n\| \leq \frac{\varepsilon(1-\gamma)}{2\gamma}$ then $\|V_{n+1} - V^*\| \leq \varepsilon$.

To analyze the complexity of our VI algorithm notice first that every time we use the Apply procedure the size of the output diagram may be as large as the product of the size of its inputs. While reductions will probably be applicable we cannot guarantee this in advance or quantify the amount of reductions. We must also consider the size of the FODD giving the regressed value function. While Block replacement is $O(N)$ where N is the size of the current value function, it is not sorted and sorting may require both exponential time and space in the worst case. For example, Bryant (1986) gives an example of how ordering may affect the size of a diagram. For a function of $2n$ arguments, the function

$x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{2n-1} \cdot x_{2n}$ only requires a diagram of $2n+2$ nodes, while the function $x_1 \cdot x_{n+1} + x_2 \cdot x_{n+2} + \cdots + x_n \cdot x_{2n}$ requires 2^{n+1} nodes. Notice that these two functions only differ by a permutation of their arguments. Now if $x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{2n-1} \cdot x_{2n}$ is the result of block replacement then clearly sorting requires exponential time and space. The same is true for our block combination procedure or any other method of calculating the result, simply because the output is of exponential size. In such a case heuristics that change variable ordering, as in propositional ADDs (Bryant, 1992), would probably be very useful.

Assuming TVDs, reward function, and probabilities all have size $\leq C$, each action has $\leq M$ action alternatives, the current value function V_n has N nodes, and worst case space expansion for regression and all Apply operations, we can calculate the following upper bound on run time for one iteration. Consider the first step calculating the parameterized Q-function for each action A. For each action alternative A_j of A, regression of V_n through A_j by block combination takes time $O(C^N)$ and has size $O(C^N)$. Multiplying the regression result with the corresponding choice probability requires time $O(C^{N+1})$. Summing over all action alternatives of A takes time $O(C^{M(N+1)})$ and has size $O(C^{M(N+1)})$. The second step, object maximization, just traverses the FODD and renames action parameters using new variable names, and therefore takes time $O(C^{M(N+1)})$. The third step maximizing over all actions takes time $O(C^{M^2(N+1)})$. Therefore the overall time complexity for one iteration is $O(C^{M^2(N+1)})$ given the current value function V_n is of size N. However note that this is the worst case analysis and does not take reduction into account. As the next example illustrates reductions can reduce diagrams substantially and therefore save considerable time in computation.

5.1.5 A Comprehensive Example of Value Iteration

Figure 5.6 traces steps in the application of value iteration to the logistics domain. Note that TVDs, action choice probabilities, and reward function are given in Figure 4.2. We assume the ordering among predicates as $Bin <$ "=" $< On < Tin < rain$.

Given $V_0 = R$ as shown in Figure 5.6(a), **Figure 5.6(b)** gives the result of regression of V_0 through $unloadS(b^*, t^*)$ by block replacement, denoted as $Regr(V_0, unloadS(b^*, t^*))$.

Figure 5.6(c) gives the result of multiplying $Regr(V_0, unloadS(b^*, t^*))$ with the choice probability of *unloadS* $Pr(unloadS(b^*, t^*))$. The result is denoted as $Pr(unloadS(b^*, t^*)) \otimes Regr(V_0, unloadS(b^*, t^*))$.

We perform the same computation for $unloadF(b^*, t^*)$. **Figure 5.6(d)** gives the result of $Pr(unloadF(b^*, t^*)) \otimes Regr(V_0, unloadF(b^*, t^*))$. Notice that this diagram is similar since *unloadF* does not change the state and the TVDs for it are trivial.

Figure 5.6(e) gives the unreduced result of adding two outcomes for $unload(b^*, t^*)$, i.e., $[Pr(unloadS(b^*, t^*)) \otimes Regr(V_0, unloadS(b^*, t^*))] \oplus [Pr(unloadF(b^*, t^*)) \otimes Regr(V_0, unloadF(b^*, t^*))]$. Note that we first standardize apart diagrams for $unloadS(b^*, t^*)$ and $unloadF(b^*, t^*)$ by renaming b in the first diagram as b_1 and b in the second diagram as b_2. Action parameters b^* and t^* at this stage are considered as constants and we do not change them. Also note that the recursive part of Apply (addition $\oplus$) has performed some reductions, i.e., removing the node *rain* when both of its children lead to value 10.

In Figure 5.6(e), for node $Bin(b_2, Paris)$ in the left branch, conditions

P6.1: $\forall b_1, [Bin(b_1, Paris) \rightarrow \exists b_2, Bin(b_2, Paris)]$,

P6.4: b_2 does not appear in the sub-FODD of $Bin(b_2, Paris)_{\downarrow t}$, and

V6.1: $min(Bin(b_2, Paris)_{\downarrow t}) = 10 \geq max(Bin(b_2, Paris)_{\downarrow f}) = 9$

hold. Therefore according to Lemma 2, conditions P6.3 and V6.2 hold, which means we can drop node $Bin(b_2, Paris)$ and connect its parent $Bin(b_1, Paris)$ to its true branch according to Lemma 5. **Figure 5.6(f)** gives the result after this reduction.

Next, consider `true` child of $Bin(b_2, Paris)$ and `true` child of the root. Conditions

P7.1: $[\exists b_1, b_2, \neg Bin(b_1, Paris) \wedge Bin(b_2, Paris)] \rightarrow [\exists b_1, Bin(b_1, Paris)]$,

V7.1: $min(Bin(b_1, Paris)_{\downarrow t}) = 10 \geq max(Bin(b_2, Paris)_{\downarrow t}) = 10$, and

V7.2: $min(Bin(b_1, Paris)_{\downarrow t}) = 10 \geq max(Bin(b_2, Paris)_{\downarrow f}) = 9$

hold. According to Lemma 6 and Lemma 8, we can drop the node $Bin(b_2, Paris)$ and connect its parent $Bin(b_1, Paris)$ to $Bin(b_2, Paris)_{\downarrow f}$. **Figure 5.6(g)** gives the result after this reduction and now we get a fully reduced diagram. This is $T_{V_0}^{unload(b^*, t^*)}$ in

Procedure 4.

In the next step we perform object maximization to maximize over action parameters b^* and t^* and get the best instance of the action *unload*. Note that b^* and t^* have now become variables, and we can perform one more reduction: we can drop the equality on the right branch by the special case of R6. **Figure 5.6(h)** gives the result after object maximization, i.e., obj-max($T_{V_0}^{unload(b^*,t^*)}$). Note that we have renamed the action parameters to avoid the repetition between iterations.

Figure 5.6(i) gives the reduced result of multiplying Figure 5.6(h), obj-max($T_{V_0}^{unload(b^*,t^*)}$), by $\gamma = 0.9$, and adding the reward function. This result is Q_1^{unload}.

We can calculate Q_1^{load} and Q_1^{drive} in the same way and results are shown in **Figure 5.6(j)** and **Figure 5.6(k)** respectively. For *drive* the TVDs are trivial and the calculation is relatively simple. For *load*, the potential loading of a box already in Paris is dropped from the diagram by the reduction operators in the process of object maximization.

Figure 5.6(l) gives V_1, the result after maximizing over Q_1^{unload}, Q_1^{load} and Q_1^{drive}. Here again we standardized apart the diagrams, maximized over them, and then reduced the result. In this case the diagram for *unload* dominates the other actions. Therefore Q_1^{unload} becomes V_1, the value function after the first iteration.

Now we can start the second iteration, i.e., computing V_2 from V_1. **Figure 5.6(m)** gives the result of block replacement in regression of V^1 through action alternative $unloadS(b^*, t^*)$. Note that we have sorted the TVD for $on(B, T)$ so that it obeys the ordering we have chosen. However, the diagram resulting from block replacement is not sorted.

To address this we use the block combination algorithm to combine blocks bottom up. **Figure 5.6(n)** illustrates how we combine blocks $Tin(t, Paris)$, which is a TVD, and its two children, which have been processed and are general FODDs. After we combine $Tin(t, Paris)$ and its two children, $On(b, t)_{\downarrow t}$ has been processed. Since $On(b, t)_{\downarrow f} = 0$, now we can combine $On(b, t)$ and its two children in the next step of block combination.

Continuing this process we get a sorted representation of $Regr(V_1, unloadS(b^*, t^*))$.

5.1.6 Representing Policies

There is more than one way to represent policies with FODDs. We can represent a policy implicitly by a set of regressed value functions. After the value iteration terminates, we can perform one more iteration and compute the set of Q-functions using the Equation 5.1.

Given a state s, we can compute the maximizing action as follows:

1. For each Q-function $Q^{A(\vec{x})}$, compute $\text{MAP}_{Q^{A(\vec{x})}}(s)$, where $\vec{x}$ are considered as variables.

2. For the maximum map obtained record the action name and action parameters (from the valuation) to obtain the maximizing action.

We can also represent a policy explicitly with a special FODD where each leaf is annotated both with a value and with a parameterized action. Note that the values are necessary; a policy with only actions at leaves is not well defined because our semantics does not support state partitions explicitly. Given a state, multiple paths may be traversed, but we only choose an action associated with the maximum value. We will give an example of such a policy in the next section.

Notice that we can extend Procedure 3 to calculate the greedy policy at the same time it calculates the updated value function. To perform this we simply annotate leaves of the original Q functions with the corresponding action schemas. We then perform the rest of the procedure while maintaining the action annotation. We use the notation $(V_{n+1}, \pi) = \text{Rel-greedy}(V_n)$ to capture this extension where V_{n+1} and π refer to the same FODD but V_{n+1} ignores the action information. Since π captures the maximizing action relative to V_n, from Theorem 4 We have:

Lemma 15 *Let* $(V_{n+1}, \pi) =$ *Rel-greedy*(V_n)*, then* $V_{n+1} = Q^{\pi}_{V_n}$.

Figure 5.7 gives the result after performing Rel-greedy(R) in the logistics domain. This corresponds to the value function given in Figure 5.6(l). It is accidental that all the

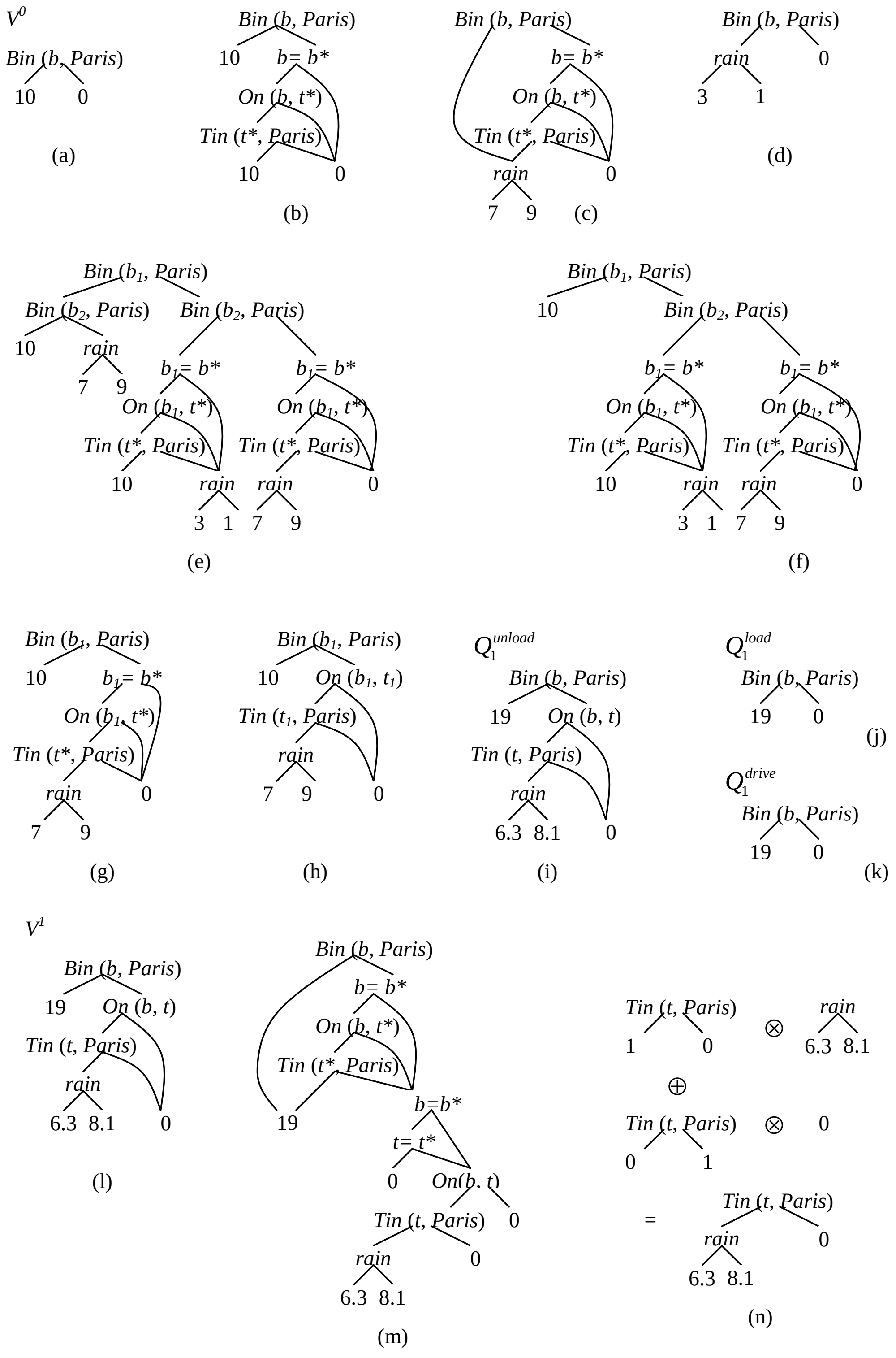


Figure 5.6: An example of value iteration in the Logistics Domain.

leaves have the same action. In general different leaves will be annotated with different actions.

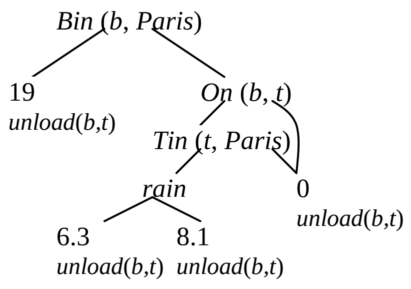


Figure 5.7: The result after performing Rel-greedy(R) in the logistics domain.

5.2 Policy Iteration for Relational MDPs

So far we have discussed how to perform value iteration for relational MDPs. We have also discussed implicit and explicit representations for policies. In this section we show how to perform policy iteration with the explicit policy representation.

5.2.1 The Value Associated with a Policy

In any particular state we evaluate the policy FODD and choose a binding that leads to a maximizing leaf to pick the action. However, note that if multiple bindings reach the same leaf then the choice among them is not specified. Thus our policies are not fully specified. The same is true for most of the work in this area where action choice in policies is based on existential conditions although in practical implementations one typically chooses randomly among the ground actions. Some under-specified policies are problematic since it is not clear how to define their value — the value depends on the unspecified portion, i.e., the choice of action among bindings. For example, consider the blocks world where the goal is $Clear(a)$ and the policy is $Clear(x) \rightarrow MoveToTable(x)$. If x is substituted with a block above a, then it is good; otherwise it will not help reach the goal. We therefore have:

Observation 1 *There exist domains and existential relational policies whose value functions are not well defined.*

Our algorithm below does not resolve this issue. However, we show that the policies we produce have well defined values.

5.2.2 Relational Modified Policy Iteration

Relational Modified Policy Iteration (RMPI) uses the MPI procedure from Section 2.1.2 with the following three changes. 1) We replace Step 2a with $(w^0_{n+1}, \pi^0_{n+1}) =$ Rel-greedy(V_n). 2) We replace the procedure regress-policy with a relational counterpart which is defined next. 3) We replace Step 2(c)i with $(w^{k+1}_{n+1}, \pi^{k+1}_{n+1}) =$ Rel-regress-policy(w^k_{n+1}, π^k_{n+1}). Notice that unlike the original MPI we change the policy which is regressed in every step. The resulting procedure is as follows:

Procedure 6 *Relational Modified Policy Iteration (RMPI)*

```
1. $n = 0$, $V_0 = R$.
2. Repeat
    (a) (Policy improvement)
        $(w^0_{n+1}, \pi^0_{n+1}) =$ Rel-greedy$(V_n)$.
    (b) If $\|w^0_{n+1} - V_n\| \leq \epsilon(1-\gamma)/2\gamma$, return $V_n$ and $\pi$, else go to step 2c.
    (c) (Partial policy evaluation)
        k=0.
        while $k < m_{n+1}$ do
            i. $(w^{k+1}_{n+1}, \pi^{k+1}_{n+1}) =$ Rel-regress-policy$(w^k_{n+1}, \pi^k_{n+1})$.
            ii. k=k+1.
    (d) $V_{n+1} = w^{m_{n+1}}_{n+1}$, $n = n + 1$.
```

Our relational policy regression generalizes an algorithm from (Boutilier et al., 2000) where propositional decision trees were used to solve factored MDPs.

Procedure 7 *Rel-regress-policy*

Input: w^i, π

Output: $w^{i+1}, \hat{\pi}$

1. For each action $A()$ *occurring in* π*, calculate the Q-function for the action type* $A(\vec{x})$*,* $Q_{w^i}^{A(\vec{x})}$*, using Equation (5.1).*

2. At each leaf of π *annotated by* $A(\vec{y})$*, delete the leaf label, and append* $Q_{w^i}^{A(\vec{x})}$ *after (a) substituting the action parameters* $\vec{x}$ *in* $Q_{w^i}^{A(\vec{x})}$ *with* $\vec{y}$*, (b) standardizing apart the Q-function from the policy FODD except for the shared action arguments* $\vec{y}$*, and (c) annotating the new leaves with the action schema* $A(\vec{y})$*.*

3. Return the FODD both as w^{i+1} *and as* $\hat{\pi}$*.*

As in the case of block replacement, a naive implementation may not be efficient since it necessitates node reordering and reductions. Instead we can use an idea similar to block combination to calculate the result as follows. For i'th leaf node L_i in the policy, let $A(\vec{y})$ be the action attached. We replace L_i with 1 and all the other leaf nodes with 0, and get a new FODD F_i. Multiply F_i with $Q_{w^i}^{A(\vec{y})}$ and denote the result as R_i. We do this for each leaf node L_j and calculate R_j. Finally add the results using $\oplus_j R_j$.

Note that all the policies produced by RMPI are calculated by Rel-greedy(V) and Rel-regress-policy(V', π'). Since these procedures guarantee a value achievable by leaf partitions, the maximum aggregation semantics implies that the value of our policies is well defined.

Finally, our notation $(w^{i+1}, \hat{\pi})$ = Rel-regress-policy(w^i, π) suggests that $\hat{\pi}$ may be different from π. This is in contrast with the propositional case (Boutilier et al., 2000) where they are necessarily the same since all the leaves of a single Q-function have the same label and every interpretation follows exactly one path. In our case the policies may indeed be different and this affects the MPI algorithm. We discuss this point at length when analyzing the algorithm.

Concerning the runtime complexity, a similar upper bound as that in value iteration can be calculated for policy iteration under worst case conditions. As for value iteration the bound is exponential in the size of the input formula. However, as the following example illustrates, reductions do save considerable computation time.

5.2.3 A Comprehensive Example of Policy Iteration

We use the following simple domain to illustrate the RMPI algorithm. The domain is deterministic but it is sufficient to demonstrate the crucial algorithmic issues. The domain includes four predicates: $p_1(x)$, $p_2(x)$, $q_1(x)$, $q_2(x)$ and three deterministic actions A_1, A_2, and no-op, where $A_1(x)$ makes $p_1(x)$ true if $q_1(x)$ is true, and $A_2(x)$ makes $p_2(x)$ true if $q_2(x)$ is true. The action "no-op" does not change anything. The reward function, capturing a planning goal, awards a reward of 10 if the formula $\exists x, p_1(x) \wedge p_2(x)$ is true. We assume the discount factor is 0.9 and that there is an absorbing state when $\exists x, p_1(x) \wedge p_2(x)$ holds, i.e., no extra value will be gained once the goal is satisfied. Notice that this is indeed a very simple domain. The optimal policy needs at most two steps using A_1 and A_2 to bring about $p_1()$ and $p_2()$ (if that is possible) to reach the goal. **Figure 5.8(a)** gives the reward function R and the value function V_0. **Figure 5.8(b)** gives the TVD for $p_1(x)$ under action $A_1(x^*)$. **Figure 5.8(c)** gives the TVD for $p_2(x)$ under action $A_2(x^*)$. All the TVDs omitted in the figure are trivial in the sense that the predicate is not affected by the action.

Figure 5.8(d) and **(e)** show parameterized Q-functions $Q_{V_0}^{A_1(x^*)}$ and $Q_{V_0}^{A_2(x^*)}$ as calculated in the first step of value iteration. The Q-function for *no-op* is the same as V_0 since this action causes no change. **Figure 5.8(f)** shows the value function w_1^0 and the policy π_1^0 that is greedy with respect to the reward function R, i.e., $(w_1^0, \pi_1^0) = \text{Rel-greedy}(V_0)$.

Figure 5.8(g) and **(h)** show parameterized Q-functions $Q_{w_1^0}^{A_1(x^*)}$ and $Q_{w_1^0}^{A_2(x^*)}$. $Q_{w_1^0}^{noop}$ is the same as w_1^0. All the steps so far are calculated exactly in the same way as we have illustrated for VI and we therefore omit the details. The next step, *Rel-regress-policy* makes use of these diagrams.

Figure 5.8(i) gives the partial result of *Rel-regress-policy* after appending $Q_{w_1^0}^{A_2(x^*)}$. Notice that we replace a leaf node annotated with action $A_2(x)$ with $Q_{w_1^0}^{A_2(x^*)}$ after the action parameter x^* is substituted with x. We also need to standardize apart the policy

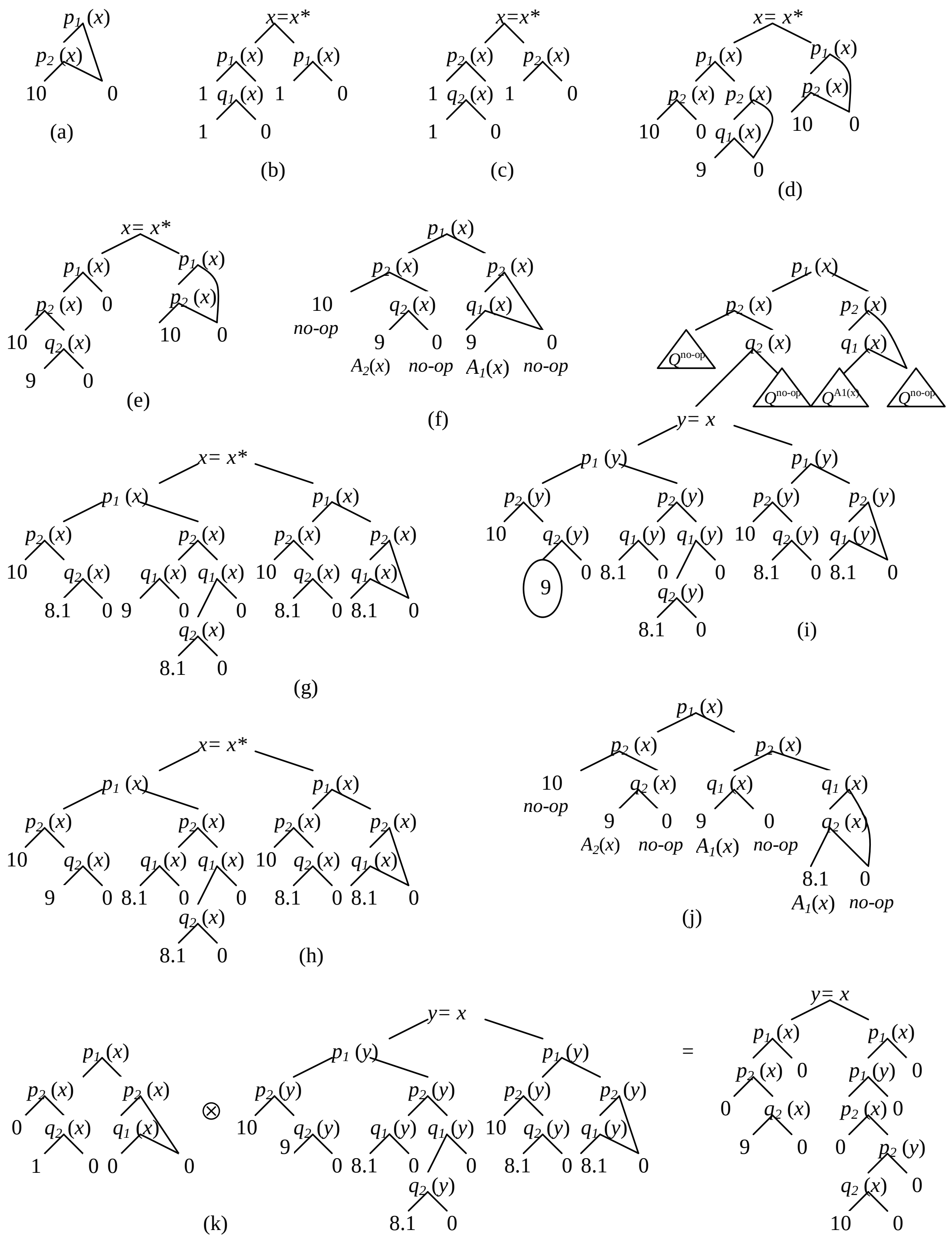


Figure 5.8: An example of policy iteration.

FODD and the Q-function except for the shared action parameter. In this case, we rename x in $Q_{w_1^0}^{A_2(x^*)}$ as y. The same is done for all the leaves but in the figure we omitted the details of other Q-functions and simply wrote Q-function names in the subtrees. It is instructive to review the effect of reductions in this case to see that significant compaction can occur in the process. On the left side of $y = x$, we can replace each y with x, therefore the node predicates are determined and only the leaf valued 9 remains. On the right side of $y = x$, the path leading to 10 will be dominated by the path $p_1(x), p_2(x) \rightarrow 10$ (not shown in the figure), therefore we can replace it with 0. Now we can see that the left side of $y = x$ dominates the other side. Since y is a new variable it is free to take value equal to x. Therefore we can drop the equality and its right side. The same type of reduction occurs for the other Q-functions replaced at the leaves, so that $w_1^1 = w_1^0$. Recall that for our example domain actions are deterministic and they have no "cascaded" effects so there is no use executing the same action twice. It is therefore not surprising that in this case $w_1^1 =$ Rel-regress-policy(w_1^0, π_1^0) is the same as w_1^0.

Figure 5.8(k) illustrates the process of appending the Q-function using block combination. Notice that reductions were used to replace the right branches of $p_1(y)$ and $p_2(y)$ with 0 in the result.

Figure 5.8(j) shows the value function and policy (w_2^0, π_2^0) that are greedy with respect to w_1^1, i.e., V_1. As above to get this diagram we first calculate the Q-functions with respect to w_1^1. The parameterized Q-functions are the same as in Figure 5.8(g) and (h). To get Figure 5.8(j), we first perform object maximization and then maximize over the Q-functions. w_2^0 and π_2^0 are also the optimal value function and policy.

Note that we assume a lexicographical ordering on actions, which determines which action gets chosen if two or more actions give the same value. In this way the policies obtained in the process are unique. In this example, *no-op* is placed first in the ordering. The ordering among actions may affect the policy obtained. If we put A_1 first, then $(\hat{w}_1^0, \hat{\pi}_1^0) =$ Rel-greedy(V_0) will be the one shown in Figure 5.9, where $A_1(x)$ replaced *no-op* on some of the leaves. Furthermore, $\hat{w}_1^1 =$ Rel-regress-policy$(\hat{w}_1^0, \hat{\pi}_1^0)$ is the same as the optimal value function.

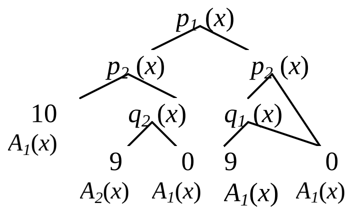


Figure 5.9: The policy $\hat{\pi}_1^0 = \text{Rel-greedy}(V_0)$ if we assume an ordering that puts A_1 first.

For further illustration of policy iteration we also include Figure 5.10 that shows policies and value functions for the example domain without the assumption of an absorbing state.

5.2.4 Correctness and Convergence

The procedure *Rel-regress-policy* is the key step in RMPI. The correctness of MPI relies on the correctness of the regression step. That is, the output of the regression procedure should equal Q_V^π. This is possible when one is using a representation expressive enough to define explicit state partitions that are mutually exclusive, as for example in SDP (Boutilier et al., 2001). But it may not be possible with restricted languages. In the following we show that it is not possible for implicit state partitions used in FODDs and in ReBel (Kersting et al., 2004). In particular, the regression procedure we presented above may in fact calculate an overestimate of Q_V^π. Before we expand on this observation we identify useful properties of the regression procedure.

Lemma 16 *Let* $(w^{i+1}, \hat{\pi}) =$ *Rel-regress-policy*(w^i, π), *then* $w^{i+1} = Q_{w^i}^{\hat{\pi}} \geq Q_{w^i}^{\pi}$.

Proof: $w^{i+1} = Q_{w^i}^{\hat{\pi}}$ holds by the definition of the procedure *Rel-regress-policy*.

Let s be any state; we want to prove $w^{i+1}(s) \geq Q_{w^i}^{\pi}(s)$. We first analyze how $Q_{w^i}^{\pi}(s)$ can be calculated using the diagram of w^{i+1}. Let $A(\vec{x})$ be the maximizing action for s according to the policy π, and ζ a valuation that reaches the leaf node annotated by the action $A(\vec{x})$ in π, i.e., $MAP_\pi(s,\zeta) = max_{\zeta_1}\{MAP_\pi(s,\zeta_1)\}$. Denote the part of ζ that corresponds to action parameters $\vec{x}$ as ζ_A and let $\zeta_{v_1} = \zeta \setminus \zeta_A$.

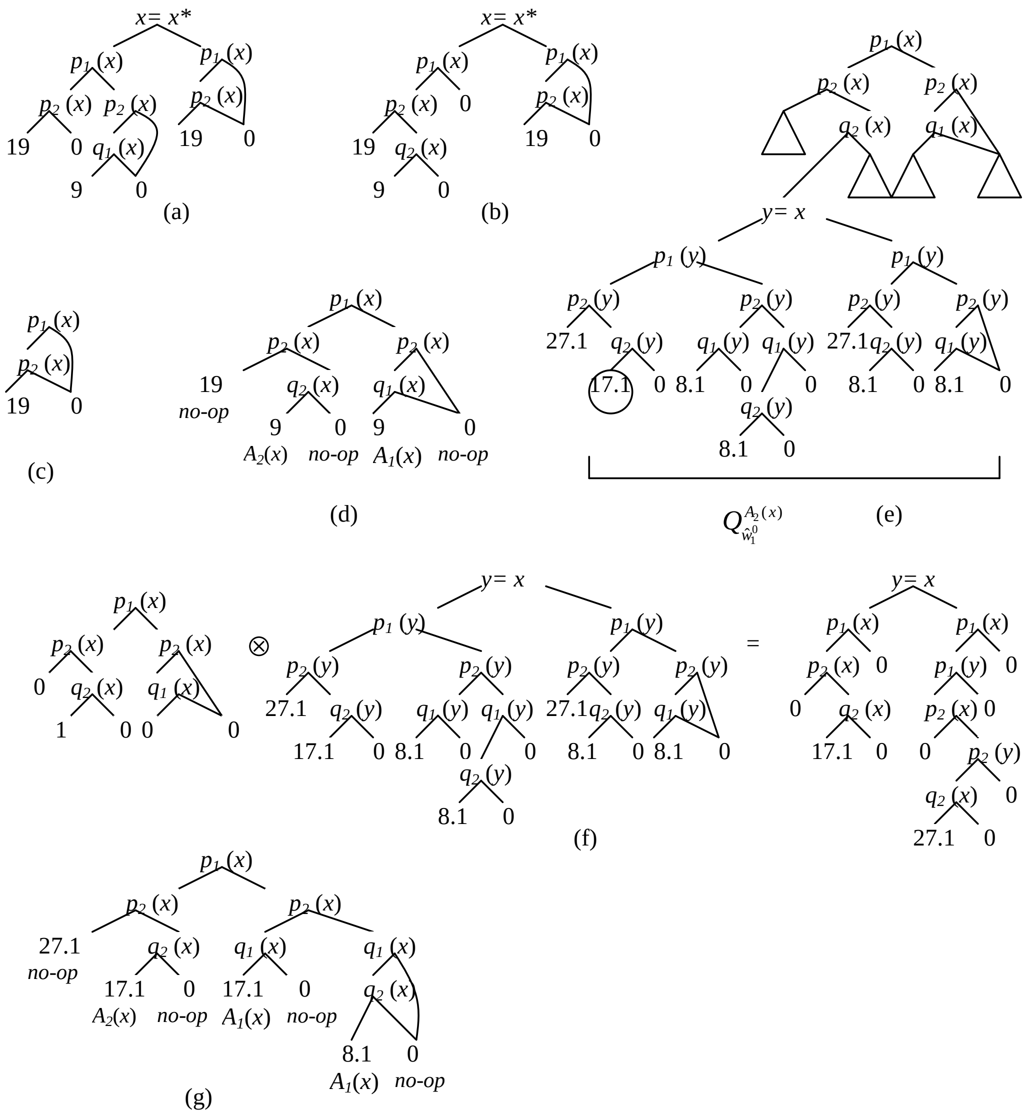


Figure 5.10: An example of policy iteration without the assumption of an absorbing state. (a) $Q_{V_0}^{A_1(x^*)}$. (b) $Q_{V_0}^{A_2(x^*)}$. (c) $Q_{V_0}^{\text{no-op}}$. (d) The value function $\hat{w}_1^0$ and the policy $\hat{\pi}_1^0$ such that $(\hat{w}_1^0, \hat{\pi}_1^0) = \text{Rel-greedy}(V_0)$. (e) An intermediate result when performing Rel-regress-policy$(\hat{w}_1^0, \hat{\pi}_1^0)$. Note that we replace a leaf node annotated with action $A_2(x)$ with $Q_{\hat{w}_1^0}^{A_2(x^*)}$ after the action parameter x^* is substituted with x. (f) Appending $Q_{\hat{w}_1^0}^{A_2(x^*)}$ through *block combination*. Reductions were used to replace the right branches of $p_1(y)$ and $p_2(y)$ with 0 in the result. (g) The result after Rel-regress-policy$(\hat{w}_1^0, \hat{\pi}_1^0)$.

Recall that w^{i+1} is obtained by replacing each leaf node of π with the corresponding Q-function. For state s, ζ will reach the root node of $Q_{w^i}^{A(\vec{x})}$ in w^{i+1}. Note that the variables in π and $Q_{w_i}^{A(\vec{x})}$ are all standardized apart except that they share action parameters $\vec{x}$, therefore the valuation to $\vec{x}$ in $Q_{w_i}^{A(\vec{x})}$ has been fixed by ζ_A. We denote the valuation to all the other variables in $Q_{w^i}^{A(\vec{x})}$ as ζ_{v_2}. The final value we get for $Q_{w^i}^{\pi}(s)$ is $max_{\zeta_{v_2}} Q_{w^i}^{A(\zeta_A)}$. That is, if η is the valuation to variables in w^{i+1} such that $\eta = \zeta + \zeta_{v_2}$ then we have $MAP_{w^{i+1}}(s, \eta) = Q_{w^i}^{\pi}(s)$. Since $w^{i+1}(s)$ is determined by the maximal value any valuation can achieve, we get $w^{i+1}(s) \geq Q_{w^i}^{\pi}(s)$. ■

We next show that the inequality in Lemma 16 may be strict, i.e., we may have $w^{i+1} = Q_{w^i}^{\hat{\pi}} > Q_{w^i}^{\pi}$, and this happens when $\pi \neq \hat{\pi}$. We first give some intuition and then demonstrate the inequality through an example. Figure 5.11 illustrates a scenario where although the policy π says that the action $A_1()$ should be taken if the 10 leaf is reachable in a state s, e.g. to execute action $A_1(1)$ in $\{p(1), \neg p(2) \cdots\}$, the policy $\hat{\pi}$ resulting from Rel-regress-policy(V, π) says that a better value may be achieved if we take action $A_2()$, e.g., $A_2(2)$. Intuitively the decision made by the leaves in part (a) of the diagram is reversed by the leaves in the expanded diagram so the choice of action is changed.

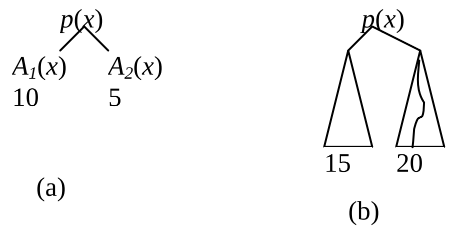


Figure 5.11: A possible scenario. (a) A policy π. (b) The resulting policy $\hat{\pi}$ after regressing over π.

Consider a domain with reward function as shown in Figure 5.1, and with three actions A_1, A_2, and *no-op*. $A_1(x)$ makes $p_1(x)$ true if $q_1(x)$ is true, i.e., $q_1(x) \rightarrow p_1(x)$. $A_2(x)$ makes $p_2(x)$ true if $q_2(x)$ is true, and $p_3(x)$ true if $p_2(x)$ is true, i.e., $q_2(x) \rightarrow p_2(x)$ and $p_2(x) \rightarrow p_3(x)$. Therefore in some states it is useful to do the same action twice. Note that once $p_1(x)$ is true, there is no way to undo it. For example, if we execute action $A_1(1)$

in state $\{q_1(1), \neg p_1(1), p_2(1), \neg p_3(1)\}$, then this state will gain the value of 10 but never be able to reach a better value 20. In the following discussion we use the decision list representation where each rule corresponds to a path in the FODD and rules are ordered by value to capture maximum aggregation. We also assume $\gamma > 0.5$. The following list gives some of the partitions in the policy and the value function $(w_1^0, \pi) = \text{Rel-greedy}(R)$.

$\neg p_1(x) \wedge p_2(x) \wedge p_3(x) \rightarrow 20 + 20\gamma(\text{no-op})$.

$\neg p_1(x) \wedge p_2(x) \wedge \neg p_3(x) \rightarrow 5 + 20\gamma(A_2(x))$.

$\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge p_3(x) \rightarrow 20\gamma(A_2(x))$.

$p_1(x) \rightarrow 10 + 10\gamma(\text{no-op})$.

$q_1(x) \rightarrow 10\gamma(A_1(x))$.

$\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x) \rightarrow 5\gamma(A_2(x))$.

Consider next the calculation of $(w_1^1, \hat{\pi}) = \text{Rel-regress-policy}(w_1^0, \pi)$. To illustrate the result consider the last state partition $\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x)$. After executing $A_2(x)$, this state will transition to the state $\neg p_1(x) \wedge p_2(x) \wedge \neg p_3(x)$ that is associated with the value $5+20\gamma$. Taking discount factor into account, the value for $\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x)$ now becomes $5\gamma + 20\gamma^2$. The following lists gives some of the partitions for $(w_1^1, \hat{\pi})$.

$\neg p_1(x) \wedge p_2(x) \wedge p_3(x) \rightarrow 20 + 20\gamma + 20\gamma^2(\text{no-op})$.

$\neg p_1(x) \wedge p_2(x) \wedge \neg p_3(x) \rightarrow 5 + 20\gamma + 20\gamma^2(A_2(x))$.

$\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge p_3(x) \rightarrow 20\gamma + 20\gamma^2(A_2(x))$.

$p_1(x) \rightarrow 10 + 10\gamma + 10\gamma^2(\text{no-op})$.

$\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x) \rightarrow 5\gamma + 20\gamma^2(A_2(x))$.

$q_1(x) \rightarrow 10\gamma + 10\gamma^2(A_1(x))$.

Note that state partitions $\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x)$ and $q_1(x)$ have now switched places because state partitions are sorted by values in decreasing order. Therefore in this example $\pi \neq \hat{\pi}$. Now suppose we have a state $\{q_1(1), \neg p_1(1), \neg p_2(1), \neg p_3(1), q_2(1)\}$. According to π we should choose $q_1(x) \rightarrow 10\gamma(A_1(x))$ and execute the action $A_1(1)$. But according to $\hat{\pi}$, we should follow $\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x) \rightarrow 5\gamma + 20\gamma^2(A_2(x))$, therefore executing the action $A_2(1)$.

Therefore, due to the use of maximum aggregation, policy evaluation with our representation incorporates an element of policy improvement. It is also important to note that this is not just a result of the procedure we use but in fact a limitation of the representation, in this case because we cannot capture universal quantification. In the example above, to represent the policy, i.e., to force executing the policy even if it does not give the maximal value for a state, we need to represent the state partition $\neg p_1(x) \wedge q_2(x) \wedge \neg p_2(x) \wedge \neg p_3(x) \wedge \neg \exists y, q_1(y) \rightarrow A_2(x)$.

The condition $\neg \exists y, q_1(y)$ must be inserted during policy evaluation to make sure that we evaluate the original policy, i.e., to execute $A_1(x)$ whenever $\exists x, q_1(x)$ is true. This involves universal quantification but our representation is not expressive enough to represent every policy. On the other hand, as we have shown above, our representation is expressive enough to represent each iterate in value iteration as well as the optimal policy when we start with a reward function with existential quantification. This is true because each value function in the sequence includes a set of existential conditions, each showing that if the condition holds then a certain value can be achieved. From the example and discussion we have:

Observation 2 *There exist domains and well defined existential relational policies such that (1) regress-policy cannot be expressed within an existential language, and (2)* $w^{i+1} = Q^{\hat{\pi}}_{w^i} > Q^{\pi}_{w^i}$.

Therefore the obvious question is whether our RMPI algorithm converges to the correct value and policy. Notice that in addition to the overestimate our algorithm differs from MPI in that it uses a different policy in every step of policy evaluation. For the following analysis we have encapsulated all the FODD dependent properties in Lemma 16. The following arguments hold for any representation for which Lemma 16 is true. We show that the overestimate will not exceed the optimal value function, i.e., there is a policy that gives us at least the value in any value function obtained in the policy iteration process. We will also show that the sequence we get is monotonically increasing in value if $R \geq 0$, and that it converges to the optimal value function.

To facilitate the analysis we represent the sequence of value functions and policies from our algorithm as $\{(y^k, \pi^k)\}$ with $(y^0, \pi^0) = (R, \text{no-op})$. Using this notation $(y^k, \pi^k) = \text{Rel-greedy}(y^{k-1})$ when y^k corresponds to w_n^0 for some n, and $(y^k, \pi^k) = \text{Rel-regress-policy}(y^{k-1}, \pi^{k-1})$ where y^k corresponds to w_n^i and $i > 0$.

Lemma 17 *(a)* $y^{n+1} = Q_{y^n}^{\pi^{n+1}} (n \geq 0)$.
(b) $Q_{y^n}^{\pi^{n+1}} \geq Q_{y^n}^{\pi^n}$.

Proof: When $(y^{n+1}, \pi^{n+1}) = \text{Rel-greedy}(y^n)$, then by Lemma 15, $y^{n+1} = Q_{y^n}^{\pi^{n+1}}$. By the property of greedy policies, i.e. $\forall \pi', Q_{y^n}^{\pi^{n+1}} \geq Q_{y^n}^{\pi'}$ we get $Q_{y^n}^{\pi^{n+1}} \geq Q_{y^n}^{\pi^n}$. When $(y^{n+1}, \pi^{n+1}) = \text{Rel-regress-policy}(y^n, \pi^n)$, then by Lemma 16, $y^{n+1} = Q_{y^n}^{\pi^{n+1}} \geq Q_{y^n}^{\pi^n}$. ■

Lemma 18 $\forall k, \exists \tilde{\pi}_k$ *such that* $V^{\tilde{\pi}_k} \geq y^k$.

Proof: The proof is by induction on k. The induction hypothesis is satisfied for k=0 because the policy that always performs *no-op* can achieve value $\geq R$. Assume it holds for $k = 1, 2, \ldots, n$, i.e., $y^k (k = 1, 2, \ldots, n)$ can be achieved by executing some policy $\tilde{\pi}_k$. By Lemma 17 y^{n+1} can be achieved by acting according to π^{n+1} in the first step, and acting according to $\tilde{\pi}_n$ in the next n steps. ■

Note that it follows from this lemma that y^i is achievable by some (possibly non-stationary) policy and since stationary policies are sufficient (Puterman, 1994) we have

Lemma 19 $y^i \leq V^*$, *where* V^* *is the optimal value function.*

Next we prove that the value function sequence is monotonically increasing if $R \geq 0$.

Lemma 20 $\forall i, y^{i+1} \geq y^i$.

Proof: We prove $y^{i+1} \geq y^i$ by induction. When $i = 0$, $(y^1, \pi^1) = \text{Rel-greedy}(y^0)$ and $y^0 = R$.

$$\begin{aligned} y^1(s) &= R(s) + \gamma \sum Pr(s'|s, \pi^1(s)) y^0(s') \\ &\geq R(s) (\text{By } y^0 = R \geq 0) \\ &= y^0(s) \end{aligned}$$

When $i = 1$, $(y^2, \pi^2) = \text{Rel-regress-policy}(y^1, \pi^1)$.

$$\begin{aligned}
y^2(s) &= R(s) + \gamma \sum Pr(s'|s, \pi^2(s))y^1(s') \\
&\geq R(s) + \gamma \sum Pr(s'|s, \pi^1(s))y^1(s') \\
&\quad \text{(By Lemma 16)} \\
&\geq R(s) + \gamma \sum Pr(s'|s, \pi^1(s))y^0(s') \\
&\quad \text{(By the first base case)} \\
&= y^1(s)
\end{aligned}$$

Assume the hypothesis holds for $i = 3, 4, \ldots, n$. Now we want to prove $y^{n+1} \geq y^n$.

$$\begin{aligned}
y^{n+1}(s) &= R(s) + \gamma \sum Pr(s'|s, \pi^{n+1}(s))y^n(s') \\
&\quad \text{(By Lemma 17 (a))} \\
&\geq R(s) + \gamma \sum Pr(s'|s, \pi^n(s))y^n(s') \\
&\quad \text{(By Lemma 17 (b))} \\
&\geq R(s) + \gamma \sum Pr(s'|s, \pi^n(s))y^{n-1}(s') \\
&\quad \text{(By assumption)} \\
&= y^n(s)
\end{aligned}$$

■

It is easy to see that we have the following lemma without the assumption $R \geq 0$.

Lemma 21 *Suppose for some* N *such that* $w_N^0 = \mathit{greedy}(V_{N-1}) \geq V_{N-1}$, *then* $\forall i \geq N, y^{i+1} \geq y^i$.

Let V_i^{VI} denote each iterate in value iteration. Lemma 20 and the fact that if $v \geq u$ then $\hat{v} \geq \hat{u}$ where $\hat{v} = \mathit{greedy}(v)$ and $\hat{u} = \mathit{greedy}(u)$ (Puterman, 1994) imply:

Lemma 22 $V_k \geq V_k^{VI}$.

Now from Lemma 19 and Lemma 22 we see that V_k always gives a better approximation of V^* than V_k^{VI}, therefore the same guarantee as in Theorem 5 hold and we have the following theorem:

Theorem 6 *Let V^* be the optimal value function and let V_k be the value function calculated by the relational MPI algorithm.*
(1) If $r(s) \leq M$ for all s then $\|V_n - V^*\| \leq \varepsilon$ for $n \geq \frac{log(\frac{2M}{\varepsilon(1-\gamma)})}{log\frac{1}{\gamma}}$.
(2) If $\|V_{n+1} - V_n\| \leq \frac{\varepsilon(1-\gamma)}{2\gamma}$ then $\|V_{n+1} - V^*\| \leq \varepsilon$.

Chapter 6

Value Iteration for Relational POMDPs

In this chapter we show how the algorithms for RMDPs using FODDs can be extended to handle Relational POMDPs (RPOMDP). In particular we show how a value iteration algorithm — incremental pruning (Cassandra et al., 1997) can be lifted to the relational case. As will be clear from our discussion, while the planning algorithm is correct our current solution is not complete, since subtle issues (that do not exist in propositional case) arise during planning and execution. These are left as open questions at this stage.

In a domain that is characterized by a relational POMDP, some predicates may be fully observable, while some are not. Some actions are deterministic, while some are stochastic. Some actions change the world while some gather information about the world. But a typical POMDP action both changes the world and gathers information. We use a simple domain to illustrate how relational POMDP domains are formalized as well as the algorithm. The domain includes three predicates $p_1(x)$, $p_2(x)$, and $q_2(x)$ where p_1 and q_2 are fully observable, but p_2 is not. The domain has three actions A_1, A_2, and A_3. $A_1(x)$ deterministically makes $p_1(x)$ true. When attempting $A_2(x)$, a successful version $A_2S(x)$ is executed with probability 0.7 and it makes $p_2(x)$ true, and an unsuccessful version A_2F (effectively a *no-op*) is executed with probability 0.3. $A_3(x)$ is a pure sensing action, i.e., it does not change state of the world, that provides imperfect information about $p_2(x)$ through the observation predicate $q_2(x)$. If $p_2(x)$ is true, then $q_2(x)$ is true with probability 0.9. If $p_2(x)$ is false, then $q_2(x)$ is true with probability 0.2. The reward function, capturing a planning goal, awards a reward of 10 if the formula $\exists x, p_1(x) \wedge p_2(x)$ is true. Note that value functions can include some predicates that are not fully observable (such as $p_2()$) because they are used in combination with the belief state which gives distribution over these predicates. We assume the discount factor is 0.9 and that there is an absorbing state $\exists x, p_1(x) \wedge p_2(x)$, i.e., no extra value will be gained once the goal is satisfied. Notice that this is a very simple domain. Each action has only one effect: it either changes the world or gathers information.

Figure 6.3(a) gives the reward function for this domain and TVDs are given in Figure 6.3(b)(c). All the TVDs omitted in the figure are trivial in the sense that the predicate is not affected by the action.

6.1 Additional Representations for VI in RPOMDPs

In previous chapters we have shown how to model the causal effect of an action, choice probabilities, rewards, and value functions. To perform VI for RPOMDPs, we need to model an action's informational effect, i.e., to specify observation probabilities. We need to represent value functions, which are different from the MDP value functions since they are evaluated over belief states instead of single states. We also give a representation for the belief state and discuss how to calculate its expected value. However our VI algorithm does not use belief states to update a value function, that is, belief states are only used during execution.

6.1.1 Specifying Observations

An observation could be complex and could include quantification. For example, in the logistics domain if the goal $\exists b, Bin(b, Paris)$ is not fully observable, we may have a sensing action which gives such information. Here we only deal with atomic observations and assume that each observation is associated with a subset of the action parameters, i.e., the agent can only have observations about the objects it operates on. In the example, if we execute $A_3(x^*)$, then the observation we get will be $q_2(x^*)$ or $\neg q_2(x^*)$.

We allow different aspects of an observation performed by an action to be modeled separately. We use predicates to specify different aspects of an observation and assume these aspects are independent. A complete observation is the cross product of different aspects. For example, if an observation $O()$ after an action $A(\vec{x})$ has two aspects $p(\vec{y})$ and $q(\vec{z})$ where $\vec{y} \subseteq \vec{x}$ and $\vec{z} \subseteq \vec{x}$, then there are four possible observations, which are $O_1(\vec{y} \cup \vec{z}) : p(\vec{y}) \wedge q(\vec{z}), O_2(\vec{y} \cup \vec{z}) : p(\vec{y}) \wedge \neg q(\vec{z}), O_3(\vec{y} \cup \vec{z}) : \neg p(\vec{y}) \wedge q(\vec{z}), O_4(\vec{y} \cup \vec{z}) : \neg p(\vec{y}) \wedge \neg q(\vec{z})$. We use $|\mathcal{O}_A|$ to denote the number of possible observations associated with an action A. Let n be the number of aspects of an observation for an action, then $|\mathcal{O}_A| = 2^n$.

There are different observation models we could use. One commonly used in the planning context is that a probability associated with an action/observation indicates the likelihood of the observation made by the action being correct, e.g., the observation model used in (Poole, 1997). This model is similar to our model of causal effects of a stochastic action, and thus may provide a unified view of actions. For example, if we have an action A_1 which intends to make p true and there is 0.9 probability that this intention succeeds. Therefore action A_1 has two deterministic alternatives, A_1S (for A_1 successfully fulfills its intention) and A_1F (for A_1 fails in fulfilling its intention), with probability 0.9 and 0.1 respectively. At the same time A_1 has an informational effect, i.e., it can observe p's value but may not be accurate. Again A_1 has two deterministic alternatives in sensing. One is A_1A (for A_1 accurately tells p's value), the other is A_1I (for A_1 inaccurately tells p's value, i.e., gives the opposite value), say with probability 0.7 and 0.3. Therefore if an agent observes that p is true, then one of following situation holds: (1) p is true and A_1A (2) p is false and A_1I. Also there are four deterministic alternatives, which are $A_1S \wedge A_1A$, $A_1F \wedge A_1A$, $A_1S \wedge A_1I$, and $A_1F \wedge A_1I$. with probability 0.63, 0.07, 0.27, and 0.03 respectively.

Another observation model follows the POMDP convention, i.e., the observation probability defines the likelihood of the observation given a state. Recall that the observation probability $Pr(o|s, a)$ in the POMDP refers to the probability of observing o when action a is executed and the *resulting* state is s. Here we make the assumption that the observation probability depends on the future state, but not the current state. This is the model we use in the thesis.

There is a tradeoff between the two approaches. On the one hand, the deterministic alternative approach provides a unified view of action effects. On the other hand, the POMDP approach may provide more compact representation. Let n be the number of aspects of an observation. The POMDP approach can capture 2^n observations compactly (by making the independence assumption) whereas with the deterministic alternative approach we need 2^n deterministic alternatives.

In this thesis, we adopt the POMDP approach. We use FODDs to specify observation probabilities $Prob(O_k(\vec{x}), A(\vec{x}))$. Just as choice probabilities, the conditions cannot contain variables — only action parameters and propositions can appear in the observation probability FODD. Figure 6.3(d) gives the observation probability $Prob(q_2(x^*), A_3(x^*))$. Note that the condition $p_2()$ refers to the truth value of $p_2()$ after the action execution. It says that if $p_2(x^*)$ is true, then $q_2(x^*)$ will be observed with probability 0.9. If $p_2(x^*)$ is false, then $q_2(x^*)$ will be observed with probability 0.2.

If we have more than one observation aspect, we can multiply the probability FODD for each aspect and get the probability of a complete observation. Figure 6.1 gives an example. Note that $Prob(\neg p(x^*), A(x^*)) = 1 \ominus Prob(p(x^*), A(x^*))$, $Prob(\neg q(x^*), A(x^*)) = 1 \ominus Prob(q(x^*), A(x^*))$, $Prob(p(x^*) \wedge q(x^*), A(x^*)) = Prob(p(x^*), A(x^*)) \otimes Prob(q(x^*), A(x^*))$, and $Prob(p(x^*) \wedge \neg q(x^*), A(x^*)) = Prob(p(x^*), A(x^*)) \otimes Prob(\neg q(x^*), A(x^*))$.

Note that if the action provides no feedback, the observation will be nil, obtained with certainty when the action is executed. In this case $|\mathcal{O}_A| = 1$ since a single feedback means no feedback. In the example domain, $|\mathcal{O}_{A_1}| = |\mathcal{O}_{A_2}| = 1$ and $|\mathcal{O}_{A_3}| = 2$.

When different observation aspects are correlated, determining the probability of an observation can be done by representing the dependencies with a Bayesian network. This requires some simple probabilistic reasoning. We will not discuss the details here.

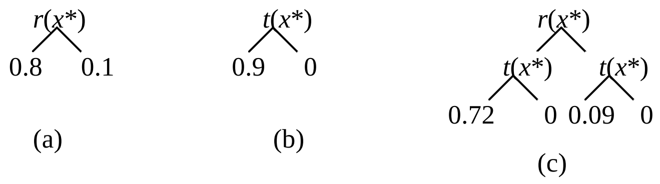


Figure 6.1: Example illustrating observation probabilities: (a) Probability for an observation aspect $p(x^*)$ $Prob(p(x^*), A(x^*))$. (b) Probability for the other observation aspect $q(x^*)$ $Prob(q(x^*), A(x^*))$. (c) Multiply (a)(b) and get the probability $Prob(o_1, A(x^*))$ for a complete observation $o_1 = p(x^*) \wedge q(x^*)$.

6.1.2 Representing Value Functions

The value function for a flat POMDP is piecewise linear and convex and represented by a set of state-value functions $\mathcal{V} = \{v^0, v^1, \cdots, v^n\}$. The value function for the RPOMDP can be represented by a set of FODDs. Each FODD v^i is a value function as in RMDP, except that now v^i contains action parameters accumulated along the decision stages. We will explain the reason for this difference from propositional case later.

6.1.3 Representing Belief States

We do not need to manipulate belief states in the process of calculating the optimal value function. But belief states are needed when we execute the policy and we need to provide an initial belief state for execution. Therefore it is reasonable to assume that a belief state is grounded, i.e., it only contains ground propositions and there is no quantification. However this does not mean that we have to enumerate the world states. Each partition in the belief state can be an abstract state, which includes a set of world states.

To represent a belief state in a compact fashion, we introduce the *sum* notation. $t = sum(\psi)[\phi_1, t_1; \cdots; \phi_n, t_n]$ describes the sum of a function ψ over sets of states, where ϕ_i is a logic formula and the ϕ_i's are disjoint. The interpretation of the above is that $\sum_{s:\phi_i(s)} \psi(s) = t_i$ for all t_i where s is an actual state. In other words, t_i equals the sum of $\psi(s)$ over all the states s in which ϕ_i is true.

We further assume that every state that satisfies the condition contributes equally to the quantity associated with the condition. We call this the *uniform distribution* property for *sum*.

For example, let $Bel(s)$ be the agent's belief that it is in state s, in a domain which includes one predicate $p(x)$ and two objects $\{1,2\}$, then $b = sum(Bel)[p(1), 0.8; \neg p(1), 0.2]$ means that if we add $Bel(s)$ over all the states in which $p(1)$ is true, the sum is 0.8. For $\neg p(1)$, the sum is 0.2. Moreover, these beliefs are uniformly distributed between $p(2)$ and $\neg p(2)$, i.e., $b = sum[p(1) \wedge p(2), 0.4; p(1) \wedge \neg p(2), 0.4; \neg p(1) \wedge p(2), 0.1; \neg p(1) \wedge \neg p(2), 0.1]$. Thus, instead of maintaining a belief state whose size is the number of states, which grows exponentially with the number of propositions, the *sum* notation keeps track of a smaller number of abstract states.

Every proposition for which the agent has no information, represented by having a 0.5 probability of being true, need not appear in the *sum* notation because of its *uniform distribution* property. Hence in the above example, $p(2)$ need not appear. Moreover, states s for which $Bel(s) = 0$ need not appear in a *sum* statement.

We leave ψ out of the sum notation for the remainder of the thesis because it is always the case that $\psi = Bel$.

The sum notation is thus useful in scenarios where (a) there are many propositions for which the agent has no information, or (b) there are many states s for which $Bel(s) = 0$, or both. As the agent acts and observes, the size of the *sum* representation can grow with respect to the number of propositions about which it gains some information that were previously unknown. The growth is polynomial in specific instances but exponential overall in the worst cases. However, as the number of states s where $Bel(s) = 0$ grows, the size of the *sum* representation shrinks.

We can also use ground FODDs as a realization of the *sum* statement. Since there are no variables in the FODD, paths are mutually exclusive. Following the semantics of the *sum* notation, the interpretation of a leaf node t_i in this FODD is that $\sum_{s:s \text{ satisfies } NF(t_i)} Bel(s) = t_i$. In other words, t_i equals the sum of $Bel(s)$ over all the states s in which $NF(t_i)$ is true. Note how the semantics of the belief state FODD is different from the (propositional) FODDs we have seen so far. For the belief state FODD, suppose n is the number of all states that satisfy the leaf node condition, then each state gets t_i/n value, while for the ordinary FODDs, every state that satisfies the condition gets t_i value and we call this type of FODDs "traditional" FODDs.

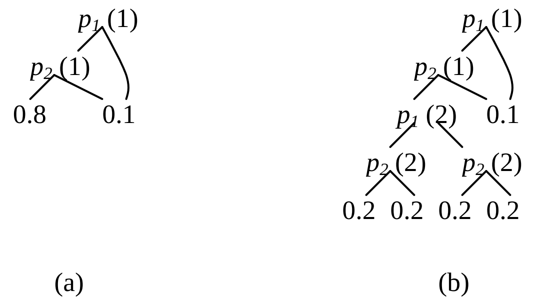


Figure 6.2: An example illustrating the belief state.

Figure 6.2(a) illustrates a belief state FODD in our example domain, which corresponds to the *sum* statement $sum[p_1(1) \wedge p_2(1) : 0.8; p_1(1) \wedge \neg p_2(1) : 0.1; \neg p_1(1) : 0.1]$. Suppose the domain has two objects $\{1, 2\}$, then $p_1(1) \wedge p_2(1) : 0.8$ means that if we add $Bel(s)$ over all the states in which $p_1(1) \wedge p_2(1)$ is true, the sum is 0.8. It can be refined as shown in Figure 6.2(b), which corresponds to the beliefs about the following four actual world states:
$p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge p_2(2) : 0.2$
$p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge \neg p_2(2) : 0.2$
$p_1(1) \wedge p_2(1) \wedge \neg p_1(2) \wedge p_2(2) : 0.2$
$p_1(1) \wedge p_2(1) \wedge \neg p_1(2) \wedge \neg p_2(2) : 0.2$

As we discussed in Chapter 2, if the initial belief state is known, we can do a forward search and only consider belief states that are reachable from the initial belief state. In this case we need to update a belief state after an action and an observation. This could involve two steps: first we determine the effects of a stochastic action without considering the observation received, and then we incorporate the observation and compute the final belief state. It is relatively easy to perform forward update with the *sum* statement. For example, to update the belief state discussed above, $sum[p_1(1) \wedge p_2(1) : 0.8; p_1(1) \wedge \neg p_2(1) : 0.1; \neg p_1(1) : 0.1]$, after $A_1(1)$, we first update each state partition with the action effect of

$A_1(1)$, and get an intermediate result $[p_1(1) \wedge p_2(1) : 0.8; p_1(1) \wedge \neg p_2(1) : 0.1; p_1(1) : 0.1]$. Note that the third partition now changes from $\neg p_1(1)$ to $p_1(1)$. Also note that the resulting states are not disjoint. Therefore we need to perform refinement as needed so that we get a "legal" *sum* statement. In this case, we need to refine $p_1(1) : 0.1$ to $p_1(1) \wedge p_2(1) : 0.05$ and $p_1(1) \wedge \neg p_2(1) : 0.05$. The final belief state after action $A_1(1)$ now becomes $sum[p_1(1) \wedge p_2(1) : 0.85; p_1(1) \wedge \neg p_2(1) : 0.15]$. However, it is not clear how to "push forward" the partitions with FODDs. It seems much more difficult to perform progression than regression over actions for FODDs.

6.1.4 The Expected Value of a Belief State

When we have 0 step to go, the expected value of a belief state given a reward function R is computed using $\sum_s b(s)MAP_R(s) = \sum_s b(s)max_\zeta MAP_R(s, \zeta)$. Recall the details of our example domain as captured in Figure 6.3(a)-(c), and consider the following belief state:
$\neg p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge p_2(2) : 0.09$
$\neg p_1(1) \wedge \neg p_2(1) \wedge p_1(2) \wedge p_2(2) : 0.01$
$\neg p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge \neg p_2(2) : 0.81$
$\neg p_1(1) \wedge \neg p_2(1) \wedge p_1(2) \wedge \neg p_2(2) : 0.09$

Note that we do not have uncertainty about $p_1()$ and the only uncertainty is about the truth value of $p_2()$. In this case we know that $p_1(1)$ is false but $p_1(2)$ is true.

For the first two belief states the valuation $x = 2$ (please refer Figure 6.3(a)) gives the value of 10, and for the next two belief states every valuation leads to a value of 0. Therefore the expected value of this belief state is $0.09 \times 10 + 0.01 \times 10 = 1$.

Consider Q-function $Q^{A(\vec{x})}$ capturing value when we have one step to go. We cannot use the same equation as above to calculate the expected value of a belief state. Note that in the relational domain, two actions are the same iff they have the same action name and parameters. If we calculate as above then it is possible that the maximizing valuations for two different states do not agree on action parameters. Thus if we calculate as above the resulting value assumes we can execute two different actions in the same belief state, based on the actual state we are in, which is clearly wrong.

We illustrate this with an example. Suppose we have the following belief state where we know that both $p_1(1)$ and $p_1(2)$ are false. The Q-function we use is depicted in Figure 6.3(e) where x_1^* is the action parameter.

$\neg p_1(1) \wedge p_2(1) \wedge \neg p_1(2) \wedge p_2(2) : 0.63$

$\neg p_1(1) \wedge \neg p_2(1) \wedge \neg p_1(2) \wedge p_2(2) : 0.27$

$\neg p_1(1) \wedge p_2(1) \wedge \neg p_1(2) \wedge \neg p_2(2) : 0.07$

$\neg p_1(1) \wedge \neg p_2(1) \wedge \neg p_1(2) \wedge \neg p_2(2) : 0.03$

For the first and the fourth state in the belief state, it does not matter what value x_1^* takes. Both $x_1^* = 1$ and $x_1^* = 2$ will give the first state value of 9 and the fourth state value of 0. For the second state, $x_1^* = 2$ will give a better value, which is 9. For the third state, $x_1^* = 1$ will give a better value, which is also 9. Therefore the expected value calculated using a state based formula is wrong because it is based on the best action for each state in a belief state thus the value may not be achievable since we do not know the underlying state.

This issue is also important within the VI algorithm for RPOMDPs. As we discuss later we cannot perform a step of object maximization as we did in RMDP because it will choose a different action per state and not per belief state.

To correctly capture the intended semantics of the value function, we define the expected value of a belief state given a Q-function as

$$Val(Q^{A(\vec{x})}, b) = max_{\zeta_{\vec{x}}} \sum_{s} b(s) max_{\zeta} MAP_Q(s, \zeta) \tag{6.1}$$

where $\zeta_{\vec{x}}$ is valuation to action parameters and ζ is valuation to all the other variables (also called free variables) in the function. Therefore unlike the propositional case, each Q-function is *a function over belief states* instead of individual states. Note that for the belief state discussed above, the expected value given the Q-function in Figure 6.3(e) is 8.73 when $x_1^* = 2$.

As we will show later in the value iteration algorithm, a set of parameterized value functions that make up a value function include all the action parameters accumulated over the decision stages. Given a parameterized value function v^i, the expected value of a belief state is defined as

$$Val(v^i, b) = max_{\zeta_{\vec{x}_i}} \sum_s b(s) max_{\zeta_i} MAP_{v^i}(s, \zeta_i) \tag{6.2}$$

where $\zeta_{\vec{x}_i}$ is valuation to all the action parameters in v^i and ζ_i is valuation to all the other variables in v^i.

The expected value given a set of parameterized value functions $\{v^1, \cdots, v^n\}$ is defined as

$$Val(b) = max_{i \leq n} \{Val(v^i, b)\} \tag{6.3}$$

While Equation 6.2 clearly defines the value assigned to any belief state b it does not provide an efficient algorithm to calculate this value. Recall that in the *sum* statement each state in the belief state can be an abstract state. The combination of abstract states in b with the FODD semantics leads to some complications and at this point we have not identified an efficient algorithm for this task. A correct but inefficient algorithm will refine the belief state so that each state corresponds to a complete world state and calculate directly. The question of efficient evaluation is an important open question in order to utilize our approach and may in fact be an issue with other representations for relational POMDPs.

6.2 Value Iteration for RPOMDPs

We start by reviewing a propositional value iteration algorithm — incremental pruning (Cassandra et al., 1997). This algorithm has been adapted for use with ADDs in the propositional case by Hansen and Feng (2000). We then discuss how to lift it to handle the relational case.

6.2.1 Incremental Pruning

We have seen in Chapter 2 that a POMDP can be considered as a belief state MDP. We have written the Bellman update as follows:

$$V_{n+1}(b) = max_{a\in A}[\sum_{s\in S} b(s)r(s) + \gamma \sum_{o\in O} Pr(o|b,a)V_n(b_o^a)] \tag{6.4}$$

In their development of Incremental pruning, Cassandra et al. (1997) break up the definition of value function V_{n+1} into combinations of simpler value functions.

$$V_{n+1}(b) = max_{a\in A}Q^a(b)$$

$$Q^a(b) = \sum_o Q^{a,o}(b)$$

$$Q^{a,o}(b) = \frac{\sum_{s\in S} r(s)b(s)}{|O|} + \gamma Pr(o|b,a)V_n(b_o^a)$$

Each of the above three value functions is piecewise linear and convex. Therefore we can use a set of state-value functions to represent each value function. That is, each such function, say $w(b)$, can be represented using $\{v^1, \cdots, v^m\}$ for some m where

$$w(b) = max_i \sum_s b(s)v^i(s) \tag{6.5}$$

We use $\mathcal{V}_{n+1}$, $\mathcal{Q}^a$, and $\mathcal{Q}^{a,o}$ to denote the minimum-size set for each value function respectively, and compute them as follows:

$$\mathcal{V}_{n+1} = \text{PRUNE}(\cup_{a\in A}\mathcal{Q}^a) \tag{6.6}$$

$$\mathcal{Q}^a = \text{PRUNE}(\bigoplus_{o\in\mathcal{O}} \mathcal{Q}^{a,o}) \tag{6.7}$$

$$\mathcal{Q}^{a,o} = \text{PRUNE}(\{Q^{a,o,i}|v^i \in \mathcal{V}_n\}) \tag{6.8}$$

where $Q^{a,o,i}$ is defined by

$$Q^{a,o,i} = \frac{R(s)}{|\mathcal{O}|} + \gamma \sum_{s' \in S} Pr(s'|s,a)Pr(o|s',a)v^i(s') \tag{6.9}$$

Equation 6.9 fixes an action and an observation, and associates it with some vector v^i in $\mathcal{V}_n$. That is, we fix a future strategy, and then calculate the expected value of executing action a and when receiving the observation o acting according to the policy encoded in v^i (simply be rewarded by v^i). We call each tuple (a, o, i) an action-observation strategy.

Equation 6.8 collects all action-observation strategies together, one corresponding to each v^i. If we maximize over this set as in Equation 6.5 then we get the best value achievable when taking action a and observing o. We will postpone the explanation of pruning until later. Without pruning the value function for $\mathcal{Q}^{a,o}$ contains $|\mathcal{V}_n|$ state-value functions.

The *cross sum* of two sets of vectors, A and B, is denoted as $A \bigoplus B$ and defined as $\{\alpha + \beta | \alpha \in A, \beta \in B\}$. Equation 6.7 sums together all the value contributions from different observations. The *cross sum* makes sure we consider every possible v^i as a continuation to every observation o. Thus if we maximize over this set using Equation 6.5 we get the best value achievable by taking action a. Notice that each o will lead to a different next state b_a^o where a potentially different action computed in correspondence to v^i can be taken. Without pruning, the value function for $\mathcal{Q}^a$ contains $|\mathcal{V}_n|^{|\mathcal{O}|}$ state-value functions.

Equation 6.7 then puts together the value function for each action. Without pruning, the value function for $\mathcal{V}_{n+1}$ contains $|A||\mathcal{V}_n|^{|\mathcal{O}|}$ state-value functions. This description fits the enumeration algorithm (Monahan, 1982), which will perform pruning at the end of each iteration. The efficiency of the incremental algorithm lies in that it performs pruning whenever possible, as shown in all equations. It also interleaves pruning and cross sum in Equation 6.7 as follows:

$\mathcal{Q}^a = \text{PRUNE}(\ldots(\text{PRUNE}(\mathcal{Q}^{a,o_1} \oplus \mathcal{Q}^{a,o_2}) \ldots \oplus \mathcal{Q}^{a,o_n})$.

6.2.2 Relational VI for POMDPs

The general first order value iteration algorithm works as follows: given as input the reward function R and the action model, we set $\mathcal{V}_0 = R$, n=0, and perform the following steps until termination. Note that the algorithm is presented without pruning, which is discussed in the next section.

Procedure 8 *Rel-VI-POMDP*

1. *For each action type $A(\vec{x})$ and each observation $O_k^{A(\vec{x})}$ associated with the action, compute:*

   For each $v^i \in \mathcal{V}_n$
   $\mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})},i} = \frac{R}{|\mathcal{O}^A|} \oplus \gamma[\sum_j (prob(A_j(\vec{x})) \otimes Regr(Prob(O_k^{A(\vec{x})}, A(\vec{x})), A_j(\vec{x})) \otimes Regr(v^i, A_j(\vec{x})))]$.

2. $\mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})}} = \cup_i \mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})},i}$

3. $\mathcal{Q}^{A(\vec{x})} = \bigoplus_{O_k^{A(\vec{x})}} \mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})}}$.

4. $\mathcal{Q}^A =$ *rename action parameters $\vec{x}$ as special constants in $\mathcal{Q}^{A(\vec{x})}$.*

5. $\mathcal{V}_{n+1} = \cup_A \mathcal{Q}^A$.

Note that when we have an absorbing state (as in our example), the equation in the first step becomes

$$\mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})},i} = \gamma[\sum_j (prob(A_j(\vec{x})) \otimes Regr(Prob(O_k^{A(\vec{x})}, A(\vec{x})), A_j(\vec{x})) \otimes Regr(v^i, A_j(\vec{x})))] \tag{6.10}$$

The equation in the third step becomes

$$\mathcal{Q}^{A(\vec{x})} = max(R, \bigoplus_{O_k^{A(\vec{x})}} \mathcal{Q}^{A(\vec{x}),O_k^{A(\vec{x})}}) \tag{6.11}$$

The first step: corresponds to Equation 6.9. It looks similar to calculating the Q-function for an action type in VI for RMDPs but it also takes observation probabilities

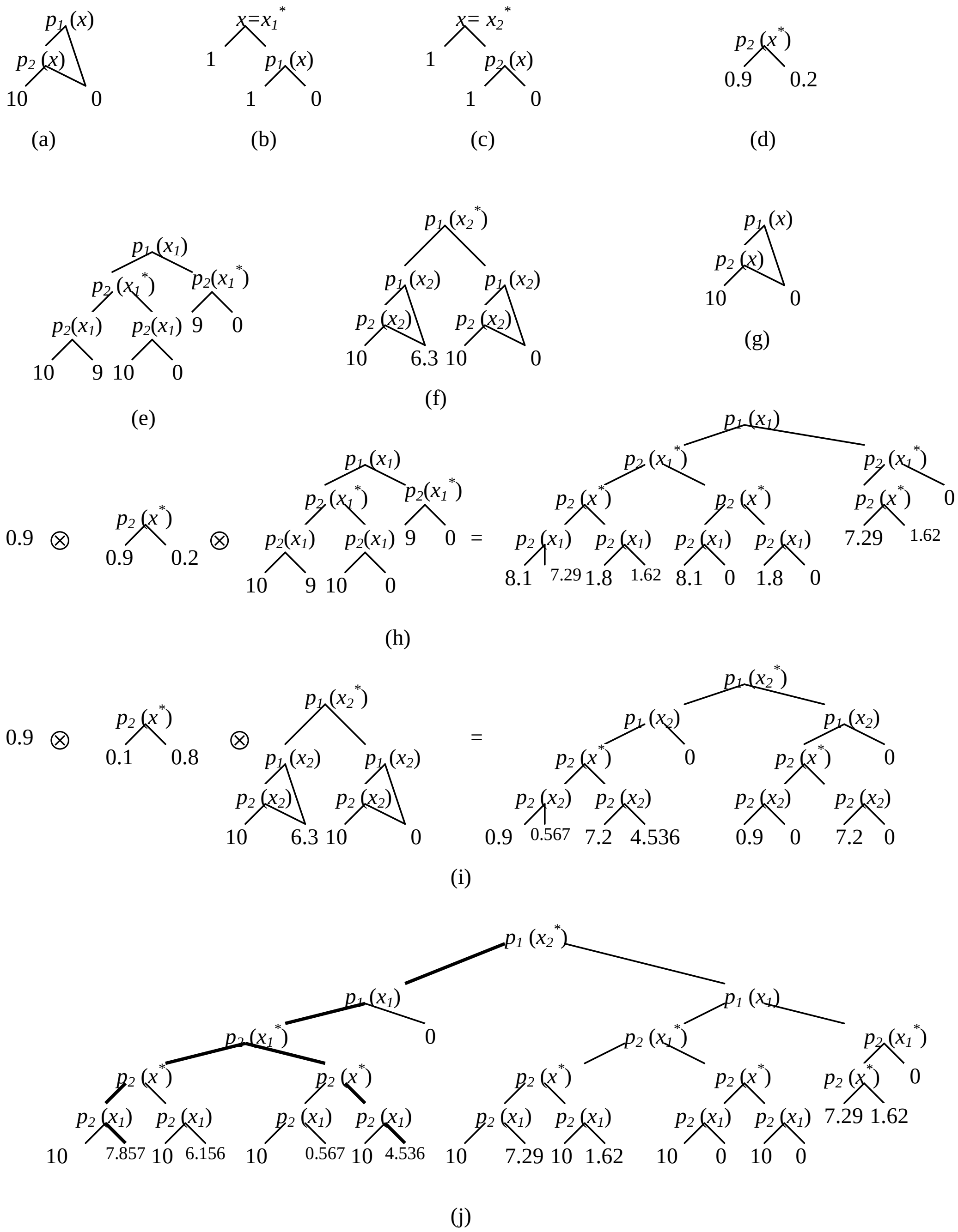


Figure 6.3: An example of value iteration. (a) The reward function R. (b) The TVD for $p_1(x)$ under action $A_1(x_1^*)$. (c) The TVD for $p_2(x)$ under action alternative $A_2S(x_2^*)$. (d) Observation probabilities $Prob(q_2(x^*), A_3(x^*))$. (e) $Q_R^{A_1(x_1^*)}(i.e., v^1)$. (f) $Q_R^{A_2(x_2^*)}(i.e., v^2)$. (g) $Q_R^{A_3(x_3^*)}(i.e., v^3)$. (h) $\mathcal{Q}^{A_3(x^*),q_2(x^*),v^1}$. (i) $\mathcal{Q}^{A_3(x^*),\neg q_2(x^*),v^2}$. (j) One parameterized value function in $\mathcal{Q}^{A_3(x^*)}$.

into account. We need to regress over conditions of observation probabilities because these conditions are about the future state.

If the action has no observation, i.e., $|\mathcal{O}_A| = 1$, the first step can be rewritten as $\mathcal{V}^{A(\vec{x}),i} = R \oplus \gamma[\sum_j (prob(A_j(\vec{x})) \otimes Regr(v^i, A_j(\vec{x})))]$. This is exactly the same as calculating the Q-function for an action type given a terminal value function in VI for RMDPs.

Note that in the first iteration, we do not need to take observations into account. When we have one step to go, the observation after executing the action will not affect the expected value of the action. It is only when determining $\mathcal{V}_2$ and so on that we need to take observations into account and determine what to do next based on the observation obtained.

Figure 6.3(e)(f)(g) give a set of FODDs $\{v^1, v^2, v^3\}$ as the result of the first iteration. We omit details of calculating these but give details of second iteration since it is more informative. Figure 6.3(h)(i) shows $\mathcal{Q}^{A_3(x^*),q_2(x^*),v^1}$ and $\mathcal{Q}^{A_3(x^*),\neg q_2(x^*),v^2}$ respectively, calculated by Equation 6.10. Figure 6.3(h) corresponds to the expected future value of executing action $A_3(x^*)$, and on observing $q_2(x^*)$ following the policy encoded in v^1. Figure 6.3(i) corresponds to the expected future value of executing action $A_3(x^*)$, and on observing $\neg q_2(x^*)$ following the policy encoded in v^2. Note that since $A_3()$ is a pure sensing action and does not change the world, there is only one deterministic alternative, which is *no-op*. Therefore the calculation is simplified since $Prob(A_j) = 1$ and FODDs before and after regression over this action are the same.

The second step: corresponds to Equation 6.8. It groups the resulting FODDs in the first step by the action and the observation.

The third step: corresponds to Equation 6.7. It calculates the cross sum $\bigoplus$ on sets of FODDs for the same parameterized action, collecting the contribution of each observation and its associated next step value function v^i. Note that we use $\bigoplus$ to denote *cross sum*, while use $\oplus$ to denote the addition of two FODDs. Figure 6.3(j) gives the result of $max(R, \mathcal{Q}^{A_3(x^*),q_2(x^*),v^1} \oplus \mathcal{Q}^{A_3(x^*),\neg q_2(x^*),v^2})$, calculated as in Equation 6.11. This gives one of the functions for the action type $A_3(x^*)$. It also encodes a non-stationary 2-step parameterized policy tree as shown in Figure 6.4.

After the third step, we get a set of FODDs including all possible value functions of an action type.

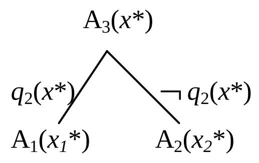


Figure 6.4: The parameterized policy tree corresponding to the value function in Figure 6.3(j)

The fourth step: is an extra step that does not exist in the propositional case. That is because the concept of action parameters does not exist in the propositional case. Recall that so far we have calculated Q-function relative to an action schema parameterized with arguments $\vec{x}$. The fourth step turns the action parameters into special constants for each value function in the set. This is also fundamentally different from RMDPs, which will perform object maximization at this stage to get the maximal value an instance of an action can achieve. However we cannot do the same thing here. We have explained this in the previous section, and we recap it here because it is very important and shows the subtlety of relational POMDPs. Note that for different paths the maximizing action parameters may be different, i.e., maximizing action parameters are based on each (abstract) state. This is fine for RMDPs because we know the current state. But for POMDPs we only have a distribution over possible states and we do not know the current state. We cannot maximize action parameters at planning stage and have to wait until the execution stage to determine what action parameters to choose to produce the maximum *expected* value. If we maximize action parameters at this stage, it would be like calculating the optimal value function V_{MDP} for the underlying MDP, and calculating the expected value of a belief state b by $\sum_s b(s)V_{MDP}(s)$. This is wrong because each $V_{MDP}(s)$ may be associated with different action but we have to use the same action for a belief state.

A value function includes all action parameters accumulated along the value iteration process. This is not convenient, but this is one way to ensure correctness and that the same action is performed for a belief state.

Consider again our running example. In the first iteration when we reach the fourth step, we have Q-functions for $A_1(x_1^*)$, $A_2(x_2^*)$, and $A_3(x_3^*)$ as shown in Figure 6.3(e)(f)(g) and we denote them as $\{v^1, v^2, v^3\}$. Here x_1^* and x_2^* are action parameters for the corresponding Q-functions. Notice that x_3^* was dropped in reducing $Q_R^{A_3(x_3^*)}$. This is simply because $A_3()$ is a pure sensing action and will not have any effect when there is only one step to go. In the second iteration, a value function for $A_3(x^*)$ shown in Figure 6.3(j) contains three action parameters: its own action parameter x^* and two action parameters, x_1^* and x_2^*, "inherited" from its future plan. The action parameters are treated as constants for the purpose of reductions. In evaluation they are treated differently than other variables as prescribed in Equation 6.2.

The fifth step: corresponds to Equation 6.6. It simply puts together the Q-function for each action (which is made up of a set of value function FODDs) and this set forms the updated value function.

6.2.3 Executing Policies

A value function implicitly encodes policies. Each function in our value function $\mathcal{V}_n$ encodes a non-stationary parameterized n step policy tree as shown in Figure 6.5. The root node determine the first action to take, the next action choice depends on the resulting observation. The policy tree decides a sequence of action types to take if we have a sequence of observations. However, we need the initial belief state to know which concrete action to take.

Given a belief state b and the optimal value function expressed as a set of FODDs $\{v^1, \cdots, v^n\}$, we use Equation 6.3 to obtain the expected value of b. At the same time, we get the specific function that gives such value (which is a policy tree), together with the valuation to all action parameters in the policy tree. Therefore we get a complete n step conditional plan for this belief state.

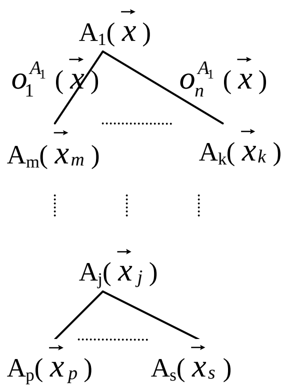


Figure 6.5: A parameterized policy tree

For example if we have an initial belief state b as follows and we have two steps to go:

$\neg p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge p_2(2) : 0.09$

$\neg p_1(1) \wedge \neg p_2(1) \wedge p_1(2) \wedge p_2(2) : 0.01$

$\neg p_1(1) \wedge p_2(1) \wedge p_1(2) \wedge \neg p_2(2) : 0.81$

$\neg p_1(1) \wedge \neg p_2(1) \wedge p_1(2) \wedge \neg p_2(2) : 0.09$

The set of functions which make up $\mathcal{V}_2$ will include the one in Figure 6.3(j), which we denote as v_2^1. First we use Equation 6.2 for each function. In calculating the expected value of b given v_2^1, we find that the valuation $\{x^*/1, x_1^*/1, x_2^*/2\}$ gives the best value. The first two states evaluate to 10 (it does not matter what action parameters to take), the third state evaluates to 7.857, and the fourth state evaluates to 4.536. We have highlighted paths corresponding to the last two states in Figure 6.3(j). The expected value of b given v_2^1 is $0.09 \times 10 + 0.01 \times 10 + 0.81 \times 7.857 + 0.09 \times 4.536 = 7.77241$. Suppose v_2^1 gives the best value among all functions, then we have a conditional plan for b when there are two steps to go, which is the policy tree shown in Figure 6.4 with instantiated action parameters $\{x^*/1, x_1^*/1, x_2^*/2\}$.

From the discussion so far we have:

Theorem 7 *If the input to Procedure 8, $\mathcal{V}_n$, is a collection of Q-functions correctly capturing the value of belief states when there are n steps to go, then the output $\mathcal{V}_{n+1}$ is a collection of Q-functions correctly capturing the value function when there are $n+1$ steps* *to go.*

Note that since we have to keep action parameters along the decision stages, this VI algorithm can only handle finite horizon POMDP problems. Moreover, we do not have pruning and the algorithm is limited to small action sets and observation sets.

6.3 Open Issues

In this section we discuss remaining issues in computing value functions for RPOMDP. We have already discussed the question of efficient calculation of value of a belief state in Section 6.1.4.

6.3.1 Pruning

The pruning step takes a set of FODDs and removes dominated FODDs. The simplest dominance is pointwise, i.e., one FODD gives better value for all states than another. To determine if v pointwise dominate u, we could calculate $t = v - u$. If all the leaf nodes are positive, then we know that this is true. But since subtraction with Apply is essentially propositional, the result is sensitive to variable names. Figure 6.6 illustrates this. Suppose x_1 and x_2 are variables (i.e., not action parameter constants). Intuitively Figure 6.6(a) dominates Figure 6.6(b). But $(a) - (b) \ngeq 0$.

To solve this problem, we develop a "generalized" reduction operator R7. It uses the same set of conditions and performs the same operations. The only difference is that the two edges e_1 and e_2 come from two independent FODDs D_1 and D_2, and all e_1's come from D_1, whereas all e_2's come from D_2. If at the end D_2 is reduced to 0, then we say that the value function D_2 is pointwise dominated by D_1. Note that action parameters are constants during this process.

Figure 6.6(c)-(e) illustrate the process. First we can replace $p(x_2)_{\downarrow t}$ with 0 because if we can reach $p(x_2)_{\downarrow t}$, we can also reach $p(x_1)_{\downarrow t}$, and $min(p(x_1)_{\downarrow t}) \geq max(p(x_2)_{\downarrow t})$. We can replace $p(x_2)_{\downarrow f}$ with 0 in the same way. Finally Figure 6.6(b) is reduced to 0.

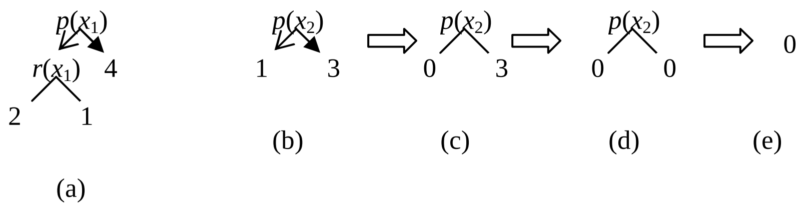


Figure 6.6: An example illustrating how to determine pointwise dominance.

Besides pointwise dominance, a state-value function v can be dominated by a set of state-value functions that does not include v. There is a linear programming method to detect this type of dominance. In the propositionally factored case, Hansen and Feng (2000) first perform a preprocessing step that creates the most refined state partitions that are consistent with state partitions in each state-value function. Then they treat each resulting state partition as a concrete state and perform pruning as in the POMDP that enumerates state space using a linear programming formulation. However they have a much smaller state space because each state actually includes a set of states.

As discussed before in the thesis due to multiple path semantics, the FODD representation does not support explicit state partitions. It is not clear at this stage how we can perform this type of pruning in the relational case and we leave this as an important open question that needs to be resolved before our technique can be applied efficiently for POMDP.

6.3.2 Non-stationary Policy Tree v.s. Stationary Policy Graph

Since we have to treat action parameters as constants and keep action parameters at all decision stages, it is impossible to get a policy graph (finite state controller) as the policy representation. This may be fine for "one shot" task-oriented problems. Here an infinite-horizon problem is more like an indefinite-horizon because the agent will always execute actions for some finite number of stages until reaching a terminal state, though the exact number cannot be determined beforehand. However, for decision problems involving process-oriented behavior where there is no persistent terminal state, it is only intuitive that the policy repeat itself at some point. I.e., the action parameters for the same action should be the same at some point. It is not clear at this stage how to get a stationary parameterized policy graph, and again we leave this as an open question.

6.4 Summary and Discussion

In this chapter, we developed first steps towards solving RPOMDPs. We developed additional representations for value iteration in RPOMDPs and lifted a value iteration algorithm to handle the relational case. We raised several open questions concerning evaluating the expected value of a belief state, pruning, policy graphs, and belief update.

Chapter 7

Discussion and Future Work

We first summarize the contributions of the thesis and then point out some directions for future research.

7.1 Contributions

In this thesis, we describe the use of first order decision diagrams as a new representation for relational MDPs. We give technical details surrounding the semantics of these diagrams and the computational operations required for symbolic dynamic programming. We also introduce Relational Modified Policy Iteration using FODD representations and provide its analysis. We further extend our work to handle relational POMDPs. Specifically, our contributions include the following:

1. We have identified important differences between the single-path and multiple-path semantics for first order decision diagrams. By contrasting the single path semantics with the multiple path semantics we see an interesting tension between the choice of representation and task. The multiple path method does not directly support state partitions, which makes it awkward to specify distributions and policies (since values and actions must both be specified at leaves). However, this semantics simplifies many steps by easily supporting disjunction and maximization over valuations which are crucial for for value iteration so it is likely to lead to significant savings in space

and time.

2. We have developed the technical details of first order ADDs and for the algorithms using them in the solution of relational MDPs. It is non-trivial to lift methods for propositional decision diagrams to the first order case, and our work highlights some of the key semantics and computational issues and proposes solutions. We have also developed novel weak reduction operations for first order decision diagrams and shown their relevance to solving relational MDPs.

3. We have developed a relational value iteration algorithm for MDPs using FODDs. Value iteration for relational MDPs has been studied before, e.g., in SDP and ReBel. Our contribution is that we have developed a calculus of FODDs to implement value iteration, which combines the strong points of the SDP and ReBel approaches. On the one hand we get simple regression algorithms directly manipulating the diagrams so regression is simple as in SDP. On the other hand we get object maximization for free as in ReBel. We also get space saving since different state partitions can share structure in the diagrams.

4. We have developed and analyzed policy iteration in the relational domain. We have introduced Relational Modified Policy Iteration using FODD representations. We have observed that policy languages have an important effect on correctness and potential of policy iteration since the value of a policy may not be expressible in the language. Our algorithm overcomes this problem by including an aspect of policy improvement into policy evaluation. We have shown that the algorithm converges to the optimal value function and policy and that it dominates the iterates from value iteration.

5. We have lifted a value iteration algorithm for propositional POMDPs to handle the relational case. Although the result is not complete, we have made first steps towards solving RPOMDPs, and identified some subtle issues that do not exist in propositional case during planning and execution.

7.2 Future Work and Open Questions

The future agenda of this work includes four major items. First, we want to complete implementation and empirical evaluation. Second, we want to improve the current representation by allowing for more compaction. Third, we want to explore several questions concerning efficiency and alternative algorithmic ideas of RMPI. Fourth, we want to tackle open questions raised in Chapter 6 and further examine how approximation techniques can be implemented in this framework for RPOMDPs.

7.2.1 Implementation and Evaluation

An implementation and empirical evaluation are under way (Joshi, 2007). Any implementation can easily incorporate the idea of approximation by combining leaves with similar values (St-Aubin et al., 2000) to control the size of FODDs. The precise choice of reduction operators and their application will be crucial to obtain an effective system, since in general there is a tradeoff between run time needed for reductions and the size of resulting FODDs. We can apply complex reduction operators to get the maximally reduced FODDs, but it takes longer to perform the reasoning required.

7.2.2 Representations

There are many open issues concerning the current representation. It would be interesting to investigate conditions that guarantee a normal form for a useful set of reduction operators. Also, the representation can be improved to allow further compression. For example it would also be interesting to investigate the effect of allowing edges to rename variables when they are traversed so as to compress isomorphic sub-FODDs. Another interesting possibility is a copy operator that evaluate several copies of a predicate (with different variable) in the same node. Figure 7.1 and Figure 7.2 show how we can get further compression by using such constructs. To be usable one must modify the FODD and MDP algorithmic steps to handle diagrams with the new syntactic notation.

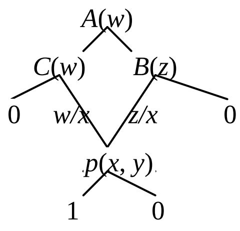


Figure 7.1: Example illustrating variable renaming on edges.

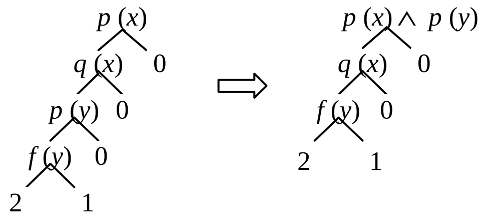


Figure 7.2: Example illustrating the copy operator.

7.2.3 Exploring Efficiency Issues and Alternative Algorithmic Ideas of RMPI

Several interesting questions remain concerning efficiency and alternative algorithmic ideas of RMPI. Some of these can become clearer through an experimental evaluation. First it would be interesting to compare the relational versions of VI and PI in terms of efficiency and convergence speed. In the basic scenario when we enumerate states, an iterate of successive approximation in PI is cheaper than one in VI since we do not calculate and maximize over actions. However, with symbolic representations we regress over each action schema in any case, so the question is whether the specialized regression algorithm presented here is faster than the one doing greedy selection. As illustrated in the example and discussion in Chapter 5, significant pruning of diagrams may occur when regressing a value function over a policy that has the same structure. However, in general for symbolic representations it is not clear from the outset that policy evaluation steps are faster than VI regression steps. A second issue is the stopping criterion. Pure PI stops when the policy does not change, but MPI adopts a value difference as in VI. It is possible that we can detect convergence of the policy at the structural level (by ranking or aggregating actual values), and in this way develop an early stopping criterion. But it is not clear how widely applicable such an approach is and whether we can guarantee correctness. Finally, RMPI uses a different policy in every step of policy evaluation. It would be interesting to analyze the algorithm if the policy is kept fixed in this process.

7.2.4 Open Issues and Approximations in RPOMDPs

Several important questions for RPOMDPs were already raised in Chapter 6, including efficient calculation of the expected value of a belief state, pruning, and representing policy as a finite state controller (or policy graph). Another direction we would like to pursue is approximation. Value iteration for RPOMDP is costly and infeasible except for very simple problems. We want to examine how approximation techniques for propositional POMDPs can be incorporated into our framework and whether they are effective. One possibility is to approximate RPOMDP value functions by a set of linear combination of first order basis functions and to develop a first order generalization of approximation techniques described in (Guestrin et al., 2001) for propositionally factored POMDPs. Sanner and Boutilier (2005; 2006) used similar techniques in the solution of first order MDPs and showed promising results.

Bibliography

Bacchus, F., Halpern, J. Y., & Levesque, H. J. (1999). Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, *111*, 171–208.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., & Somenzi, F. (1993). Algebraic decision diagrams and their applications. *Proceedings of the International Conference on Computer-Aided Design* (pp. 188–191).

Barto, A., Bradtke, S., & Singh, S. (1995). Learning to act using real-time dynamic programming. *IEEE Transactions on Automatic Control*, *34(6)*, 589–598.

Bertoli, P., Cimatti, A., Roveri, M., & Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 473–478).

Blockeel, H., & De Raedt, L. (1998). Top down induction of first order logical decision trees. *Artificial Intelligence*, *101*, 285–297.

Blum, A., & Furst, M. (1995). Fast planning through planning graph analysis. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 1636–1642).

Blythe, J. (1998). *Planning under uncertainty in dynamic domains.* Doctoral dissertation, Carnegie Mellon University.

Bonet, B., & Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. *Proceedings of the International Conference on Artificial Intelligence Planning Systems* (pp. 52–61).

Bonet, B., & Geffner, H. (2001). Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, *14(3)*, 237–252.

Boutilier, C., Dean, T., & Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, *121(1)*, 49–107.

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

Boutilier, C., Friedman, N., Goldszmidt, M., & Koller, D. (1996). Context-specific independence in Bayesian networks. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 115–123).

Boutilier, C., & Poole, D. (1996). Computing optimal policies for partially observable decision processes using compact representations. *Proceedings of the National Conference on Artificial Intelligence* (pp. 1168–1175).

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order MDPs. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 690–700).

Boyen, X., & Koller, D. (1998). Tractable inference for complex stochastic processes. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 33–42).

Brafman, R. (1997). A heuristic variable grid solution method of POMDPs. *Proceedings of the National Conference on Artificial Intelligence* (pp. 76–81).

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *C-35*, 677–691.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, *24*, 293–318.

Cassandra, A. (1998). *Exact and approximate algorithms for partially observable markov decision processes*. Doctoral dissertation, Brown University.

Cassandra, A., Littman, M., & Zhang, N. (1997). Incremental pruning: A simple, fast, exact method for partially observable Markov Decision Processes. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 54–61).

Cheng, H. (1988). *Algorithms for partially observable Markov decision processes*. Doctoral dissertation, University of British Columbia.

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

Drape, D., Hanks, S., & Weld, D. (1994). Probabilistic planning with information gathering and contingent execution. *Proceedings of the International Conference on Artificial Intelligence Planning Systems* (pp. 31–36).

Feng, Z., & Hansen, E. A. (2002). Symbolic heuristic search for factored Markov Decision Processes. *Proceedings of the National Conference on Artificial Intelligence* (pp. 455–460).

Fern, A., Yoon, S., & Givan, R. (2003). Approximate policy iteration with a policy language bias. *International Conference on Neural Information Processing Systems*.

Fikes, R., & Nilsson, N. (1972). A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Forbes, J., Huang, T., Kanazawa, K., & Russell, S. (1995). The BATmobile: towards a bayesian automated taxi. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 1878–1885).

Gardiol, N. H., & Kaelbling, L. P. (2003). Envelop-based planning in relational MDPs. *International Conference on Neural Information Processing Systems*.

Garriga, G., Khardon, R., & Raedt, L. D. (2007). On mining closed sets in multi-relational data. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 804–809).

Geffner, H., & Bonet, B. (1998). High-level planning and control with incomplete information using POMDPs. *Proceedings of Fall AAAI Symposium on Cognitive Robotics.*

Gerevini, A., Bonet, B., & Givan, B. (2006). Fifth international planning competition.

Gretton, C., & Thiebaux, S. (2004). Exploiting first-order regression in inductive policy selection. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 217–225).

Groote, J. F., & Tveretina, O. (2003). Binary decision diagrams for first-order predicate logic. *The Journal of Logic and Algebraic Programming*, *57*, 1–22.

Großmann, A., Hölldobler, S., & Skvortsova, O. (2002). Symbolic dynamic programming within the fluent calculus. *Proceedings of the IASTED International Conference on Artificial and Computational Intelligence.*

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003a). Generalizing plans to new environments in relational MDPs. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 1003–1010).

Guestrin, C., Koller, D., Par, R., & Venktaraman, S. (2003b). Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, *19*, 399–468.

Guestrin, C., Koller, D., & Parr, R. (2001). Solving factored POMDPs with linear value functions. *IJCAI-01 workshop on Planning under Uncertainty and Incomplete Information.*

Hansen, E. (1998). Solving POMDPs by search in policy space. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 211–219).

Hansen, E. A., & Feng, Z. (2000). Dynamic programming for POMDPs using a factored state representation. *Proceedings of the International Conference on Artificial Intelligence Planning Systems* (pp. 130–139).

Hansen, E.A.and Zilberstein, S. (2001). LAO*: a heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, *129*, 35–62.

Hauskrecht, M. (1997). A heuristic variable-grid solution method for POMDPs. *Proceedings of the National Conference on Artificial Intelligence* (pp. 727–733).

Hauskrecht, M. (2000). Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, *13*, 33–94.

Hoey, J., St-Aubin, R., Hu, A., & Boutilier, C. (1999). SPUDD: Stochastic planning using decision diagrams. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 279–288).

Hölldobler, S., & Skvortsova, O. (2004). A logic-based approach to dynamic programming. *AAAI-04 workshop on learning and planning in Markov Processes – advances and challenges*.

Hyafil, N., & Bacchus, F. (2003). Conformant probabilistic planning via CSPs. *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 205–214).

Joshi, S. (2007). Implementing value iteration with FODDs for RMDPs. Private communication.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101*, 99–134.

Kautz, H., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. *Proceedings of the National Conference on Artificial Intelligence* (pp. 1194–1201).

Kersting, K., Otterlo, M. V., & Raedt, L. D. (2004). Bellman goes relational. *Proceedings of the International Conference on Machine Learning*.

Koller, D., & Fratkina, R. (1998). Using learning for approximation in stochastic processes. *Proceedings of the International Conference on Machine Learning* (pp. 287–295).

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, *42(2-3)*, 189–211.

Kushmerick, N., Hanks, S., & Weld, D. S. (1995). An algorithm for probabilistic planning. *Artificial Intelligence*, *76*, 239–286.

Littman, M. (1997). Probabilistic propositional planning: Representations and complexity. *Proceedings of the National Conference on Artificial Intelligence* (pp. 748–754).

Lovejoy, W. (1991). Computationally feasible bounds for partially observed markov decision processes. *Operations Research*, *39*, 162–175.

Madani, O., Hanks, S., & Condon, A. (2003). On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, *147*, 5–34.

Martelli, A., & Montanari, U. (1973). Additive and/or graphs. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 1–11).

Mausam, & Weld, D. (2003). Solving relational MDPs with first-order machine learning. *Proceedings of ICAPS Workshop on Planning under Uncertainty and Incomplete Information.*

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, *21*, 35–55.

McMillan, K. L. (1993). *Symbolic model checking.* Kluwer Academic Publishers.

Monahan, G. (1982). A survey of partially observable Markov decision processes. *Management Science*, *28*, 1–16.

Nilsson, N. J. (1971). *Problem-solving methods in artificial intelligence.* McGraw-Hill, New York.

Onder, N. (1997). *Contingency selection in plan generation.* Doctoral dissertation, University of Pittsburgh.

Papadimitrios, C., & Tsitsiklis, N. (1987). The complexity of Markov decision processes. *Mathematics of Operations Research*, *12(3)*, 441–450.

Penberthy, J., & Weld, D. (1992). UCPOP: A sound, complete, partial order planner for ADL. *Third International Conference on Principles of Knowledge Representation and Reasoning* (pp. 103–114).

Poole, D. (1997). The independent choice logic for modeling multiple agents under uncertainty. *Artificial Intelligence, Special Issue on Economic Principles of Multi-Agent Systems*, *94*, 7–56.

Poupart, P., & Boutilier, C. (2000). Value-directed belief state approximation for POMDPs. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 497–506).

Puterman, M. L. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. Wiley.

Rintanen, J. (2003). Expressive equivalence of formalism for planning with sensing. *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 185–194).

Rivest, R. L. (1987). Learning decision lists. *Machine Learning*, *2*, 229–246.

Sanghai, S., Domingos, P., & Weld, D. (2005). Relational dynamic bayesian networks. *Journal of Artificial Intelligence Research*, *24*, 759–797.

Sanner, S., & Boutilier, C. (2005). Approximate linear programming for first-order MDPs. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence*.

Sanner, S., & Boutilier, C. (2006). Practical linear value-approximation techniques for first-order MDPs. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence*.

Schuurmans, D., & Patrascu, R. (2001). Direct value approximation for factored MDPs. *International Conference on Neural Information Processing Systems* (pp. 1579–1586).

Smallwood, R. D., & Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, *21*, 1071–1088.

Sondik, E. (1971). *The optimal control of partially observable markov decision processes.* Doctoral dissertation, Stanford University.

Sondik, E. (1978). The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operation Research*, *26(2)*, 282–304.

St-Aubin, R., Hoey, J., & Boutilier, C. (2000). APRICODD: Approximate policy construction using decision diagrams. *International Conference on Neural Information Processing Systems* (pp. 1089–1095).

Washington, R. (1996). Incremental markov-model planning. *IEEE International Conference on Tools with Artificial Intelligence* (pp. 41–47).

Washington, R. (1997). BI-POMDP: Bounded, incremental, partially-observable Markov-model planning. *Proceedings of the European Conference on Planning* (pp. 440–451).

Weld, D. S. (1999). Recent advances in AI planning. *AI Magazine*, *20*, 93–123.

Younes, H., & Littman, M. (2004). *PPDDL1.0:an extension to PDDL for expressing planning domains with probabilistic effects* (Technical Report CMU-CS-04-167). Carnegie Mellon University.

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

Zhang, N., Lee, S., & Zhang, W. (1999). A method for speeding up value iteration in partially observable markov decision processes. *Proceedings of the Workshop on Uncertainty in Artificial Intelligence* (pp. 696–703).

Zhang, N. L., & Liu, W. (1997). A model approximation scheme for planning in partially observable stochastic domains. *Journal of Artificial Intelligence Research*, *7*, 199–230.

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

Zhou, R., & Hansen, E. (2001). An improved grid-based approximation algorithm for POMDPs. *Proceedings of the International Joint Conference of Artificial Intelligence* (pp. 707–716).

Zubek, V., & Dietterich, T. (2000). A POMDP approximation algorithm that anticipates the need to observe. *Pacific Rim International Conferences on Artificial Intelligence* (pp. 521–532).

Zubek, V., & Dietterich, T. (2001). Two heuristics for solving POMDPs having a delayed need to observe. *IJCAI Workshop on Planning under Uncertainty and Incomplete Information*.